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.

May—June 2011 Digital Edition Volume 18 Issue 2

The Basics About WDF Queues

After having the opportunity to teach our course *Writing WDF Drivers for Windows* several zillion times now, we've learned quite a few things. One of the most important things we've learned is that the WDFQUEUE is one of the most underappreciated of the Framework objects. This is unfortunate, because WDF Queues are one of the most interesting and powerful objects that you'll use in your WDF driver. Plus, the WDF Queue is an object type that just about every WDF driver uses. It therefore seems reasonable that you should know a bit about these most excellent WDF Queue Objects.

Not Your Ordinary Queue

When we explain WDF Queues in our classes, the first problem that arises is that most devs have a pre-conceived notion of the meaning of the word "queue". When they hear this term, they immediately think of linked lists. If they're WDM driver writers, they think of LIST_ENTRY. While WDFQUEUEs *can* be used similarly to standard linked lists if you want, they are in fact *much* more special than those ordinary linked lists. In fact, WDF Queues are the primary mechanism that WDF drivers use to sort, manage, and control the delivery of Requests for processing.

Queues are of primary importance to WDF Drivers because they provide *the* most common method for delivering I/O requests from the Framework to the driver. As the Framework receives I/O requests from the Windows I/O Manager, it will insert the WDF Request object that represents that request on one of the driver's Incoming Request Queues. How those Requests are ultimately delivered to the driver depends on the Queues Dispatch Type.

Queue Dispatch Types

When you configure and subsequently create a WDF Queue, you specify the Queue's Dispatch Type. Dispatch Type controls how many Requests from the

(Continued on page 16)

Win8: The Speculation Continues

There's plenty of fodder in the press on Windows 8 these days, and whether or not you believe or even follow any of it, it's clear that many are in a tizzy over the next major OS release from Microsoft.

If I may summarize: you can call it Windows 8 now (so speaketh Sinofsky), it'll support ARM chips (at some point), we expect a beta (how public, we'll see), and there will be much hoopla around all this at the freshly announced "Build\Windows" conference scheduled for 13-16 September in Anaheim, CA.

As of this issue going to publication (notice we didn't say, "going to print"), we have no idea what the BUILD conference will entail. Thus, it remains to be seen how much hard-core technical content will make it into the conference agenda, especially what fraction of it may be relevant to the Windows driver developer community.

On the other hand, the availability of a beta of Win8 could make things interesting.

As you might expect, we'll be tracking all this here at OSR, and keeping you updated!

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:

WDK Community Bug Bash

Prize Winners Announced!

Another successful "Bug Bash" is behind us, and we'd like to take another opportunity to thank all those that took the time to participate, and announce the winners here in *The NT Insider*.

First prize winners for the WDK Community Bug Bash were **Girish Aithal Basrur** and **Wei Qing Hao**, who each scored the HP netbook, Vistual Studio Ultimate with MSDN subscription, and free attendance at one of OSR's Kernel Debugging & Crash Analysis seminars.

Second prize winners were **Dejan Maksimovic**, **Ramon Royo**, and **Rod Widdowson**, who didn't do to bad for themselves, winning the iPod Touch, a copy of Windows 7 Ultimate, and an OSR Prize Pack each.

And finally, we had so many excellent entries, we decided to offer two Honorable Mention prizes, which included a copy of Windows 7 Ultimate and choice of choice of hardware from the OSR Online store. These went to **Pradeep Bisht** and **Moso Lee**.

The Bug Bash couldn't be possible without the bug contributors from the Windows driver community, but we've got other important folks to thank as well.

Sponsor ITT Defense & Information Solutions worked with us to organize and carry out the Bug Bash, and their support was key to kicking off the contest. In addition, the cooperation and assistance received from Microsoft both in supporting the contest and in carrying out the back end integration for bug submissions help make everything go smoothly.

Use what you know. Do what you've always imagined.

In 1995, Windows changed the PC.
BUILD will show you that Windows 8 will change **everything.**

//build/ 9.13 - 9.16 Anaheim, CA

BUILD is the event that shows you how to take advantage of the future of Windows. Get insight on creating touch-centric user experiences, fast, fluid, and dynamic applications that leverage the power and flexibility of the core of Windows, used by more than a billion people around the world. See how web-connected and web-powered apps built using HTML5 and JavaScript have full access to the power of the PC.

Explore how the full power of hardware-accelerated Internet Explorer 10 transforms your experiences with the web. BUILD is the first place to dive deep into the future of Windows.

Learn more and register now: http://go.microsoft.com/?linkid=9775865

Peter Pontificates:

Of Clients, Clean Rooms and Copyrights

It would be a significant understatement to say that I hate the whole topic of intellectual property (IP) law as it's applied to software. Yet, a week barely goes by where I don't have to negotiate something about IP with a client. That's part of the "fun" of being a consultant, I guess.

