The NT Insider

The only publication dedicated entirely to Windows® system software development

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

July - August 2010 Digital Edition Volume 17 Issue 2

Writing Filters Is Hard Work: Undocumented DFS & RDR Interactions

The Distributed File System (DFS) is a Microsoft technology that allows for an enterprise to configure a single server with a root share that contains multiple links. To clients, the links appear as regular folders under the root share, however they can point to shares either on the same server system or on any other computer in the organization.

For example, you may access a share from your client machine via the path \\DfsServer\DfsRoot\Documents. This could then end up being a link to an entirely different server, say \\FooServer\Share. Thus, ignoring the other features and options that DFS provides, DFS is effectively a name aliasing technology that allows an administrator to redirect client requests from one machine to another.

Frustratingly, over the years we've regularly run into nagging issues with DFS' interaction with the file system filter drivers that we have written. To the casual observer, one would expect that a file system filter driver writer would care little to nothing about DFS. Aside from the potential aliasing issues that this architecture introduces, which exist in other scenarios anyway, this seems like the sort of thing that should be handled transparently to the client. Unfortunately, due to some architectural and design decisions made in the implementation of XP's DFS client support, dealing with DFS can quickly become a hornet's nest at the bottom of a rat hole. In addition, the Filter Manager mini-filter abstraction attempts to hide some of the implementation details and in the process complicates things even more.

Before we begin, we should note that substantial changes have been made to this space in Vista and thus the discussions in this article only apply to O/S releases prior to Vista unless noted otherwise.

(Continued on page 18)

The NT Insider Goes Digital- Good?

You are reading the first electronic version of *The NT Insider*! We're saving trees, a gazillion dollars in print and mail services and getting the issue into your hands at least 2-3 weeks earlier than the print edition.

Does this medium work for you? Why or why not? What can we do better?

We want to hear from you! Send comments to ntinsider@osr.com...Please!

WDK Community Bug Bash Contest

It's back! After almost ten years since the first "Bug Bash", OSR is back to kickoff a new edition, now the *WDK Community Bug Bash Contest*.

What's a Bug Bash Contest? Well, aren't you going to be glad you asked. It's easy: you find and report bugs in the Windows Driver Kit (WDK). That's it. That's all you have to do.

From there, the bugs will be triaged and reported directly to the Microsoft WDK team—so at a bare minimum, you are helping to get bugs fixed in the WDK.

But wait, there's more! It wouldn't be any fun if we didn't run a contest, where *every valid bug submission earned a prize*, and where all valid bug submissions were judged against one another for eligibility to win one of several really worthwhile *grand* prizes (if you will).

Really, in the end, if you read this publication, the WDK is required for some portion of your job function, and thus improving upon it can only help you.

And hey, you could use a iPod touch, an Visual Studio Ultimate with MSDN subscription, or a netbook to play around with, right?

For more, see the center page spread on pages 16-17.

The NT Insider™

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

http://www.osr.com

Consulting Partners
W. Anthony Mason
Peter G. Viscarola
Executive Editor
Daniel D. Root
Contributing Editors
Mark J. Cariddi
Scott J. Noone
OSR Associate Staff
Consultant At Large
Hector J. Rodriguez
Send Stuff To Us:
email: NtInsider@osr.com

Single Issue Price: $15.00

Inside This Issue:

OSR USB FX2 Learning Kit

Don't forget, the popular OSR USB FX2 Learning Kit is available in the Store at www.osronline.com.

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

The NT Insider—Digital Edition

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

Seeking International Partners & Contacts

OSR is in the process of growing its practice to more effectively address both US and international markets. We'd be interested in hearing from you if your company has experience in Windows internals and kernel-mode development and an interest in leveraging that experience in a partnership with OSR.

Partnership Areas

While OSR is open to discussing partner proposals involving all aspects of the business, currently our focus is on international growth in the following areas:

- Training
- Development
- Toolkits

Regions of Interest

While this is not a partner search that is limited to any specific country or region, of particular interest are partners in India, Greater China and Eastern Europe. If you believe your organization would be a valuable partner, we *do* want to hear from you regardless of the market you serve.

What Constitutes "Partner" Potential?

In order to align with international partners that will be a value-add and credit to the OSR brand, we are most interested in discussing proposals with companies and individuals with the following:

- Technical expertise in Windows internals and kernel mode software development and testing
- Experience in providing services or solutions that involve significant Windows kernel-mode components
- Clear understanding of the markets served
- A history of success in business, accompanied by a long list of happy customers
- Ownership and staff that demonstrate integrity, moral character and strong sense of business ethics
- Effective communication skills both with partners and customers

What does OSR Offer in Return?

Every partnership should strive to be a mutually beneficial relationship. OSR has made its reputation based on definitive expertise in Windows internals and kernel-mode software development. This expertise is provided through the various products and services that are licensed and delivered to a world-wide audience. OSR is offering the opportunity to leverage its expertise, knowledge base, resources, intellectual property, business strategy and revenue, with a partner that brings complimentary value in its country or region.

Next Steps

Please contact us at partners@osr.com if your organization has interest in discussing partner potential with OSR. Tell us a bit about your company, and the general areas in which you might be interested in partnering.

What OSR Students Say

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

"The word I would use to describe the class is - great. I will recommend anyone starting out with device drivers to attend this seminar. For me, being a developer who has never written a driver for Windows, it was a great start."

"This was my first OSR seminar and probably won't be my last. I have always heard good things about them and totally agree. The week was enjoyable, entertaining, informative and I left Munich feeling 100% more confident in tackling more debugging issues."

"[Instructor] is a very good tutor. He presents the material with a (loud and) clear voice, often repeating stuff so one won't miss a thing. His long experience in the field shines through. When he got a question he could often refer to how it was designed and implemented in the Windows source code itself, and thus bring a very clear and understandable answer."

Peter Pontificates:

Pros & Cons of Agile SW Development Methodology

I know that you, dear reader, have come to expect a certain level of sophistication and journalistic integrity when it comes to my monthly pontifications: A gloss of civility, a certain level of refinement, and an even airing of all sides of an issue. Thus, on those very rare occasions when I have a personal opinion to present, I feel the need to make my viewpoint on the issue under discussion extremely clear. And so it is for today's topic: Agile software development methodology.

Let me start, then, by clearly, dispassionately, and logically describing my point of view:

Agile software development methodology sucks ass. Big time. Totally. 100%. Entirely.

Gosh, I hope I've stated that clearly enough.

Let me explain further: I have never, not once, not ever, participated in, seen, or even heard of a project in which Agile works better than any well-organized alternative. Agile is without question the refuge of those with small minds, poor writing skills, and extremely high-levels of tolerance for incomplete functionality that's written in a way that's "good enough" to pass whatever tests happen to exist.

Just to make sure that you understand that I mean to slag the entire, wide-ranging, genre that is Agile, I specifically mean to include Agile permutations like Scrum. Basically, I mean to include any software development methodology that relies on references to barnyard animals, where you stand-up to have meetings, or where you're supposed to account for what you're doing in units of, like, 45 minutes.
My major objections to Agile come from what I view as the unmitigated stupidity of its three most important tenets:

- A maniacal emphasis on "just getting something working" as opposed to thinking something through, designing how it should work, and getting something working correctly.

- Requiring detailed estimates of how long it's going to take to implement some piece of functionality.

- An insane reliance on – and limitation to implementing features for – user stories (AKA scenarios), which yield fragmented feature sets as opposed to designing for well-reasoned comprehensive functionality.

For the life of me, I do not understand why I would ever want to write code for a complex facility before I have had the chance to design that facility and consider how it will interact with everything else in its environment. For sure, *managers* like it. They think designing things is a waste of time anyhow. This reminds me of clients who tell me to "just write some code" to do so-and-so. I work in kernel-mode, boys. I often work on complex pieces of larger systems. It doesn't make a lot of sense to just "write some code" that's "good enough" – when you do that, you get code that doesn't handle corner cases, doesn't work or play well with others, and is generally ill-conceived.

But Agile is *all about* "just writing code." System architecture? Design? In Agile, that's not in the plan. What's in the plan is "code it up and see how it works" and "you can fix it in the next sprint." That's hosed. It's a waste of time. And while I'm all about "plan to write one to throw away", I sorta think that the word "plan" is important there.

(Continued on page 5)

Peer Help? Turn to NTDEV/NTFSD/WINDBG

Writing and debugging Windows system software isn't easy. Sometimes, connecting with the right people at the right time can make the difference. This might be a subject matter expert, or just another driver dev tackling a similar project or problem to yours. Either way, your peers from all over the world gather on the lists hosted at http://www.osronline.com/page.cfm?name=ListServer.

NTDEV—internals and driver development
NTFSD—file system and file system filter development
WINDBG—kernel debugging and crash analysis

Peter Pontificates...

(Continued from page 4)

Ever build a house? You don't start building a house before the architect draws-up the plans. It's the planning phase where you get to consider costs, trade-offs, and how one part of the house will relate to and work with other parts. You probably don't want the garage to open onto the living room. Or maybe you do. But at least you need to think about it. And, having thought it through, you begin building.

Now, wait… wait… wait. Before you go all "that's not flexible" and start screaming "waterfall, waterfall, waterfall" on me: As you progress through your building project, there will be problems, issues, and changes. You need to deal with those. You go back to the plans, you discuss what's changed, and you retro-fit those changes to the plan. Then you resume building.

And so it should be in software engineering. It is vital that you at least design what you want to build, even if this means your manager-troid will have to wait an extra week to see some demo he won't understand anyways. And if you're designing a larger system with multiple interconnected sub-parts, you must define an architecture for that system. As you progress through your architecture, design, and implementation, at any point you can revise the plan, revisit what you're trying to do, and take the time to think through (and document) the consequences of your changes.

And, that brings me to the second thing that makes Agile suck so badly: The whole "estimation" process. Every time somebody insists that I estimate how long it's going to take me to implement some particular functionality, and in Agile it's not uncommon to have to do these estimates with great precision, I realize I am dealing with an idiot. Worse, I realize that I am being asked to lie. And I don't like lying. Well, at least not to most people and not usually very much.

Why is estimating software development time so hard? Duh! Truthfully, *I can't even estimate with good precision how long it'll take me to go to store and get a case of beer*. There could be construction, a traffic jam, the store might be out of the beer I want, I could get a flat tire, I might get distracted and decide I need to get a sandwich before I return home. If I can't estimate how long it'll take me buy a case of beer, how the hell can I possibly estimate how long it's going to take me to implement some functionality that nobody's ever done before, for some device that's just had new silicon fabbed? The whole concept is ludicrous, not to mention insulting.

And don't even get me started on Planning Poker. If you like being treated like a mentally defective 6 year old, you will seriously like Planning Poker. If you haven't heard of Planning Poker, just pray that your manager doesn't discover it. It's the kind of thing manager-troids love, because "it's fun for everybody" and leads to "really good estimates."

On the Agile projects I've worked on, the planning process is just a big "everyone's in on it but the boss" joke:

> **Boss man:** You, copiously large engineering unit over there, how long will it take you to create the huzzy-bub for the what's-a-ma-whosits?
>
> **Me:** Well, boss man, I thought long and hard about this, and after consulting with my team members and the folks in test, I'm confident I can have it done mid-way through Sprint 832.
>
> **Boss man:** Well done! That's what I like, a compliant and focused, if slightly overweight, engineering unit. Do you have milestones for that?
>
> **Me:** Oh, yes, boss man. I do. I definitely do! It'll take 2 days to modify the frobnitz, 3 days to code-up the blortz-fart, 2 days to test the blortz-fart and frobnitz together, 2 days to modify the frazzle-blow, oh… no need to bore you with the details, great one… I'll just put it all in our Online Agile Super Planner, oh mighty master.
>
> **Boss man:** Well done! I like specific milestones. Why can't the rest of you give me nice, clear, specific milestones like the fat guy over there with the ponytail. Whatever his name is.

Of course, I'd written all the code already, during the previous sprint. I just hadn't checked it in. So, during Spring 832, I could update the Online Agile Super Planner to indicate that I hit all my milestones, while I worked on whatever it was I wanted to commit for the *next* sprint.

Don't try to tell me you do anything different, cuz I *know* that's what you do. Everyone I know works this way. When you know what you want to build, you just build it. Then, having already built it, you provide astoundingly precise estimates for when you'll have it done. Then you check in what you've already written according to the estimates you provided. Ka-Ching! Goals and objectives: MET. Conformance to process: PERFECT. Ability to estimate: AWESOME. Ability to deliver: EPIC. Now *give me my bonus*.

And that leads me to the final Agile precept that fries my potato: User Stories. Every time I hear somebody say "I've entered that as a user story" I want to puke. Why? User stories are just that: Stories. They're data. They're not wisdom from the ages. And, the way I see them used in most cases, they're not even necessarily fully representative of what the majority of users want or need to do. When your development process is focused entirely on implementing functionality based on a handful of user stories, you more often than not end up with a random, inconsistent, poorly integrated, mess.

(Continued on page 31)

Getting Better

Virtual Storport Tweaks

Storport is a welcome relief to storage driver writers wishing to develop a driver that exports a virtual device. This article updates the Virtual Storport Miniport Driver that we developed in the earlier three issues of *The NT Insider*.

Another Article?

Well to be honest, when we completed the last article we put the Virtual Storport Miniport Driver code away and expected not to have to revisit it for a long while. However, we were contacted by Mike Berhan of busTRACE Technologies who used their busTRACE 9.0 product (www.bustrace.com) to see how accurate our implementation was in emulating a SCSI adapter/device.

So, How Did We Do?

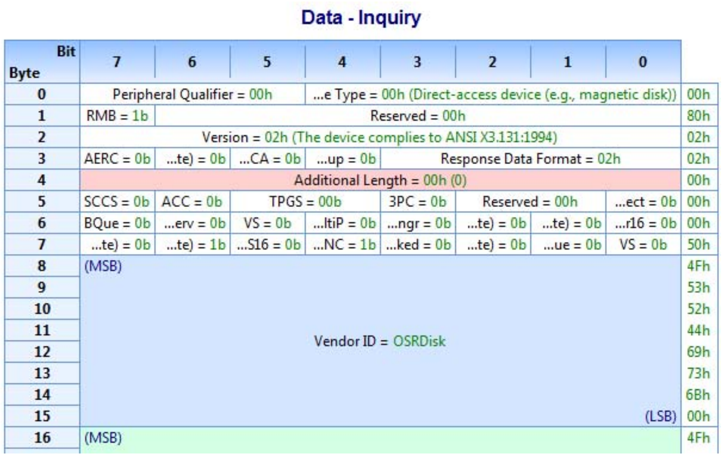
Well, it seems that we missed some stuff. And while windows didn't complain, we'd be doing the development community a disservice if we didn't clean the errors up and indicate where those errors were. So the purpose of this article is to tell you the errors that were found and how they were corrected. Of course, we'll replace the current download code with the updated version.

So let's get into what Mike found, using **busTRACE** and **busPROBE** (which are terrific tools, by the way).

Assumption of Pre-Zeroed Data Buffers

For SCSI operations other than read or write, Mike noted their **busPROBE** product found that the Storport Virtual Miniport driver did not pre-zero the data buffer which will contain the output from the driver. Thus when the driver filled in the return data, and fields that were not explicitly zeroed were filled with random data..

While this may not seem like a big deal, remember that the Storport Virtual Miniport Driver is trying to create and emulate fully compliant SCSI devices, so in order to be compliant the driver was changed to do this.



Data - Inquiry

Byte \ Bit	7	6	5	4	3	2	1	0	
0	Peripheral Qualifier = 00h			...e Type = 00h (Direct-access device (e.g., magnetic disk))					00h
1	RMB = 1b	Reserved = 00h							80h
2	Version = 02h (The device complies to ANSI X3.131:1994)								02h
3	AERC = 0b	...te) = 0b	...CA = 0b	...up = 0b	Response Data Format = 02h				02h
4	Additional Length = 00h (0)								00h
5	SCCS = 0b	ACC = 0b	TPGS = 00b		3PC = 0b	Reserved = 00h		...ect = 0b	00h
6	BQue = 0b	...erv = 0b	VS = 0b	...ltiP = 0b	...ngr = 0b	...te) = 0b	...te) = 0b	...r16 = 0b	00h
7	...te) = 0b	...te) = 1b	...S16 = 0b	...NC = 1b	...ked = 0b	...te) = 0b	...ue = 0b	VS = 0b	50h
8	(MSB) Vendor ID = OSRDisk								4Fh
9									53h
10									52h
11									44h
12									69h
13									73h
14									6Bh
15								(LSB)	00h
16	(MSB)								4Fh

Figure 1—busTRACE INQUIRY Output (abbrev.)

No SRB DataTransferLength Update

The biggest violation that **busTRACE** found was the failure of the Storport Virtual Miniport to update the **DataTransferLength** field in the SRB when completing a command that involved a data transfer. While this isn't critical, in order to be properly compliant with the Microsoft's SRB definition requirement, it must be filled in. Thus, for example, if you diff the old and new code you'll see that **OsrUserReadData** and **OsrUserWriteData** now set the SRB **DataTransferLength** field to the size of the data transfer.

busTRACE also pointed out that when the driver failed a request, it was not setting the SRB's **DataTransferLength** filed to zero to indicate that no data was transferred.

Validating the SRB DataTransferLength

Another interesting **busTRACE** find was that the Storport Virtual Miniport Driver was not validating that the size of the buffer described by the SRBs' **DataTransferLength** field was large enough to hold the data requested. What this means is that when the Storport Virtual Miniport gets a request, for example a read or write, it needs to get the length of the transfer described in the CDB and ensure that the SRB **DataTransferLength** field is large enough. Failure to make this check would lead to crashes or data corruption since the driver could attempt to access past the end of the allocated buffer.

(Continued on page 7)

Getting Better...

(Continued from page 6)

SCSIOP_INQUIRY – Additional Length Field Not Set

busTRACE also flagged the Storport Virtual Miniport Driver for an error processing the **SCSIOP_INQUIRY** command, because the Miniport wasn't filling in the **AdditionalLength** field of the **SCSI_INQUIRY_DATA** that we were returning. The field was being left as zero which is invalid. In the code, it should be set to 0x1F (0x1F plus 5 = 0x24 bytes which is the valid length). In the driver the code sets **AdditionalLength** to "**INQUIRYDATABUFFERSIZE**-5" where **INQUIRYDATABUFFERSIZE** is 36 and the "-5" accounts for the 5 bytes of data (this includes that **AdditionalLength** byte) preceed the additional data returned.

Figure 1 (previous page) shows the busTRACE output that highlights the error *(see Additional* Length in the figure). In the retail version of the product, the user can float the mouse over the red highlighted area and busTRACE tells you what firmware bug it has detected

Figure 2 shows how the **SCSIOP_INQUIRY** processing code was corrected to correct the errors that busTRACE flagged. Notice the setting of the **pInquiryData->AdditionalLength** field as well as the setting of the **PSRB->DataTransferLength** field.

Another SCSIOP_INQUIRY issue that busTRACE found was that the driver was not checking the **EnableVitalProductData** bit (VPD bit) and PAGE CODE field on input. This request, shown in *Figure 3* (page 22), is sent by the Storport driver, after it has enumerated a device in order to enumerate a devices' VPD pages. The Storport Virtual Miniport Driver completely ignored this bit. Since we saw no reason to support this feature the Storport Virtual Miniport driver was changed to fail the SCSIOP_INQUIRY request if the VPD bit and the PAGE CODE field is zero. This was done to be compliant with the SCSI specifications.

Mode Sense Emulation

As with most of the errors in the Virtual Storport Miniport Driver, the error in handling the Mode Sense command was in setting the Mode Data Length field in the returned data. As *Figure 4* (page 22) illustrates the code was returning 0x4 in the Mode Data Length field of the returned data.

(Continued on page 22)

```
case SCSIOP_INQUIRY            : {// 0x12
            PCDB       pCdb = (PCDB) &PSrb->Cdb;
            PUCHAR     pBuffer = (PUCHAR) OsrSpGetSrbDataAddress(pIInfo->OsrSPLocalHandle,PSrb);
            PINQUIRYDATA            pInquiryData;

            if(!pBuffer || PSrb->DataTransferLength < INQUIRYDATABUFFERSIZE) {
                status = STATUS_INSUFFICIENT_RESOURCES;
                //Irp->IoStatus.Information = 0;
                PSrb->SrbStatus = SRB_STATUS_ERROR;
                goto completeRequest;
            }
            pInquiryData = (PINQUIRYDATA) pBuffer;

            //
            // The media is a regular disk, return the correct information.
            //
           ASSERT(pIInfo->StorageType == OsrDisk);

            pInquiryData->DeviceType = DIRECT_ACCESS_DEVICE;
            pInquiryData->DeviceTypeQualifier = DEVICE_CONNECTED;
            pInquiryData->DeviceTypeModifier = 0;
            pInquiryData->RemovableMedia = TRUE;
            pInquiryData->Versions = 2;              // SCSI-2 support
            pInquiryData->ResponseDataFormat = 2;    // Same as Version?? according to SCSI book
            pInquiryData->Wide32Bit = TRUE;          // 32 bit wide transfers
            pInquiryData->Synchronous = TRUE;        // Synchronous commands
            pInquiryData->CommandQueue = FALSE;      // Does not support tagged commands
            pInquiryData->AdditionalLength =
                          INQUIRYDATABUFFERSIZE-5;   // Amount of data we are returning
            pInquiryData->LinkedCommands = FALSE;    // No Linked Commands
            RtlCopyMemory((PUCHAR) &pInquiryData->VendorId[0],OSR_INQUIRY_VENDOR_ID,
                                 strlen(OSR_INQUIRY_VENDOR_ID));
            RtlCopyMemory((PUCHAR) &pInquiryData->ProductId[0],OSR_INQUIRY_PRODUCT_ID,
                                 strlen(OSR_INQUIRY_PRODUCT_ID));
            RtlCopyMemory((PUCHAR)
                   &pInquiryData->ProductRevisionLevel[0],OSR_INQUIRY_PRODUCT_REVISION,
                                   strlen(OSR_INQUIRY_PRODUCT_REVISION));

            status = STATUS_SUCCESS;
            PSrb->SrbStatus = SRB_STATUS_SUCCESS;
            PSrb->DataTransferLength = INQUIRYDATABUFFERSIZE;
            goto completeRequest;
            }
```

Figure 2—SCSIOP_INQUIRY Processing

The Basics of Debugger Extensions

Short Term Effort, Long Term Gain

In order to talk about debugger extensions, we need to first talk about the Debugger Engine library, (DbgEng). DbgEng is a generic interface that can be used to manipulate various *debugging targets*, which can be things such as crash dumps, a live kernel, an application, etc. The library provides access to all of the types of actions that you might want to perform on a debugging target, such as setting or clearing breakpoints, reading or writing memory, translating addresses into symbolic debugging information, receiving events from the target, and so on.

The DbgEng library is quite flexible and can actually be used in standalone applications. If you wrote an application that exported all of the features of the DbgEng library, you'd end up with a debugger exactly like WinDBG or KD because, well, that's what they are. In addition, the DbgEng library is what we're going to talk to when we write our debugger extensions. Thus, in effect we have the same API set but two different uses. This can lead to a fair amount of confusion if you start looking through the WinDBG SDK samples because there's a mix of applications and extensions supplied. If you're not aware of this fact it can be difficult to find a sample to get started with.

Sessions

In either case, DbgEng relies on the concept of *sessions*. In order to examine or manipulate the target, a debugging session must be active. When a debug event is raised from the target, the target may become *acquired* at which point the debug session is active. The application or extension is now allowed to perform whatever options it wants against the target. Once finished with the target, it must be *released*, at which point the session is no longer active and operations are no longer allowed against the target.

Aside from sessions, the other major concept to learn about DbgEng is the set of *objects* that are available to the application or extension. The objects supplied are broken down into two basic categories: client objects and callback objects.

Client Objects

Client objects are used to interact with the debugger engine and provide access to all of the available DbgEng APIs used to manipulate the target via COM interfaces. The way this works is that each client object provides a *QueryInterface* method that is used to create instances of the various COM interfaces. So, for example, if you wanted to set a breakpoint on the target, you'd call the *QueryInterface* method on a client object to retrieve an instance of the COM interface that exports the set breakpoint API. Sounds a little scary, but it's going to turn out to be straightforward so don't worry. The list of available client COM interfaces and their uses are as follows:

- IDebugAdvanced – Thread context, source server, and some symbol information
- IDebugClient – Connect to targets, register input/output callbacks, register debugger event callbacks, etc.
- IDebugControl – Evaluate expressions, disassemble, execute commands, add/remove breakpoints, get input, send output, etc.
- IDebugDataSpaces – Read/write virtual memory, physical memory, I/O ports, MSRs, etc.
- IDebugRegisters – Read/write pseudo and hardware registers
- IDebugSymbols – Type information, module information, source file information, etc.
- IDebugSystemObjects – Thread, process, and processor information

Callback Objects

As their name implies, callback objects export the COM interfaces necessary to receive notifications of events happening on the target. There are three types of callback objects: input objects, output object, and debugger event objects. These allow you to scan the input supplied to the debugger engine from the user as well as the output of any debugger commands or the debugging target. In addition to this, callback objects provide a way to receive notification of process creation, thread creation, breakpoints, etc. The list of available callback COM interfaces and their uses are as follows:

- IDebugInputCallbacks – Receive notification of the engine waiting for user input
- IDebugOutputCallbacks – Receive notification of output being sent to debugger engine
- IDebugEventCallbacks – Receive notification of debug events

Creating Your Own Extension

Now that we have some of the basics down, we can see how to actually create a debugger extension. Debugger extensions are nothing more than DLLs that provide commands via exports and rely on the DbgEng library and any other Win32 API of their choosing. They can be built just like any other DLL, though we'll use the WDK seeing as how that's what the available samples use. The only other thing that you'll

(Continued on page 9)

Debugger Extensions...