The topic that bothers me most in negotiations with clients is that of driver code ownership. When somebody hires us to write some code, they typically want to own what we've written for them. At first blush, this might seem reasonable – they're paying, they should own it, right? But when you actually stop to think, specifically, about how we write drivers for Windows, ownership is probably both impractical and undesirable. Let me explain.

Unless you're just starting out, almost anyone who develops Windows drivers brings a lot of specialized skill and expertise to the task. You've "been there before." You know what potholes are in the road to completion of a working driver, and you know how to avoid many of those potholes before falling into them. Despite this fact, writing drivers for Windows (or, any operating system, really) largely follows a formula. There are a set of pre-defined entry points. You do a set of common, well-understood, things in these entry points. You do these things in ways that are well defined and agreed. Or, at least, if you're doing a good job that's what you do.

It stands to reason, therefore, that after you've written your first five or ten drivers you've *already written* much of the code that will form the infrastructure of any future driver you'll have to write. And even if you haven't written a dozen drivers on your own yet, there are tons of Microsoft-supplied samples that provide you the basics on which you will build. And if you've written zillions of drivers over a period of almost 20 years like we have here at OSR, you have a lot – and I do mean a *ton* – of code that does all sorts of cool driver-related things.

Having this code, and being able to reuse it, is terrific for everyone involved. It's good for you, because you don't have to start from scratch and write yet *another* clever incarnation of an EvtDriverDeviceAdd routine (or whatever). Thus you are saved from being consigned to the pit of hopeless tedium, coding and debugging stuff that you've written before and understand well enough that you probably won't pay enough attention to writing it the Nth time. It's also good for the people who are paying you to develop the driver, because they get code that's already been written, debugged, and probably even field-proven, and they get it much more quickly than they would otherwise. It's a classic win-win, don't you think? Well, that's what *I* think. Unfortunately, this probably runs afoul of the idea that you can transfer "copyright, plus all right, title and interest" in the code you're delivering to whoever's paying you for it. If you started with a sample from the Windows Driver Kit (WDK) and either modified the sample or cut and pasted parts of the sample into the new driver you're developing, you do not own that code. At best, you've created a derivative work. See the WDK samples? They all have a copyright at the top of each module that looks something like the following:

```
Copyright (c) 1995-1999 Microsoft Corporation.  All
rights reserved.
```

That copyright means that Microsoft owns the code, and is providing you with some specific rights to use it. What are those rights? You need to read the license to know. But you sure as hell don't own the code. And you can't validly give to others what you don't yourself own. Thus, there's no way that you can write a driver using this code and provide "copyright, plus all right, title and interest" in the code to whomever pays you to write the driver. And, by the way, it doesn't matter whether you're working as a consultant or as an employee. If you're an employee, check both your

(Continued on page 5)

Design & Code Reviews

Have a great product design, but looking for extra security to validate internal operations before bringing it to your board of directors? Or perhaps you're in the late stages of development of your driver and are looking to have an expert pour over the code to ensure stability and robustness before release to your client base.

A small investment in time and money in either service can "save face" in front of those who will be contributing to your bottom line. OSR has worked with both startups and multi-national behemoths. Consider what a team of internals, device driver and file system experts can do for you. Contact OSR Sales — sales@osr.com.

Peter Pontificates...

(Continued from page 4)

employment agreement and your company's policies. Your employer might have a clause that says that you will provide unencumbered rights to the work you produce unless you tell them otherwise.

By the way, deleting the copyright and replacing it with your own doesn't change the situation, either, except that deleting the existing copyright hides what is now your theft. And, before you laugh… let me assure you this *happens all the time*. It is not the least bit uncommon for us, here at OSR, to be given a driver to work on that has only the copyright of the company that engaged us at the top – but the code was clearly lifted directly from one of the samples we hand out in one of our classes. Or, the driver is nearly identical to a sample in the WDK.

Assuming you've taken a pre-existing sample, and added some new and original stuff and thus made your new driver substantially different from the sample, you've created what's known as a "derivative work." And you own the new, original, parts that you've added. To signify this ownership, you can add your copyright to any existing copyrights on the work. Congratulations. Go ahead… add that copyright.

But here, once again, things can get tricky in the driver world. Let's say you're working on a driver for a USB device. You start with the KMDF USB FX2 sample in the WDK. But your device implements some unique vendor commands to do clever, device-specific, things. So, you make a few changes to the code – you know, you change the parameters to the WDF_USB_CONTROL_SETUP_PACKET_INIT_VENDOR call. You change the completion routine to do some additional stuff. Whatever. But, you've change the code, right? So, you can own, at the very least, those changes you made?

Well, I'm told it depends. Does changing the values plugged-into a macro make the code original and substantially different?