(Continued from page 8)

need is the SDK subdirectory of the WinDBG installation, as that's where the necessary header and library files are located.

Extension DLL Entry Points

There is a single required entry point for extension DLLs, *DebugExtensionInitialize*. This is where you will do all of your one time initialization for the extension DLL. A description of the entry point and the function prototype can be found in dbgeng.h:

```
// Initialization routine.  Called once when the
// extension DLL is loaded.  Returns a version and
// returns flags detailing overall qualities of the
// extension DLL. A session may or may not be active
// at the time the DLL is loaded so initialization
// routines should not expect to be able to query
// session information.
typedef HRESULT (CALLBACK*
PDEBUG_EXTENSION_INITIALIZE)
    (__out PULONG Version, __out PULONG Flags);
```

A simple example of a complete DebugExtensionInitialize is as follows:

```
extern "C"
HRESULT
CALLBACK
DebugExtensionInitialize(PULONG Version, PULONG
Flags)
{

    //
    // We're version 1.0 of our extension DLL
    //
    *Version = DEBUG_EXTENSION_VERSION(1, 0);

    //
    // Flags must be zero
    //
    *Flags = 0;

    //
    // Done!
    //
    return S_OK;
}
```

There are two other entirely optional exports that you driver can provide, *DebugExtensionUninitialize* and *DebugExtensionNotify*. DebugExtensionUninitialize can be used to undo anything that you might have done in DebugExtensionInitialize:

```
// Exit routine.  Called once just before the
// extension DLL is unloaded.  As with
// initialization, a session may or may not be
// active at the time of the call.
typedef void (CALLBACK*
PDEBUG_EXTENSION_UNINITIALIZE)
    (void);
```

If present, DbgEng calls the DebugExtensionNotify callback for session state changes:

```
// A debuggee has been discovered for the session.
// It is not necessarily halted.
#define DEBUG_NOTIFY_SESSION_ACTIVE       0x00000000
// The session no longer has a debuggee.
#define DEBUG_NOTIFY_SESSION_INACTIVE     0x00000001
// The debuggee is halted and accessible.
#define DEBUG_NOTIFY_SESSION_ACCESSIBLE   0x00000002
// The debuggee is running or inaccessible.
#define DEBUG_NOTIFY_SESSION_INACCESSIBLE 0x00000003

typedef void (CALLBACK* PDEBUG_EXTENSION_NOTIFY)
    (__in ULONG Notify, __in ULONG64 Argument);
```

The entire point of writing this DLL is to create your own debugger commands, and those will also be exports of your DLL. These commands or, *extensions* must appear in the .DEF file associated with your DLL and the exports must contain only lower case letters. The function prototype of an extension command is as follows:

```
// Every routine in an extension DLL has the
// following prototype. The extension may be called
// from multiple clients so it should not cache the
// client value between calls.
typedef HRESULT (CALLBACK* PDEBUG_EXTENSION_CALL)
    (__in PDEBUG_CLIENT Client, __in_opt PCSTR
Args);
```

You'll note that the function prototype indicates that a pointer to a DEBUG_CLIENT structure is passed as the first parameter to the extension command. This is actually the client object whose *QueryInterface* method you will use to gain access to the various client COM interfaces for target manipulation.

Let's see a simple example command that uses the built in expression evaluator to add two and two together and then displays the result. This would be the equivalent of typing *? 2 + 2* in the WinDBG command prompt.

```
HRESULT CALLBACK
mycommand(PDEBUG_CLIENT4 Client, PCSTR args)
{
    PDEBUG_CONTROL debugControl;
    HRESULT        hr;
    DEBUG_VALUE    result;

    UNREFERENCED_PARAMETER(args);

    //
    // Let's do a couple of simple things. First
    // thing to do is use the passed in client to
    // access the debugger engine APIs.
    //
    // First, we'll get an IDebugControl so that we
    // can print messages.
    //
    hr = Client->QueryInterface
                             (__uuidof(IDebugControl),
                              (void **)&debugControl);

    if (hr != S_OK) {
        return hr;
    }

    //
    // Now we can print.
    //
    debugControl->Output
                             (DEBUG_OUTCTL_ALL_CLIENTS,
                              "mycommand running...\n");

    //
    // Use the evaluator to evaluate an expression
```

(Continued on page 24)

Getting Away From It All

The Isolation Driver (Part I)

In working with a number of file system filter driver projects in recent years we have constructed a model that we refer to as the "isolation" driver – a file system filter (usually a mini-filter) driver that intercepts operations aimed at some set of files and/or directories. The filter changes the content (and potentially the size) of individual files independent of the actual contents of the underlying file. Our goal, in a series of articles, is to describe the basic model, explore development issues and provide a sample mini-filter that implements a model for a data isolation driver. Herein, we first explore common usages, a model, and some high level complications relevant to an isolation driver.

Applications

One model for an isolation filter would be a hierarchical storage manager. Such a component is responsible for moving infrequently used data from online storage to nearline storage devices (traditionally tape, but tape has fallen out of favor in recent years). Another model might be to think of a typical cache system, where one can keep recently used and/or modified data in the cache, while older data is moved to some larger pool of storage.

It turns out that "data isolation" models are actually applicable to a variety of different uses including (but certainly not limited by):

- **Encryption filters** - they want to show a view of the data that is different than the data contained in the file.
- **Compression filters** - they want to show a view of the data that is both different in size AND content from the data contained in the underlying file.
- **Streaming content delivery** - the file starts out zero-filled with data, but information is provided in a "just in time" fashion. For example, some packages might contain huge bodies of information and rather than install all of the data, only the most essential information is installed before the user can begin using the package. The remaining data is then streamed in the background, with priority given to information that is needed by the running application.
- **Data deduplication** – in this case, the file contents are replaced with some sort of reference to the data, thus allowing duplication to be eliminated or removed.
- **Data versioning** – in this model, the original file remains, but any changes that have been made to that file are "layered" on top and the filter driver provides the unified (or snapshot-in-time) view of the file.

The key to each of these is that they work best if the filter controls the actual view of the file in order to ensure that memory mappings of the file work properly.

Basic Model

The basic model for an isolation driver is that it *controls the cache*. This might seem to be more complicated than a "typical" filter. In fact in our experience, any filter that finds itself in the position of trying to manipulate the cache when it is owned by the underlying file system driver becomes *much* more complicated.

(Continued on page 11)

Getting Away...

(Continued from page 10)

So, our initial and forthcoming sample implementation will start with the simplest of data isolation drivers – one that merely replaces the actual contents of the file with the desired contents of the file. In our model, we consider three different "consumers" for our file (see *Figure 1*):

- The *provider* that gives the filter the actual file contents. This *could* be implemented in the filter itself, although we typically find that some (or all) of this functionality is provided by a user mode service or application of some sort. The isolation filter needs to coordinate access with the provider in such cases to ensure that the provider can access the underlying native file.
- The *target* that is the application for which we wish to alter the view. This would normally be some "typical" application, for example Word or Acrobat.
- The *ignored* application for which we do not wish to alter the view. This would normally be some specific application for which we do not want to provide the isolated view. For example, many encryption filters do not decrypt the contents of a file when the application is Windows Explorer.

For our sample, the native file will be the same size as the presented view.

Issues to Consider

Over the years we have seen a number of different filters that try to:

- Flush/purge the underlying file system's cache.
- Acquire locks (typically via the common header) in hopes that the underlying file system uses that specific locking model (a dangerous assumption, in fact).
- Uses undocumented fields in the underlying file system's control blocks to determine caching behavior (this is a problem for encryption filters sitting on the network redirector, since the redirector may change caching policy "in flight").

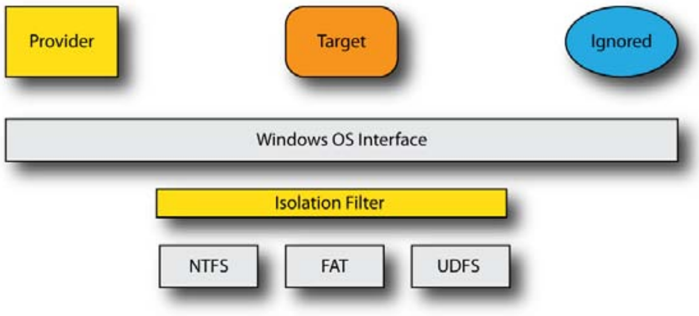


Figure 1—Basic Isolation Filter Model

The model we suggest will work better is to ensure that your filter is *not tied* to the cache that belongs to the underlying file system driver. We achieve this in the isolation model by:

- Utilizing Shadow File Objects; and
- Providing our own Section Object Pointer (SOP) structure

While this might sound unusual, we observe that using a different SOP structure is how NTFS implements transactions on its files – thus a file has a distinct "view" inside the transaction versus outside the transaction.

The other complicating factor here is whether or not your filter needs to modify the *size* of the file. If the file size does not change, you have a simple isolation filter (which is what we'll build in later parts of this series). The complication is if you need to change the size of the file. Why? Because applications will retrieve those sizes via a *directory enumeration* operation and *not* by opening the file and retrieving the size. While it might come as a surprise to some, this really doesn't work 100% of the time for application

(Continued on page 25)

Check This Out

A Primer on Signature Checks in Windows

By Tim Roberts

I find an enormous amount of confusion and a fair amount of misinformation about driver signing in Windows. It's a topic that hasn't traditionally had a lot of attention, because in the past, many of us just ignored driver signing. Those of us who had to learn it did so once, and then used that recipe repeatedly.

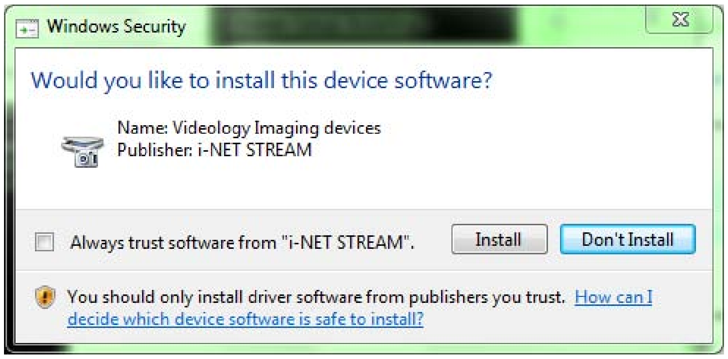
The most important point to realize is that there are two very different signature checks that Windows performs. It's easy to miss this point, as evidenced by many newsgroup and forum postings, but it's important to remember because the two checks require different handling.

Check 1: Signature Check for Trusted Vendor
The first signature check occurs to determine if the vendor supplying the driver is known and trusted. This check occurs once, when your driver is installed. Specifically, the check occurs when your driver is copied into the driver store, or when a new device is detected that requires your INF to be executed. This check happens on all Windows versions since Windows 2000. It occurs on all architectures, so it is *not* one of those checks that only apply to x64-based platforms. However, it's important to note that this check applies *only to plug-and-play devices*.

This signature check only involves the catalog (CAT) file. If you do not have a CAT file, or if your CAT file is not signed, you get the dreaded red-flag "unsigned driver" warning. If your CAT is signed with a non-WHQL signature, you get a "do you trust this publisher" warning shown in *Figure 1* (on Vista and beyond). In either case, the user can simply proceed with the installation with the appropriate mouse click. Once you have passed this check, you can disable, enable, unplug and replug forever, and the signature will not be rechecked, unless there is a new device that matches your INF file. Your CAT can be signed with any valid code-signing certificate.

Check 2: KMCS Check
The second signature check is the kernel-mode code signing (KMCS) check. This happens only on x64 architecture systems, but it happens *every time the driver is loaded*, and it applies to *all* kernel modules. In other words, unlike the check for the WHQL signature or known publisher described previously, this check occurs regardless of whether the driver being loaded is plug-and-play. During driver installation, and each time thereafter when the system is booted, the check is performed. When you unplug/replug, the driver is checked

(Continued on page 13)


Windows Security

Would you like to install this device software?

Name: Videology Imaging devices
Publisher: i-NET STREAM

Always trust software from "i-NET STREAM".

You should only install driver software from publishers you trust. How can I decide which device software is safe to install?

Figure 1—Trusted Publisher Warning

Check This Out...

(Continued from page 12)

again. Even if your driver package is correctly installed, the driver will not be loaded unless it passes the KMCS check. Attaching a kernel debugger disables the KMCS check.

The KMCS signature can be present either in your SYS file, or in your CAT file. Non-plug-and-play drivers usually won't have a CAT file, so the SYS file must be signed. If you sign the CAT file only, you cannot replace the SYS file (for example, during debugging). The CAT file contains a checksum of the matching SYS file. For this reason, I usually sign both the CAT and the SYS. For drivers used during system boot, Microsoft strongly suggests that you sign the SYS file, because it can take too long to locate the corresponding CAT file with the limited disk driver stack used during boot.

Kernel-mode code signing check requires a very specific type of signature, and that the driver be signed in a very particular way. Specifically:

- You must use a valid Class 3 Code Signing Certificate – an ordinary certificate, such as used for SSL, is not acceptable
- The Class 3 Code Signing Certificate must be issued by a Microsoft-approved Certification Authority (see [1] at the conclusion of the article for a list)
- At the time of signing, you must reference the "cross-certificate" for the specific Certification Authority that issued the Class 3 Code Signing Certificate you're using.

When you sign your SYS and/or CAT file, you must select both your code-signing certificate, and the associated cross certificate (using the /ac parameter of signtool). This produces a certificate "chain" that ends at the "Microsoft Code Verification Root". That certificate chain must be present either in the SYS file or the CAT file or both, but wherever it exists, it must be a complete chain with cross-certificate.

My "signtool" command line looks like this:

```
signtool sign
   /v
   /ac MSVC-GlobalSign.cer
   /sha1 xxxxxxxxxx
   /t http://timestamp.verisign.com/scripts/timestamp.dll
   objfre_wlh_amd64\amd64\mydriver.sys
```

Where:

/v – verbose output
/ac – to select the cross certificate
/sha1 – to indentify your code signing certificate's thumbprint
/t to embed a timestamp so the signature does not expire

Where to Go from Here?
Hopefully, this short article has help clarify some of the more confusing aspects of driver signing.

Microsoft has published a thorough hands-on tutorial describing the KMCS process on the WHDC web site (see [2] further below). By carefully following the steps outlined in this document, you're *guaranteed* to be able to successfully sign your driver. Of course, this assumes you have the correct certificate (as described above)!

[1] http://www.microsoft.com/whdc/driver/install/drvsign/crosscert.mspx
[2] http://www.microsoft.com/whdc/driver/install/drvsign/kmcs-walkthrough.mspx

Tim Roberts (timr@probo.com) is a developer with Providenza & Boekelheide, Inc., a consulting firm in beautiful Tigard, Oregon. He's been programming for 35 years, and living in the Windows world since version 3.0. Tim spends way too much time with computers.

EResources Close Cousin

Implementing Reader/Writer Locks in User Mode

Being primarily kernel mode programmers at OSR, we have gotten into the habit of using Reader/Writer locks. These locks, known as Executive Resources (**EResources**), are great because they can be used at IRQLs less than DISPATCH_LEVEL and allow shared readers and exclusive writers. In addition they also allow a thread recursive access, as long as the acquisition type is compatible with the lock's current acquisition level.

That being said, it has always baffled us, at OSR, why something similar to **EResources** was never available in user mode. We recently had a need to move a piece of software that previously ran in kernel mode to user mode and that software happened to use **EResources** as its primary locking model. Our choices were either to reengineer the locking model used by the code or to implement **EResource** semantics in user mode. Our choice was to implement **EResource** semantics in user mode and this article documents our implementation.

Overview

We're going to assume that readers of this article are familiar with **EResources**, but realize that some people are not, so for the unfamiliar here's an overview…..

As we mentioned previously an **EResource** is a reader/writer lock. This lock can be acquired in 1 of 2 modes, either shared or exclusive As the modes imply this lock can be held simultaneously by multiple readers (**EResource** acquired in shared mode) or by one writer (**EResource** acquired in exclusive mode).

When a thread attempts to acquire an **EResource**, it can specify whether or not it wants to wait if the specified **EResource** is not immediately available. If it elects to wait, the thread will be put into a wait state until such time as the lock becomes available, otherwise the acquisition of the **EResource** will fail. An **EResource** can be acquired by a thread recursively as long as the acquisition type (i.e. shared or exclusive) is compatible with the current acquisition type of the lock. What this means is that if a thread holds a lock exclusive and tries to require the lock shared, the acquisition will be granted. If, however, the thread holds the lock shared and tries to reacquire the lock exclusive, a deadlock will occur (as it does in kernel mode) since the desired acquisition type is incompatible with the currently held acquisition type. Threads other than the current thread can attempt to acquire the **EResource** also and will only be allowed access if the mode of the request is compatible with the current state of the lock. So, for example, if Thread A owns the **EResource** shared, Thread B will also be allowed access to the **EResource** if it attempts to acquire it shared. If it tries to acquire the **EResource** exclusively, access will be denied (i.e. it will either be blocked if the user specified to wait for the **EResource**, or a status will be returned indicating that the **EResource** was not acquired).

Other features of **EResources** are:

- If an **EResource** is held exclusively, it can be converted to shared access
- Users can check to see if an **EResource** is currently held exclusively or shared
- Users can check to retrieve the number of Shared Waiters or Exclusive Waiters for an **EResource**.
- **EResources** must be initialized before being used and must be deleted when they are finished being used.

Now that the basics have been explained, let's see the functions available to us.

(Continued on page 15)

EResources...

(Continued from page 14)

Functions

In order to use **OSRERESOURCE**s (our version of **ERESOURCE**s) there are a set of functions that the user has available to them. The functions are listed in *Figure 1*.

To use an **OSRERESOURCE** the caller would call **OSRInitializeResource** in some initialization routine, and then call one of the **OSRAcquireResourceXXXLite** functions where XXX is either shared or exclusive. After doing some work it would then call **OsrReleaseResource** to release the held resource. Finally, when done with the resource, the user would call **OSRDeleteResourceLite** to delete the resource.

So as you can see **OSRERESOURCE**s are pretty easy to use. Since this is a user mode library free for anyone to use, I guess it would be a good thing to discuss what is going on underneath the covers of the **OSRERESOURCE** implementation, so that is what we'll discuss next.

Underneath the Covers of OSRERESOURCEs

In this section we will discuss how we implemented OSRERESOURCEs. Our implementation creates a Dynamic Link Library called "**UMERESOURCE.DLL**" This DLL has a set of "C" interfaces that we listed in Table 1 that are defined in the include file "**UMERESOURCE.H**". These interfaces work on an **OSRERESOURCE** structure and are backed by a "C++" **OSREResource** class which is defined in "**ERESOURCE.H**". The **OSREResource** class and associated **OSRERESOURCE_STATE** structure are defined in *Figure 2* (page 26).

This class contains:

- **m_CriticalSection** which is used by the code to synchronize access to the **OSREResource** data structures, ensuring that only one thread is accessing them at one time.
- **m_LockStateMap** table which maps a Thread identifier to its resource state (i.e. the mode in which it acquired the lock). This resource state is an **OSRERESOURCE_STATE** structure, defined in Table 2, which is composed of 2 fields, **State** and **AcquireCount**. **State** is used to keep track of how the resource was acquired, **ACQUIRED_NONE**, **ACQUIRED_SHARED**, or **ACQUIRED_ EXCLUSIVE,** and **AcquireCount** is used to keep track of the number of times the thread has acquired the resource (remember that resources can be acquired recursively).
- **m_WaitEventHandle** – the handle to the notification event that all threads wait on when trying to acquire the resource.
- **m_SharedCount** – number of shared holders of the

(Continued on page 26)

Function Name:	Purpose:
OSRInitializeResource	Initializes a resource for use. If not done, the resource is unusable
OSRDeleteResourceLite	Deletes an resource, the resource must not be held by any thread when deleted.
OSRAcquireResourceExclusiveLite	Attempts to acquire the resource exclusively
OSRAcquireResourceSharedLite	Attempts to acquire the resource shared
OSRAcquireSharedStarveExclusive	Attempts to acquire the resource shared and starves any thread waiting to get the resource exclusively
OSRIsResourceAcquiredExclusiveLite	Returns TRUE if the resource is held exclusively
OSRIsResourceAcquiredSharedLite	Returns TRUE if the resource is held shared or exclusive
OSRReleaseResourceLite	Releases the held resource
OSRGetCurrentResourceThread	Returns the current thread identifier
OSRConvertExclusiveToSharedLite	Converts a currently held exclusive reservation into a shared reservation
OSRGetSharedWaiterCount	Returns the number of threads waiting for shared access
OSRGetExclusiveWaiterCount	Returns the number of threads waiting for exclusive access
OSRReleaseResourceForThreadLite	Release the resource on behalf of the input thread identifier
OSRReleaseResource	Release the resource held by the current thread

Figure 1—OSRERESOURCE Functions

WDK Community Bug Bash Contest 2010

Help Eradicate Bugs!

We can't be more thrilled to be organizing another event to help bring the Windows driver development community together for a good cause—in this case, to help eradicate bugs in the Windows Driver Kit (WDK). Thus, on behalf of sponsors **OSR**, and **ITT Defense & Information Solutions**, and in cooperation with *Microsoft*, we're pleased to announce the WDK Community Bug Bash Contest!

Purpose: help fix bugs in the Windows Driver Kit (WDK)

Bugs? Yes, bugs. Kit bugs, doc bugs, sample bugs, utility bugs—if it's provided as part of the WDK we (and our friends on the WDK team at Microsoft) want to know about it. This is your chance. You have to work with this toolset for your job. Why not take a bit of effort to report bugs so that they can get fixed for you and future Windows kernel devs. Earning goodwill not enough of an incentive? How about…

Prizes?Yes, prizes.

Specifically, every valid submission of a bug against the WDK earns an award. *Every valid submission*. In addition, all valid submissions are then eligible to win one of two First Prizes, or one of three Second Prizes, as judged by the organizers.

Soooo, how does it all work?

1. Find bugs.
2. Report the bug(s). You've got six months, but remember, only the first submission of the same bug will be valid! While you're only eligible for one award, keep submitting to better your chances to win a First or Second place prize!
3. We'll validate submissions as they come in. If rejected, we'll let you know. If valid, we'll be contacting you to arrange for your award to be shipped.
4. We'll submit your bug directly into Microsoft's bug-tracking system. Feel free to check the Bug Bash Bug List from time to time for updates.
5. At the close of the contest, we'll get together and vote on the submissions—and award the First and Second Prizes.

Where do I start?

How about at the Bug Bash Homepage (http://www.osronline.com/page.cfm?name=bugbash). From there, you'll want to make sure to check out the contest rules, FAQ, Bug List and of course the Bug Submission pages.

Happy Hunting!

WDK Community Bug Bash Contest 2010

Get Your Stuff!

Everyone who submits a valid bug, gets cool stuff. Really!

If you're the first to report a bug in a sample driver that's been annoying you for ages, you'll get yourself a Technical Award package with a 4GB USB Flash Drive, a Windows 7 Keychain/Bottle Opener, and a cool WDK Community Bug Bash T-Shirt.

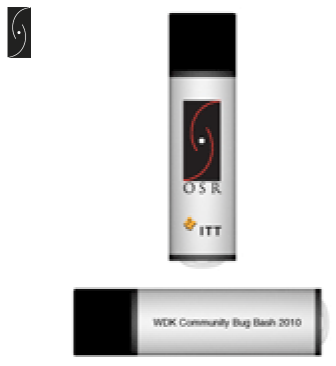
Or, say there's some hideous grammatical error in the WDK Docs that you noticed a while back. Report it, and get an Editorial Award package with a 1GB USB Flash Drive, a WDK Community Bug Bash Mouse Pad and one of those cool Windows 7 Keychain/Bottle Openers.

Report both a technical bug AND an editorial bug, and earn yourself BOTH awards!

Even BETTER: If your bug is chosen to be among THE BEST that were submitted, you can win yourself some VERY cool prizes: an HP Mini-210 Netbook, a copy of Visual Studio Ultimate with MSDN Subscription, or an Apple iPod Touch.

The idea behind the WDK Community Bug Bash is that **everybody** benefits. The community gets involved, bugs get fixed, and we all have a good time.

See the Bug Bash Homepage at http://www.osronline.com/page.cfm?name=bugbash for details, official contest rules and to submit your bugs. Now… get to work submitting those bugs!



The WDK Community Bug Bash Contest is sponsored by:

OSR open systems resources inc. ITT

In cooperation with:

Microsoft

Writing Filters...

(Continued from page 1)

The Players: DFS Client, MUP, and Network Redirectors

In order to understand the complexities of the problems we've experienced, we'll need to first get a feel for the way things work without any filters involved. During the boot process, special drivers called network redirectors (typically either monolithic network file system or mini-redirector drivers) register with the Multiple UNC Provider (MUP) driver by calling **FsRtlRegisterUncProvider** (or **FsRtlRegisterUncProviderEx** in Vista and later), providing the name of their redirector device object as part of that registration process. For example, in the case of the SMB mini-redirector driver, the name provided to MUP is \Device\LanmanRedirector. Other popular redirector device object names are \Device\WebDavRedirector and \Device\NetWareRedirector, for WebDAV and Novell Netware support, respectively.