Or, perhaps you're working on a filter driver where you forward a request and get it back in your completion routine. Are you *really* going to code the way you set up your send options from scratch? You're not going to look for, and maybe cut and paste, an example of calling WdfRequestSend? I mean, how many ways can you even *call* WdfRequestSend to make it "original"? And you've never written that code you're using in the completion routine before?

Here at OSR, if I'm writing code for a driver, I'm probably going to start with one of the drivers we've already written. And as I change that original driver, I'm going to be looking at, and liberally cutting and pasting from, many of the other drivers we've written over the years.

The point is that, save proprietary APIs and the rare invention, most drivers – and particularly most device and filter drivers – have little *truly* new and original content. There are only so many ways to write "status = IoCallDriver(DeviceObject, Irp);" or "WdfRequestCompleteWithInformation(Request, STATUS_SUCCESS, 0);" or many of the other statements that form the vast majority of the driver code that we all write. And much of the delta between the original driver with which we start and the driver code that we deliver might very well have originated from some other project. Again, here at OSR if I need a hash table to look some stuff up based on a string key, I'm going to head right to the depot to see if I can find someplace else where we've already written a string hash table implementation.

To repeat what I said at the outset of this pontification, this is all good – very good, in fact – for everyone involved. This is true as long as nobody decides they need to own the code that's being produced. Who actually owns the code really shouldn't matter. What matters is the license granted to the code by the actual owners.

The license that you get to the code, whether it's code from the WDK or code we at OSR write for a client, dictates how that code can be used. As long as the license allows the user to do with the code what they want (create derivative works, redistribute, sub-license, or whatever) for the length of time they need, ownership shouldn't be an issue.

So, if I'm writing some driver code based on an OSR driver I wrote a few years back, and I grab some hash table code from another project, and then cut and paste and modify some code for my DpcForIsr, as long as I'm in a position to be able to grant the rights required to use that code, all should be well. Ownership is not necessary. Rights are necessary.

So, I'm asking the driver world to face-up to the issues of code ownership and licensing. Please, think a little about these issues the next time you're coding-up a driver for your corporate masters. If you're using code you got from taking a seminar, or from the WDK, or from (heaven forbid!) Code Project, don't just blithely cut, pastes, and delete the copyright. And don't guarantee that you can transfer ownership of code that you never owned in the first place.

And if you're a manager responsible for getting driver code written for your company, I respectfully ask you please: Don't insist on owning the code that's produced. Rather, steadfastly ensure that you get the rights you need licensed to you. In addition, if you have a set of proprietary APIs, be sure you own those APIs. If you're licensing the rights to an invention that'll be used in the code, understand that you still own the invention. Don't put your driver developers or third party partners in the position of having to determine if changing the fields in a hash table package qualifies as making that use substantially different. And, most of all: treat as highly suspicious any guarantees you receive that you'll

(Continued on page 31)

Being Resourceful!

Creating a Proper Version Information Resource

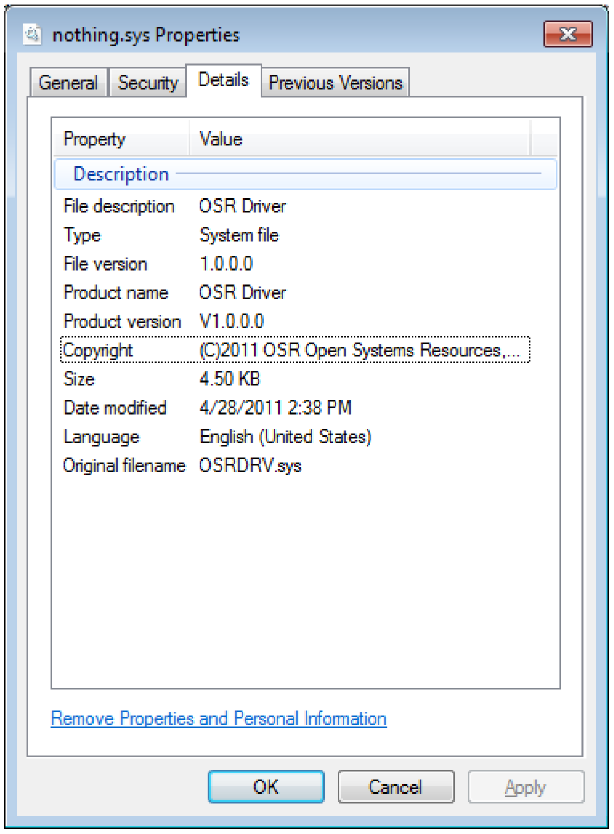
One of our biggest pet peeves is finding a driver installed on a system, written by a 3rd party that either doesn't have a Version-Information resource in the image at all, or has one that is blatantly incorrect. The most common Version-Information resource error that we see is a 3rd party containing a Microsoft copyright string (!) as a result of not overriding the default values supplied in the WDK.