Upon receipt of this request, MUP uses the device object to extract and store the name of the redirector device as well as a pointer to the redirector device object. Later, when a user attempts to open a name in the form of \\FooServer\Share, MUP begins calling each of the redirector drivers to determine if this path represents a server that the redirector supports. If the redirector indicates that the server is indeed one of its own, then the name resolution stops and MUP reparses the open (using the standard **STATUS_REPARSE** method for Windows) to the appropriate redirector device object. This restarts the create operation targeted at the appropriate redirector device and MUP, with its job done at this point, is no longer in the picture. A simplified depiction of this configuration is shown in *Figure 1* (note that the calling of the redirectors for the name resolution step is not depicted).

The introduction of DFS in this picture complicates things, as shown in *Figure 2* (next page). With DFS, things begin identically to the non-DFS case behavior, with the open of \\DfsServer\DfsRoot first arriving at MUP's dispatch entry point for create operations, **MupCreate**. MUP then does a probe of the server to determine if it is a DFS server (details of this probe are outside of the scope of this article), and receives a response that indicates that \\DfsServer does in fact support DFS. At this point, MUP then reparses this open of the DFS root back to its own driver, by replacing the name in the file object with \Device\WinDfs\Root\DfsServer\DfsShare and returning **STATUS_REPARSE**. \Device\WinDfs\Root happens to be another device object that the MUP driver creates to handle opens targeted to DFS servers.

As a result of this reparse, processing of the open restarts back at the **MupCreate** function. However, this time the create is targeted to the DFS device object and proceeds down a different code path in MUP that handles DFS related activity. At the start of processing, MUP finds the appropriate redirector to use to communicate with this DFS server and prepares to send this create request to the DFS server via the redirector. Due to the fact that DFS runs over SMB, normally the redirector device chosen will be \Device\LanmanRedirector. If this operation completes successfully, MUP completes the create IRP back to the original caller.

Note that the end result of this is different than in the non-DFS case. In the non-DFS case, the create operation was reparsed to the appropriate redirector. In the DFS case, the create is reparsed back to MUP, passed to the redirector, and then completed. Due to these differences, in the non-DFS case the resulting file object points to a redirector device object and in the DFS case the resulting file object points to a MUP device object.

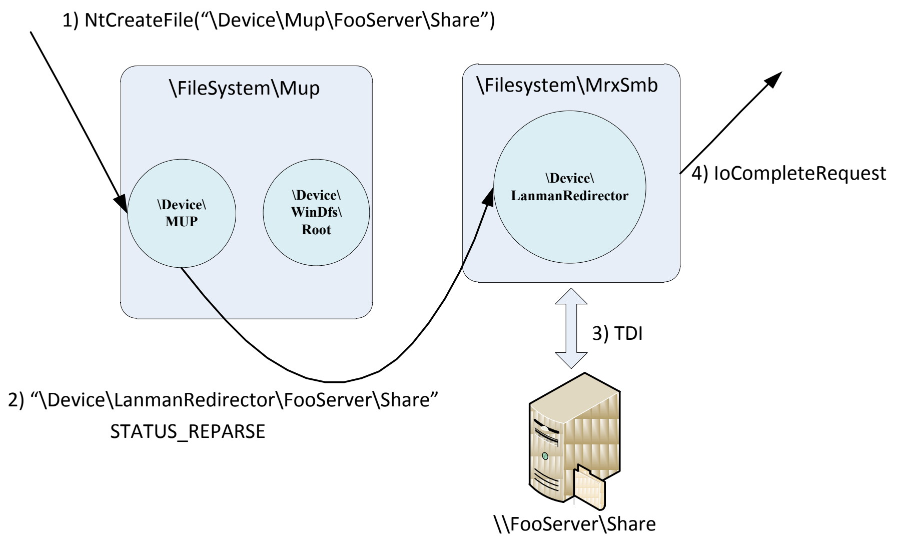


Figure 1—The Lay of the Land, Sans DFS

This Is Where It Starts To Get Messy...

If you had a hard time following the above, it's going to get a bit worse. As it turns out, LanmanRedirector must have knowledge of the fact that the create operation is targeted to a DFS server. Unfortunately, the IRP_MJ_CREATE IRP structure is already entirely packed and there's no room for custom parameters. To accommodate the additional information the DFS developers had to find a creative way to pass the extra information.

To achieve this the DFS and redirector designers decided to use some fields of the file object passed along with the create operation. Under normal circumstances,

(Continued on page 19)

Writing Filters...

(Continued from page 18)

the FsContext and FsContext2 fields of the file object are set to NULL when the I/O Manager sends the create operation to the file system. Before completion of the create IRP, it is the job of the file system to set the FsContext field to a structure that is unique to the stream and the FsContext2 field to a structure that is unique to this open instance of the stream.

Based on the knowledge that nothing is usually put there before calling the file system, MUP stores a magic value of 0xFF444653 (0xFF"MUP") into FsContext2 and a pointer to a data structure in FsContext before passing the request to the redirector. Attempting to find more details about the magic value brings up practically nothing, though you can find the following definitions in the LanmanRedirector sources that used to be provided with the WDK:

```
#define DFS_OPEN_CONTEXT
0xFF444653

typedef struct _DFS_NAME_CONTEXT_ {
    UNICODE_STRING  UNCFileName;
    LONG            NameContextType;
    ULONG           Flags;
} DFS_NAME_CONTEXT, *PDFS_NAME_CONTEXT;
```

And some further details on their usage throughout the source of the sample.

As we'll soon learn, this approach creates issues for file system filters that need to filter the SMB redirector. However, before we point out the complexities this creates for filters, let's continue our investigation of how this all works without filters involved.

What About Accessing a Link?

So far, we've only seen accessing a standalone UNC share and the root of a DFS server. Accessing a link of a DFS server complicates this situation even further and brings to light another feature of the MUP DFS implementation.

When a user attempts to open \\DfsServer\DfsRoot\DfsLink, everything proceeds as it did previously. The create first arrives at MUP and MUP determines that this is a DFS server, so MUP reparses the create back to itself at its DFS root device object. DFS then finds the appropriate redirector device for \\DfsServer\DfsRoot, which will again be \Device\LanmanRedirector.

MUP then attempts to open \\DfsServer\DfsRoot\DfsLink on the DFS server via the redirector. Because this is not an actual share on the server but a link to another server, the DFS server will return a special status of **STATUS_PATH_NOT_COVERED** to the client. MUP responds to this special status by sending what is called a, "DFS referral packet" via the redirector to the server for the failing path. The response to this referral packet is the actual location of the DFS link, for example \\FooServer\Share.

MUP is now at the end of its name resolution process and it knows the real path that is to be used for this create operation. However, MUP is also now back to square one in processing the create request. In other words, MUP must process the open of \\FooServer\Share as if this is what the user initially opened. This is due to the fact that MUP is not sure which redirector to use to communicate with this server/share combination.

Therefore, at this point MUP sends the create IRP back to itself at **MupCreate**. Processing at this point proceeds exactly as it did in the non-DFS case, with MUP finding the appropriate redirector for this open, pre-pending the redirector device name to the path, and returning **STATUS_REPARSE**. For example, in this case the resulting name might end up as, \Device\LanmanRedirector\FooServer\Share.

(Continued on page 20)

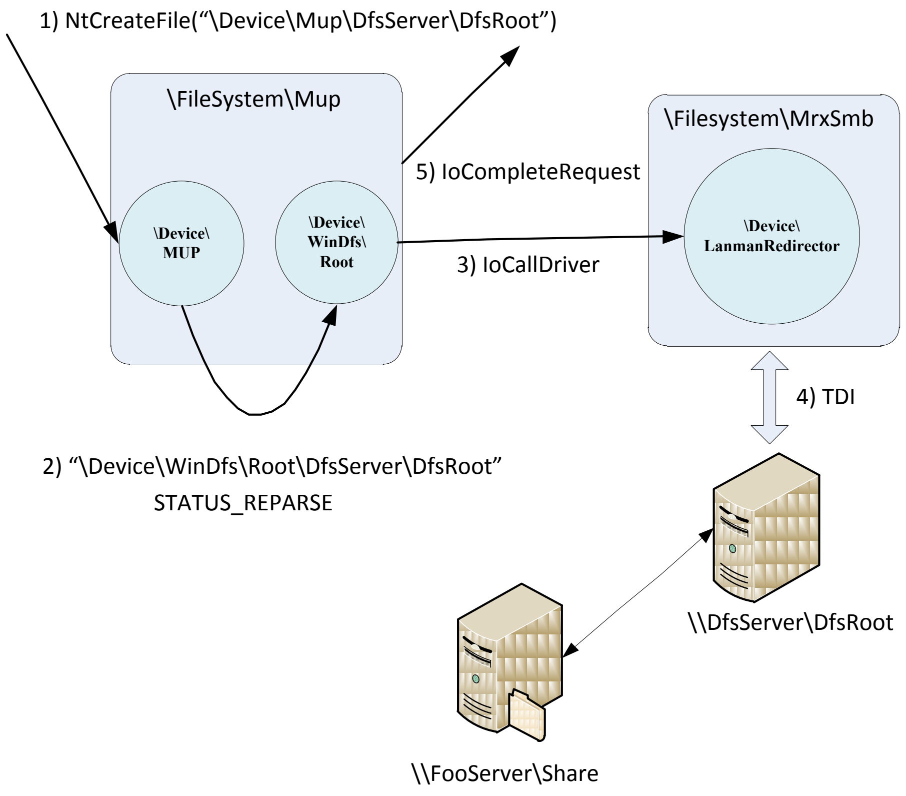


Figure 2—And Now, with DFS

Writing Filters...

(Continued from page 19)

You would believe that at this point MUP would return **STATUS_REPARSE** back to the I/O Manager and the resulting file object would point to \Device\Lanman Redirector, just as in the non-DFS case that we first looked at. However, that is not at all what happens. Instead, MUP internally handles the reparse processing and forwards the request to the appropriate redirector device object. Upon successful completion, MUP will then (finally) complete the user request with success, resulting in the file object pointing to the MUP DFS device object.

Why This Creates Additional Complexity for Filters

Why does filtering DFS turn out to be such a big pain for filter writers?

The first issue that you'll have when dealing with DFS is load ordering. When the redirector drivers register with MUP via FsRtlRegisterUncProvider, MUP opens the target device name and caches the returned device object. Thus, if your filter is not instantiated on the redirector device at the time of this open your filter will be bypassed for the communication between DFS and the redirector. Mini-filters luck out a bit here because you just need to make sure that Filter Manager is loaded and attached early, this ensures that the Filter Manager device object will be in the call chain and then you can just attach your filter instance later. Usual load ordering rules can be used to make sure that fltmgr.sys is loaded early enough in the boot process. Prior to Vista, in order to guarantee that Filter Manager will actually *attach* early in the boot process make sure that the Filter Manager **AttachWhenLoaded** registry value is correctly set to 1.

With your filter properly inserted between MUP and the redirector you have a new set of issues to handle. Remember that DFS and the redirector have a secret handshake stuffed in a couple of fields of the file object. In order for this design to work in all cases, those fields must be propagated to whatever file object is sent to the redirector. This is a monster problem for a filter that wants to perform its own open of the file or directory with **IoCreateFileSpecifyDeviceObjectHint** or **FltCreateFile**. These APIs generate their own file objects with no opportunity for the caller to modify the resulting file object before it is sent to the target device. Thus, a filter performing these types of operations breaks the chain between DFS and the redirector, leading to unexpected results from the create.

This is also a case where writing your filter as a mini-filter will create surprising complications. For reasons unknown to us, before calling the mini-filters in the create path, Filter Manager will set the FsContext field of the file object to NULL, even if it was not NULL on entry to the Filter Manager filter device. Once all of the mini-filters are called, the value is restored before calling the underlying FSD with the request. Thus, even if you could figure out a way to get the fields set appropriately in the file object, the values are hidden from your filter during pre-create processing.

And Don't Ever Try to Return STATUS_REPARSE...

If you noticed above, MUP heavily relies on **STATUS_REPARSE** in order to perform its work. We snuck something by you in a previous section as well when we said:

> *You would believe that at this point MUP would return **STATUS_REPARSE** back to the I/O Manager and the resulting file object would point to \Device\Lanman Redirector, just as in the non-DFS case. However, that's not at all what happens. Instead, MUP internally handles the reparse processing and forwards the request to the appropriate redirector device object.*

What we omitted in this explanation *how* MUP decides to handle this particular instance of **STATUS_REPARSE** itself.

(Continued on page 21)

Writing Filters...

(Continued from page 20)

Believe it or not, what MUP actually does is dig into the device object to which that the create IRP was sent, finds the driver object, and captures the driver object name. This driver object name is then compared to the hardcoded value, "\Filesystem\Mup". If the names match, MUP knows that he just reparsed the create back to himself and can handle it internally. However, if the names do not match then MUP allows the reparse to travel back to the I/O Manager.

This means that if you reparse a DFS open to something like a shadow stack (a technique we have used in some of our layered file system work), DFS will entirely step out of the way of the create processing. The result of this is that DFS create operations no longer pass through the DFS client code, which can again lead to unexpected results.

Note that this also prevents one from attaching a filter to \Device\Mup, which you might want to do in an attempt to avoid sitting between the DFS code and the redirector. Because this specially-handled create IRP will be sent to your filter device object instead of directly to the MUP device object, the driver name check will fail due to the fact that the driver name in the target device will be your filter driver's name. The end result will be a broken DFS client that can no longer communicate with any DFS servers.

Not All Stories Have a Happy Ending

Unfortunately, we continue fighting with DFS to this day and have open bugs for configurations that do not work and, quite possibly, might never be made to work. Complicating things for us is that MUP and DFS have been redesigned on Windows Vista and later such that filters are no longer sit in between MUP and the redirector. Because the design has moved forward, we have no chance of getting any changes made to legacy platforms. Until every last one of our clients has upgraded to Vista and later, we're going to be stuck trying to hammer filter drivers into an architecture that clearly wasn't designed to interact with other components in the system.

Getting Better...

(Continued from page 7)

If there are only four bytes actually valid, then the "Mode Data Length" should be set to 3 (i.e. 3 bytes are valid following the Mode Data Length field) since the Mode Data Length field is not included in the count.

Prevent/Allow Medium Handling

When the Storport Virtual Miniport received a SCSI operation that it did not handle, the driver set the SRB status to SRB_STATUS_ERROR and completed the request. While this isn't necessarily wrong, the Storport Virtual Miniport Driver is trying to emulate a device capable of handling CDBs. Therefore, the Miniport should really be responding as a real conformant device would. What this means is that, if an SRB **SenseInfoBuffer** has been specified, the driver should be filling it correctly.

Figure 5 (page 23), shows the routine **ProcessScsiCommandError** that sets the sense data that is returned when a command error occurs. This data will be returned if the input SRB contains a **SenseInfoBuffer**. Of course, your implementation of this may vary depending on what your driver needs to accomplish.

Notice that the driver "OR'd" SRB_STATUS_AUTOSENSE_VALID to SRB_STATUS_ERROR to indicate to the caller that the **SenseInfoBuffer** contains valid data.

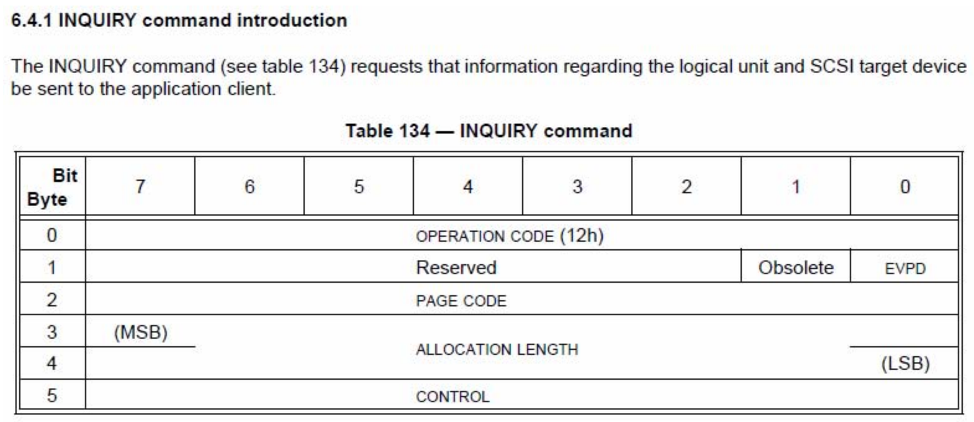
When you look at the new code, you will see that the driver has been modified to call the routine in Figure 5 whenever it receives a command that it does not handle.

Adapter Type

One interesting thing that Mike pointed out was that the **AdapterInterfaceType** for the Virtual Storport Driver was showing up as "Fibre". This **AdapterInterfaceType** is a field in the **PORT_CONFIGURATION_INFORMATION** block that is set up in our HwFindAdapter routine. Figuring that the field was not initialized the driver was modified to set the field to PNPBus. To our surprise setting this field had no effect. Since we were stumped as to why, we contacted James Antognini of Microsoft WDK support who informed us that Storport gets the setting from the field "\Registry\Machine\System\CurrentControlSet\Services\<driver name>\Parmeters\BusType". This field is a 32-bit REG_DWORD. Thus we modified the INF file to set the value of this key to 0xE (BusTypeVirtual) and it worked. If you don't set a value, it defaults to BusTypeFibre.

Crash

Alas, during his testing Mike managed to get the driver to crash. This problem has been fixed in the new code drop. It turns out that the code in **CreateConnection** that added a new connection to the list of current connections did not synchronize access to the **ConnectionList**. Thus if you added and removed connections



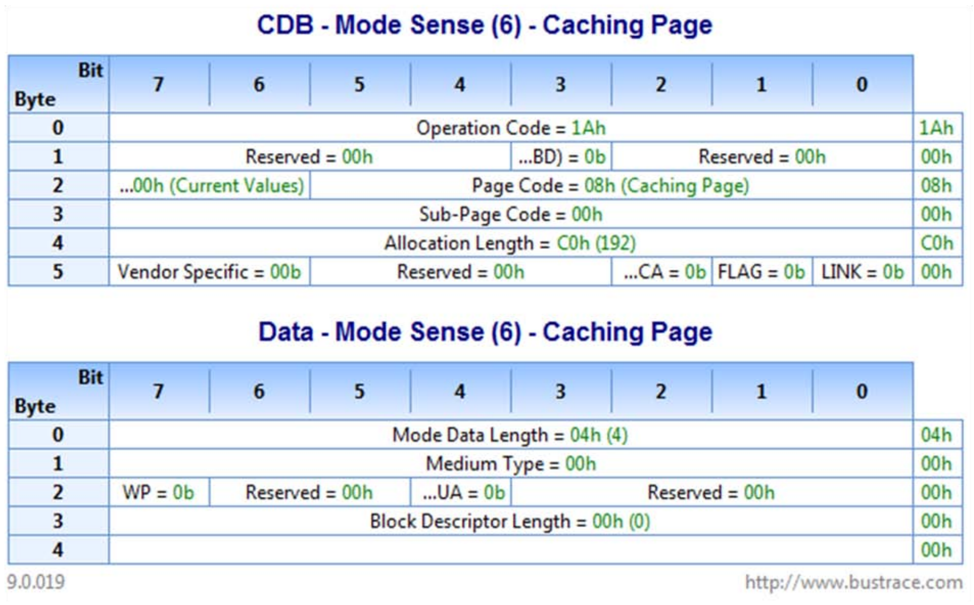
6.4.1 INQUIRY command introduction

The INQUIRY command (see table 134) requests that information regarding the logical unit and SCSI target device be sent to the application client.

Table 134 — INQUIRY command

Bit / Byte	7	6	5	4	3	2	1	0
0	OPERATION CODE (12h)							
1	Reserved						Obsolete	EVPD
2	PAGE CODE							
3	(MSB)	ALLOCATION LENGTH						
4								(LSB)
5	CONTROL							

Figure 3—SCSIOP_INQUIRY CDB



CDB - Mode Sense (6) - Caching Page

Bit / Byte	7	6	5	4	3	2	1	0	
0	Operation Code = 1Ah								1Ah
1	Reserved = 00h				...BD) = 0b	Reserved = 00h			00h
2	...00h (Current Values)		Page Code = 08h (Caching Page)						08h
3	Sub-Page Code = 00h								00h
4	Allocation Length = C0h (192)								C0h
5	Vendor Specific = 00b		Reserved = 00h			...CA = 0b	FLAG = 0b	LINK = 0b	00h

Data - Mode Sense (6) - Caching Page

Bit / Byte	7	6	5	4	3	2	1	0	
0	Mode Data Length = 04h (4)								04h
1	Medium Type = 00h								00h
2	WP = 0b	Reserved = 00h		...UA = 0b	Reserved = 00h				00h
3	Block Descriptor Length = 00h (0)								00h
4									00h

9.0.019 http://www.bustrace.com

Figure 4—busTRACE Mode Sense Output

(Continued on page 23)

Getting Better...

(Continued from page 22)

quickly enough you would cause the list to become corrupted. The fix was simple; just hold the **ConnectionListLock** around the adding of the new entry to the list in “**CreateConnection**”.

Summary

In our article series Writing a Virtual Storport Miniport Driver we described key aspects of the architecture, design, and implementation of this type of driver. We've built on that series, and made the driver more compliant, with the tweaks described in this article. And while these tweaks might not be absolutely critical, they do make the driver more compliant with the SCSI specification and thus make it act more properly like a physical device. This compliance helps increase the driver's chance of interoperating with all sorts of software, and that's a certainly good thing.

Sample code to OSR's Virtual Storport Miniport Driver can be downloaded from http://www.osronline.com/OsrDown.cfm/osrvmmemsample.zip?name=osrvmmemsample.zip&id=558

OSR would like to thank Mike Berhan of busTRACE Technologies for taking the time to test and analyze the OSR Virtual Storport Miniport Driver and for providing us with a copy of busTRACE so that we could find and fix the issues discussed in this article. Without the help of Mike and his fine tool, we wouldn't have known about the driver's failure to conform to the specifications.

```
NTSTATUS ProcessScsiCommandError(PSCSI_REQUEST_BLOCK PSrb)
{
    NTSTATUS status = STATUS_SUCCESS;

    if(PSrb->SenseInfoBuffer && PSrb->SenseInfoBufferLength) {
        PSENSE_DATA pSense = (PSENSE_DATA) PSrb->SenseInfoBuffer;
        RtlZeroMemory(pSense,PSrb->SenseInfoBufferLength);
        pSense->ErrorCode = 0x70;
        pSense->Valid = 0;
        pSense->SenseKey = SCSI_SENSE_ILLEGAL_REQUEST;
        pSense->AdditionalSenseLength = 0x15;
        pSense->AdditionalSenseCode = SCSI_ADSENSE_ILLEGAL_COMMAND;
        pSense->AdditionalSenseCodeQualifier = 0;
        PSrb->ScsiStatus = SCSISTAT_CHECK_CONDITION;
        PSrb->SrbStatus = SRB_STATUS_AUTOSENSE_VALID | SRB_STATUS_ERROR;
    } else {
        status = STATUS_UNSUCCESSFUL;
        PSrb->SrbStatus = SRB_STATUS_ERROR;
    }

    return status;
}
```

Figure 5—ProcessScsiCommandError

Debugger Extensions...

(Continued from page 9)

```
    //
    hr = debugControl->Evaluate("2 + 2",
                                DEBUG_VALUE_INT32,
                                &result,
                                NULL);

    if (hr != S_OK) {
        debugControl->Release();
        return hr;
    }

    debugControl->Output(DEBUG_OUTCTL_ALL_CLIENTS,
                         "Result is %d\n",
result.I32);


    //
    // Done with this.
    //
    debugControl->Release();

    return S_OK;
}
```

Hopefully the steps followed are fairly straightforward at this point. We've taken the passed-in client object, created an instance of IDebugControl, and then executed some methods on it to perform actions.

To further drive home the pattern, we can see how we'd get the offset of a field of a data structure. For that, we need an instance of IDebugSymbols:

```
HRESULT CALLBACK
myothercommand(PDEBUG_CLIENT4 Client, PCSTR args)
{
    PDEBUG_SYMBOLS debugSymbols;
    HRESULT        hr;
    ULONG          fieldOffset;
    ULONG          typeId;
    ULONG64        module;

    UNREFERENCED_PARAMETER(args);


    //
    // Let's find the offset of the CurrentThread
    // field of the PRCB
    //
    hr = Client->QueryInterface
                        (__uuidof(IDebugSymbols),
                         (void **)&debugSymbols);

    if (hr != S_OK) {
        return hr;
    }

    //
    // We need the "type identifier" and module
    // containing the symbol that we're interested
    // in.
    //
    hr = debugSymbols->GetSymbolTypeId("nt!_KPRCB",
                                       &typeId,
                                       &module);

    if (hr != S_OK) {
        debugSymbols->Release();
        return hr;
    }

    //
    // Now we can get the offset.
    //
    hr = debugSymbols->GetFieldOffset
                   (module,
                    typeId,
                    "CurrentThread",
                    &fieldOffset);

    if (hr != S_OK) {
        debugSymbols->Release();
        return hr;
    }

    debugControl->Output
                   (DEBUG_OUTCTL_ALL_CLIENTS,
                   "Offset of CurrentThread is %d\n",
                   fieldOffset);

    debugSymbols->Release();
    return S_OK;
}
```

Dealing with 32-bit vs. 64-bit

Note that when you're writing a debugger extension command, your code is always running on the *host* machine. Also note that the pointer size of the host machine does not necessarily match the pointer size of the target machine, due to the fact that 32-bit hosts can debug 64-bit targets and vice versa. In order to deal with this, debugger extensions treat *all* addresses from the target as 64-bit values. Thus you'll note that the DbgEng APIs express pointer addresses as ULONG64 values. Any 32-bit value used in your extension command must be sign extended out to a full 64-bit value.

Alternative Debugger Extension Interfaces

To add to the confusion when it comes to writing a debugger extension, there are two alternative interfaces that you can use to write your debugger extensions. The first is the WdbgExts interface, which is the legacy debugger extension interface that existed before DbgEng came around. This interface is still available in the newest versions of the debugger, however it has two drawbacks. First, it is not the forward moving API thus it is frozen in time and will provide no new features. Second, this interface does not have any support for writing standalone applications, thus it won't port to any kind of automated analysis tool that you might write. If you're interested in learning more about the legacy extension model, see this article on OSR Online.

The other interface available to you is the EngExtCpp interface. This is actually just a C++ wrapper library around DbgEng to simplify common tasks such as manipulating typed data. Unlike WdbgExts, this is a fully supported interface and can be used alongside the direct DbgEng calls that we've discussed in this article.

Go forth!

Hopefully we've been able to clear up the cloud of mystery that hangs over writing debugger extensions and have set you on a journey of creating your own.

Source to a debug extension (!uniqstack) can be downloaded from: http://www.osronline.com/OsrDown.cfm/apexts.zip?name=apexts.zip&id=559

Getting Away...

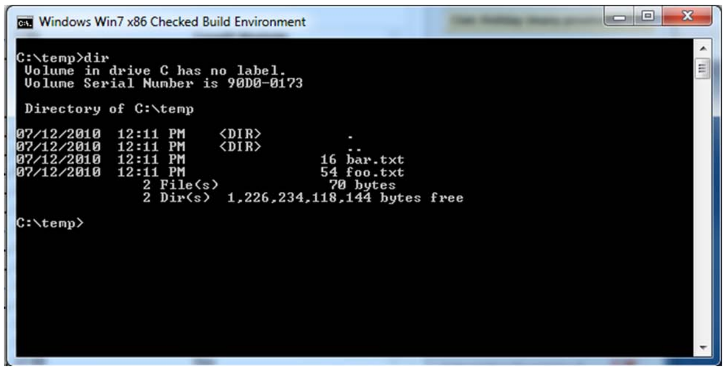
(Continued from page 11)

programs. In *Figure 2* we show what appears to be a pair of files – one of these is 16 bytes long and the other is 54 bytes long. In fact, there really is only one file as we have created a hard link. If we access the file via the hard link, we can see the size change (in *Figure 3*).

This is because NTFS actually stores the file sizes within the directory entry – but when a file contains hard links, only the link that is actually used is updated. Despite this, applications rely upon correct size information and in some cases they use it for validation.

You can address this by adding "directory size correction" however the overhead for directory size correction can be surprisingly harsh, particularly if it requires opening the file to determine what the correct size should be.

For example, we have found in our own work that adding *any* additional I/O overhead in directory enumeration can have dramatic performance impact. When this occurs in a high



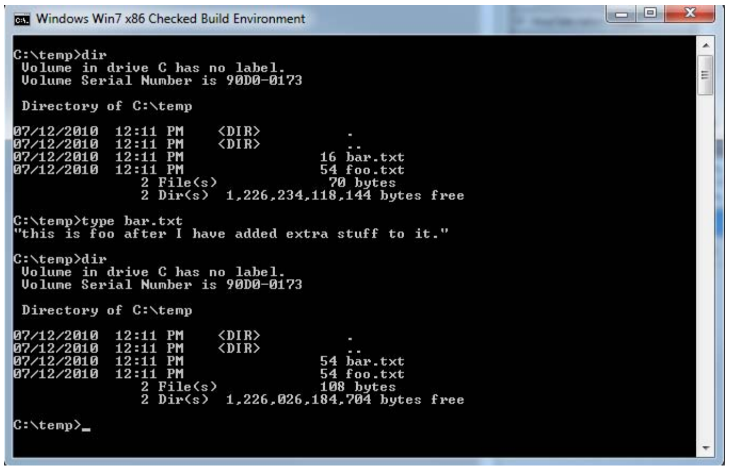
```
C:\temp>dir
 Volume in drive C has no label.
 Volume Serial Number is 90D0-0173

 Directory of C:\temp

07/12/2010  12:11 PM    <DIR>          .
07/12/2010  12:11 PM    <DIR>          ..
07/12/2010  12:11 PM                16 bar.txt
07/12/2010  12:11 PM                54 foo.txt
               2 File(s)             70 bytes
               2 Dir(s)  1,226,234,118,144 bytes free

C:\temp>
```

Figure 2



```
C:\temp>dir
 Volume in drive C has no label.
 Volume Serial Number is 90D0-0173

 Directory of C:\temp

07/12/2010  12:11 PM    <DIR>          .
07/12/2010  12:11 PM    <DIR>          ..
07/12/2010  12:11 PM                16 bar.txt
07/12/2010  12:11 PM                54 foo.txt
               2 File(s)             70 bytes
               2 Dir(s)  1,226,234,118,144 bytes free

C:\temp>type bar.txt
"this is foo after I have added extra stuff to it."

C:\temp>dir
 Volume in drive C has no label.
 Volume Serial Number is 90D0-0173

 Directory of C:\temp

07/12/2010  12:11 PM    <DIR>          .
07/12/2010  12:11 PM    <DIR>          ..
07/12/2010  12:11 PM                54 bar.txt
07/12/2010  12:11 PM                54 foo.txt
               2 File(s)            108 bytes
               2 Dir(s)  1,226,026,184,704 bytes free

C:\temp>_
```

Figure 3

access directory (e.g., "C:\Windows") the performance of the system can become so poor that your driver will be deemed "unusable" as a result. In such situations we've been forced to add caching to the size correction process. Even that will not make the first enumeration of a directory "fast" however. Directory size correction is an area we will leave for a future document and discussion, because it is itself surprisingly complicated.

Summary

In future installments as we build our sample, we will focus on the mechanisms necessary to manage our shadow file objects, contexts and section object pointer structures.
Stay tuned!

EResources...

(Continued from page 15)

resource. Since threads can hold the resource recursively this does not necessarily reflect the number of threads holding the resource simultaneously.

- **m_StarveExclusiveCount** – number of threads waiting to acquire the resource in shared mode, but want to starve exclusive waiters.
- **m_ExclusiveCount** – number of exclusive holders of the resource, which is usually one, but since a thread can acquire resource lock recursively, this count would reflect that.
- **m_SharedWaitersCount** – number of threads waiting to acquire the resource in shared mode.
- **m_ExclusiveWaitersCount** – number of threads waiting to acquire the resource in exclusive mode.
- **m_Initialized** – Boolean indicating whether or not the resource was initialized by a call to **OSRInitializeResource**.

The code which implements this class uses the **m_CriticalSection** to synchronize access to the rest of the **OSREResource** data, while the **m_LockStateMap** is used to keep track of each holder of the resource, the mode of acquisition (**State**), and the number of times that the acquisition was made since a holder could acquire the resource recursively (**AcquireCount**). The data members' **m_SharedCount** and **m_ExclusiveCount** are also used to keep track of the acquisition counts.

```
// This structure keeps track of the state of the resource.  We need
// to know how the resource was acquired and how many times it was
// recursively acquired.
//
typedef struct _OSRERESOURCE_STATE
{
    union {
        struct {
            ULONG   State;          // Acquisition State
            ULONG   AcquireCount;   // Count of times acquired
        };
        ULONGLONG   Reserved;
    };
} OSRERESOURCE_STATE, *POSRERESOURCE_STATE;


class OSREResource : public CObject
{
public:
        OSREResource();
        virtual ~OSREResource();

        enum {
                ACQUIRED_NONE,
                ACQUIRED_SHARED,
                ACQUIRED_EXCLUSIVE
        } ACQUIRED_STATE;

        BOOLEAN AcquireResourceExclusive(BOOLEAN Wait);
        BOOLEAN AcquireResourceShared(BOOLEAN Wait);
        BOOLEAN AcquireSharedStarveExclusive(BOOLEAN Wait);
        BOOLEAN IsResourceAcquiredExclusive();
        BOOLEAN IsResourceAcquiredShared();
        VOID ReleaseResource();
        VOID ConvertExclusiveToShared();
        VOID ReleaseResourceForThread(ERESOURCE_THREAD ResourceThreadId);
        ULONG   GetSharedWaiterCount();
        ULONG   GetExclusiveWaiterCount();
        NTSTATUS InitializeResource();
        NTSTATUS DeleteResource();

private:
        mutable CRITICAL_SECTION      m_CriticalSection;

        typedef std::map<DWORD,OSRERESOURCE_STATE> CMapThreadToLockState;
        CMapThreadToLockState m_LockStateMap;

        volatile DWORD m_SharedCount;
        volatile DWORD m_StarveExclusiveCount;
        volatile DWORD m_ExclusiveCount;
        volatile DWORD m_SharedWaitersCount;
        volatile DWORD m_ExclusiveWaitersCount;
        HANDLE  m_WaitEventHandle;
        BOOLEAN m_Initialized;
};
```

Figure 2—OSREResource Class and OSRERESOURCE_STATE Structure

For any thread which tries to acquire the **OSREResource** and elects to wait for the resource, the **m_WaitEventHandle** is the handle to the event that the thread will wait on. This event is called a "**NotificationEvent**". What this means is that unlike a "**SynchronizationEvent**" where only the first waiting thread is woken up, when the event is set, all waiting threads are woken up. This means that each awakened thread has the possibility of acquiring the resource. Consider the scenario where a thread holds the resource exclusively and 10 threads are waiting to acquire the resource shared. In our implementation all the shared waiters will be awakened when the exclusive holder releases the resources and will be able to acquire the resource.

Since the source code is readily available for download, we don't want to spend a lot of time going through it. What we do want to do however is go over **AcquireResourceShared** and **ReleaseResource** to at least give you a feel for how the code works.

AcquireResourceShared

Figure 3 (page 27) contains the **AcquireResourceShared** function. As the name implies, this function attempts to acquire

(Continued on page 27)

EResources...

(Continued from page 26)

the **OSREResource** shared and if it cannot, waits if the user indicated wait for the resource to become available. Remember that our code acquires **m_CriticalSection** to ensure proper synchronization when access the class data structures.

This routine first looks to see if the calling thread already holds the resource by looking for the threads' Id in the **m_LockStateMap**. If the thread is found, then the code updates the ownership counts and returns TRUE to the caller indicating that the resource has been acquired.

If the thread Id was not found in the **m_LockStateMap**, the code then checks to see if the resource is held exclusively or if there are threads waiting for the resource exclusively. If there

(Continued on page 28)

```
BOOLEAN OSREResource::AcquireResourceShared(BOOLEAN Wait)
{
        DWORD cThread = GetCurrentThreadId();
        ULONG loopCount = 0;
        EnterCriticalSection(&m_CriticalSection);

                    m_SharedWaitersCount++;

        while(TRUE) {
                // See if we already own the lock.
                //
                CMapThreadToLockState::iterator ite = m_LockStateMap.find(cThread);
                if(ite != m_LockStateMap.end()) {
                        //
                        // We already own it, let's see what access we already have.
                        //
                        ite->second.AcquireCount++;
                        if(ite->second.State & ACQUIRED_SHARED) {
                                // We already have it shared, so just bump the count.
                                //
                                m_SharedCount++;
                        } else if(ite->second.State & ACQUIRED_EXCLUSIVE) {
                                // We have it exclusive, so just give it to the caller exclusive
                                //
                                m_ExclusiveCount++;
                        }
                                                        m_SharedWaitersCount--;
                        LeaveCriticalSection(&m_CriticalSection);
                        return TRUE;
                } else if(m_ExclusiveCount || m_ExclusiveWaitersCount) {
                        // Someone else owns it exclusive or there are exclusive waiters, so we have to wait.
                        //
                        if(!Wait) {
                                                            m_SharedWaitersCount--;
                            LeaveCriticalSection(&m_CriticalSection);
                            break;
                        }
                        LeaveCriticalSection(&m_CriticalSection);
                        WaitForSingleObject(m_WaitEventHandle,INFINITE);
                        // Add a wait loop to prevent this thread from hogging the CPU.  The event gets set
                       // when the holder releases the lock and we have to ensure that everyone gets a chance
                        // to run before continuing.
                        //
                        loopCount++;
                        if(loopCount == 5) {
                                Sleep(200);
                                loopCount = 0;
                        }
                        EnterCriticalSection(&m_CriticalSection);
                        continue;
                } else {
                        // Nobody has it or wants it, so we will take it. We reset the event, so that it is no
                       // longer signaled. Anyone who was waiting for the event should already be woken up.
                        //
                        OSRERESOURCE_STATE state;
                        state.State = ACQUIRED_SHARED;
                        state.AcquireCount = 1;
                        m_SharedCount++;
                                                        m_SharedWaitersCount--;
                        m_LockStateMap.insert(std::make_pair(cThread,state));
                                                        ResetEvent(m_WaitEventHandle);
                        LeaveCriticalSection(&m_CriticalSection);
                        return TRUE;
                }
        }
        return FALSE;
}
```

Figure 3—AcquireResourceShared

EResources...

(Continued from page 27)

are, then this indicates that the caller cannot acquire the resource. What is done next depends on the input Wait parameter. If this parameter is **TRUE**, then the caller will wait on the **m_WaitEventHandle**. If this parameter is **FALSE**, the code will return to the caller indicating that the resource was not acquired.

Finally, if the resource is not held exclusively, or if there are no threads waiting for the resource exclusively, then the code inserts an entry into the **m_LockStateMap**. This entry indicates how the resource was acquired, updates the appropriate counts in the class, and returns to the caller indicating that the resource was acquired.

There is one thing that we must highlight here and it is that the loop code follows the **WaitForSingleObject**. Keep in mind that there could be many threads, with various scheduling priorities, wanting to acquire the **OSREResource** in different ways, all waiting for it to become free. If we automatically gave acquisition of the **OSREResource** to the first thread through the code, we could potentially starve other waiters, since we know that highest priority will almost always get the CPU first. We didn't want to keep track of the waiters in a list and heuristically pick the next owner, so we decided to make it somewhat random (BTW, if you don't like it, change it and let us know what your solution is and why it is better!). Therefore we have all threads coming out of the **WaitForSingleObject** sleeping for about 200 milliseconds, to let all threads that were blocked have time to run through the acquire code again and reassess their access to the lock.

That's all there is to acquiring the **OSREResource** shared, so let's see how you release it.

```
VOID OSREResource::ReleaseResource()
{
        DWORD cThread = GetCurrentThreadId();

        EnterCriticalSection(&m_CriticalSection);

        //
        // See if we own the lock.  We'd better.....
        //
        CMapThreadToLockState::iterator ite = m_LockStateMap.find(cThread);

        if(ite != m_LockStateMap.end()) {
                //
                // We own it.   Decrement the ownership count based upon
                // our access.
                //
                if(ite->second.State & ACQUIRED_SHARED) {
                        m_SharedCount--;
                } else {
                        m_ExclusiveCount--;
                }

                //
                // Look at our ownership count in the lowpart of the
                // large integer.   If it is 1, then this is our
                // last reference to the lock, so we will delete
                // ourselves from the ownership map.
                //
                if(ite->second.AcquireCount == 1) {
                        //
                        // erase us, we no longer own the lock.
                        //
                        m_LockStateMap.erase(ite);

                        //
                        // Wake up anyone waiting on access to the lock.
                        //
                        SetEvent(m_WaitEventHandle);
                } else {
                        //
                        // We still have outstanding references....
                        //
                        ite->second.AcquireCount--;
                }
                LeaveCriticalSection(&m_CriticalSection);
        } else {
                RaiseException(0xC0000002,EXCEPTION_NONCONTINUABLE,0,NULL);
        }
}
```

Figure 4—ReleaseResource

ReleaseResource

Figure 4 contains the code for **ReleaseResource**. As the name implies, this code releases an owned **OSREResource**. Remember that our code acquires **m_Critical Section** to ensure proper synchronization when accessing the class data structures.

The first step for the code is to attempt to find the calling thread in the **m_LockStateMap**, which keeps track of the threads that are currently holding the **OSREResource**. If the calling thread is not found in the **m_LockStateMap** structure, the code will raise a 0xC0000002 exception to indicate that the **OSREResource** was not held by the caller. If the caller was found in the **m_LockStateMap**, the code will decrement either the **m_Shared Count** or **m_ExclusiveCount** field, depending upon how the caller had acquired the **OSREResource.**

The code then looks at whether or not this is the calling thread's last reference to the **OSREResource**, which would indicate that the **OSREResource** is now free to be acquired by other threads. It checks for the last reference by looking at the **AcquiredCount** field of the returned **OSRERESOURCE _STATE** structure. If the

(Continued on page 29)

EResources...

(Continued from page 28)

AcquireCount is not 1, then the code knows that the **OSREResource** is still not available to other acquirers. If however the **AcquireCount** field is 1, then the code knows this is the last reference to the **OSREResource**, at which point it can delete the calling threads entry from the **m_LockStateMap**, and can signal the **OSREResource's** event (**m_WaitEventHandle**). Signaling the event will cause all the other threads waiting for this **OSREResource** to be awakened so that they can attempt to acquire the **OSREResource**.

Problems

As we mentioned earlier, the code will raise different exceptions if it determines that something bad is going on. These were added to aid in debugging problems that you may run into and are shown in *Figure 5*. As for deadlocks, the only way to debug those is to examine each thread in your application and see what it is waiting for in hopes that you can determine the deadlock.

Summary

So there you have it, an OSR implementation of Executive Resources in user mode. As you can see the code is pretty simple, and provides the advantages of **ERESOURCEs** to user mode, i.e. a true reader/writer lock.

You will find the code associated with this article at:
http://www.osronline.com/OsrDown.cfm/osreresource.zip?name=osreresource.zip&id=561

Exception Code:	Explanation:
0xC0000001	IsResourceAcquiredShared –invalid lock state, could indicate corruption
0xC0000002	ReleaseResource – resource not owned by caller
0xC0000003	ConvertExclusiveToShared – resource not owned exclusive
0xC0000004	[NOT USED]
0xC0000005	CovertExclusiveToShared – resource not owned by caller
0xC0000006	ReleaseResourceForThread – resource not owned by input thread
0xC0000007	DeleteResource – resource still in use
0xC0000008	AcquireResourceExclusive – already have the lock shared
0xC0000009	Resource not initialized
0xC000000A	Resource already initialized

Figure 5—Exceptions Thrown by UMERESOURCEDLL

Analyst's Perspective

Debug Smarter

OK, OK, I'll admit it…Debugging crash dumps can get tedious *fast*. In a recent system hang that I analyzed, there were 1,200 threads in the system. Given no details on what was going on at the time of the hang, my eyes would probably start bleeding before I found the threads that were interesting. And that wasn't even a big terminal server machine with lots of sessions running, so it was fairly tame by modern standards.

This is of course why we rely on our automated tools to do the heavy lifting for us. When the system crashes, we don't go look up the bugcheck code and start trying to decode trap frames or context records on the stack, we instead rely on *!analyze –v* to do this work for us. In my 1,200 thread case, I didn't bother taking a detailed look at every thread in the system but instead relied on *!stacks 2* to give me a summary of threads so that I could quickly scan for something that looked "interesting."

What I think we often lose sight of though is that these commands aren't magic. At some point in time, someone thought that their job would be made easier if there was a command that would quickly provide summary analysis information at the time of a crash. Someone else thought sifting through the output of *!process 0 7* was also far too painful and came up with a command to provide a summary view.

Why is it that we don't all think this way? Instead, we tend to rely on the existing commands and then complain that they don't work the way we'd like. Between WinDBG's scripting capabilities and its debugger extension support it's hard to say that there is anything it can't do. The common responses to that though are usually, "the scripting language is too cryptic" or, "I don't know how to write an extension." But, in reality, *any* language is cryptic until you take the time to learn it and most of us didn't know how to put pants on at some point in our lives. However, one day we decided that it was an important skill and decided to look at some examples and practice (if you still don't know how to put pants on, I apologize and you are deemed exempt from the remainder of this perspective).

The other trap that we all fall into is that we just don't allow ourselves the time to write a script or extension that will potentially save us hours down the road. When debugging a difficult problem, we tend to get tunnel vision and refuse to tear ourselves away from the problem to do something so frivolous. I definitely get dragged down into this one, and by the time I've figured the problem out the extension idea leaves my mind. Until I get the next dump of course, at which point I *really* wish I had written that extension…

So, I think it's time for a regime change. Let's promote debugging to first class citizenship and spend the time necessary to make our lives easier. Let's declare August 2010 the Month of Debugging Smarter and start rethinking how we approach solving our debugging problems. See P 8 of this issue for an article on the basics of writing a debugger extension and, to get your creative juices flowing, there's even the source of a kernel mode implementation of *!uniqstack* for you to play with. This command scans all of the threads in the system and, when finished, provides the *!thread* output of the threads in the system with unique call stack sequences. As an example of how many threads this eliminates, my 1,200 thread system ended up only having 105 unique call chain sequences, which is certainly much more manageable.

So no more excuses about not knowing how to write your own extension! And, if you do write your own extension, email me at ap@osr.com to let me know about it. Hopefully I can gather submissions from *The NT Insider* readership and put together some interesting tidbits for our next issue.

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

Peter Pontificates...

(Continued from page 5)

User stories are the software equivalent of task based documentation – another stoopid fad that I truly despise. But, you know, manager-troids love task-based documentation for the same reasons they love user stories for software development: It's easy to understand, it's easy to demonstrate that it's correct (just follow the steps and see if they work), and you can get an idiot to do it. The problem? It doesn't consider anything that's not specifically required for you to achieve the task. For example, you can write a document chapter entitled "How to Peel an Onion" that describes the process in detail. It will cover all the steps. But, on finishing the chapter, the reader hasn't learned anything about onions, peeling, or anything else. They've just learned how to peel an onion, and even then only under the conditions that the chapter contemplates, using the steps the chapter prescribes.

And so it is for user stories. Let's say you get a couple of user stories like "Bob is a church-going motorcycle rider that needs to store digital copies of his 'special' magazines in encrypted form on his flash drive" and "Hector served with Che Guevara in Bolivia and needs a safe place to store his secret communications." You code to these particular stories. No, you don't get a chance to think through the overall experience for *any* user. This is Agile software development. You don't get to think. You're not allowed to design. You're allowed to "get some code working" so you can try things out. And that code just needs to meet the user stories, and pass the tests that were so lovingly crafted and stored with those stories. Anything else? Well, that's for next sprint.

So, in Agile what you get are ridiculously incomplete requirements, driving a development process that emphasizes sloppy implementation over design, that's tracked using a long list of bogus and irrelevant milestones. The only thing that's left to wonder about is why *everybody* doesn't think Agile development sucks ass.

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

WINDOWS SYSTEM SOFTWARE

UNIQUE EXPERTISE, GUARANTEED RESULTS...THAT'S OSR

Training

OSR training services consist of public and private seminars on a variety of topics including Windows internals, driver development, file system development and debugging. Public seminar presentations are scheduled and presented in a variety of locations around the world, and customized, private presentations are delivered to corporate clients based on demand.

Consulting

In consultative engagements, OSR works with clients to determine needs and provide options to proceed with OSR, or suggest alternative solutions external to OSR. "Consulting" assistance from OSR can be had in many forms, but no matter how it is acquired, you can be assured that we'll be bringing our definitive expertise, industry experience, and solid reputation to bear on our engagement with you.

Custom Development

At OSR, we're experts in Windows system software: Windows device drivers, Windows file systems, and most things related to Windows internals. It's all we do. As a result, most OSR solutions can be proposed on a firm, fixed-price basis. Clients will know the cost of a project phase and deliverable dates *before* they have to make a commitment.

Toolkits

OSR software development toolkits provide solutions that package stable, time-testing technology, with support from an engineering staff that has helped dozens of customers deliver successful solutions to market.

More information on OSR products and services can be found at the www.osr.com.

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

RETURN SERVICE REQUESTED

PRESORTED
FIRST CLASS MAIL
U.S. POSTAGE
PAID
PERMIT # 456
MANCHESTER, NH

First Class

The NT Insider™ is a subscription-based publication

Subscribe to The NT Insider—Digital Edition

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

New OSR Seminar Schedule!

Seminar	Dates	Location
Writing WDM Drivers (Lab)	16-20 August	Seattle, WA
Writing WDF Drivers (Lab)	27-September-October 1	Santa Clara, CA
Kernel Debugging & Crash Analysis (Lab)	18-22 October	Portland, OR
Developing File Systems for Windows	26-29 October	Santa Clara, CA
Internals and Software Drivers (Lab)	15-19 November	Santa Clara, CA
Writing WDM Drivers (Lab)	6-10 December	Boston, MA

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