This article will discuss what a Version-Information is and how to create one for your driver.

What is a Version-Information Resource

A Version-Information (VI) resource is a type of Resource Compiler (RC) object that provides information about the executable image that contains it. The information within the VI resource typically includes the executable image's version number, intended operating system, and original filename. This information is viewed in Windows Explorer as shown in *Figure 1*, and also may be retrieved programmatically using the Windows Version Information functions.

The VI resource of an executable is defined by a VERSIONINFO block contained within an RC file (a file with the .RC extension) that is compiled along with the other files that are part of the executable. In the case of a WDK project, the RC file would be declared as an element in the **SOURCES=** line of a project's **sources** file.



Property	Value
Description	
File description	OSR Driver
Type	System file
File version	1.0.0.0
Product name	OSR Driver
Product version	V1.0.0.0
Copyright	(C)2011 OSR Open Systems Resources,...
Size	4.50 KB
Date modified	4/28/2011 2:38 PM
Language	English (United States)
Original filename	OSRDRV.sys

Figure 1—Version Resource as Viewed from Explorer

For readers unfamiliar with an RC file, it is a resource-definition script file (in text format) that describes the resources used by your executable object. For applications, these objects can be cursors, icons, bitmaps, dialog boxes, fonts, version-information, strings, or user-definable objects. For drivers, only the version-information is typically specified in the RC file.

For the purposes of defining a VI resource for a driver, the definitions needed to compile the RC file can be satisfied by including **winver.h**. The RC file is compiled with the resource compiler, which generates a compiled resource (.res) file that the linker links into your driver's executable image. *Figure 2* (page 7) is a typical example of an OSR created RC file that defines a VERSIONINFO resource.

As you can see from *Figure 2* the VI resource contains a number of elements that describe the executable with which it is associated. Let us discuss the VERSIONINFO resource in the next section.

One thing that should be mentioned here: This RC file in *Figure 2* that OSR typically uses *does not* follow the typical format from the examples that you find in a WDK. That's because the OSR resource file is "fully specified" – it's not built using a set of Microsoft-supplied macros. Contrast the OSR file in *Figure 2* with the typical RC file from the WDK shown in *Figure 3* (page 7).

Notice that the OSR created RC does not include "**ntverp.h**" or "**common.ver**". Why? We don't include these files because they contain Microsoft-specific defaults that will be used if you fail to properly set all the fields in your driver's VI resource. These defaults are appropriate for Microsoft developers, but probably not the best choice for others. Driver writers who use a WDK RC file as a template for their RC file are the ones who typically end up with a VI resource that contains things like the default Microsoft copyright because they forgot to specify a copyright of their own.

Now we're not saying that using the WDK-provided method is *bad*. All we are trying doing is warning you of the consequences of using that method. Namely, if you're not careful when you use the WDK-provided method, your RC file you may end up with information in your VI resource that you did not intend to be there.

(Continued on page 7)

Being Resourceful...

(Continued from page 6)

(Continued on page 21)

```
// Include the necessary resources
//
#include <winver.h>
#include <ntdef.h>

#ifdef RC_INVOKED

//
// Set up debug information
//
#if DBG
#define VER_DBG VS_FF_DEBUG
#else
#define VER_DBG 0
#endif

// ------- version info -------------------------------------------------------

VS_VERSION_INFO VERSIONINFO
FILEVERSION             1,0,0,0
PRODUCTVERSION          1,0,0,0
FILEFLAGSMASK           VS_FFI_FILEFLAGSMASK
FILEFLAGS               VER_DBG
FILEOS                  VOS_NT
FILETYPE                VFT_DRV
FILESUBTYPE             VFT2_DRV_SYSTEM
BEGIN
        BLOCK "StringFileInfo"
        BEGIN
                BLOCK "040904b0"
        BEGIN
                VALUE "Comments",         "OSR Driver"
                VALUE "CompanyName",      "OSR Open Systems Resources, Inc."
                VALUE "FileDescription",  "OSR Driver"
                VALUE "FileVersion",      "V1.0.0.0"
                VALUE "InternalName",     "A OSR Written Driver"
                VALUE "LegalCopyright",   "(C)2011 OSR Open Systems Resources, Inc."
                VALUE "OriginalFilename", "OSRDRV.sys"
                VALUE "ProductName",      "OSR Driver"
                VALUE "ProductVersion",   "V1.0.0.0"
        END
        END
        BLOCK "VarFileInfo"
        BEGIN
                VALUE "Translation", 0x0409,1200
        END
END
#endif
```

Figure 2—An Example of a Simple Resource File

```
#include <windows.h>
#include <ntverp.h>

#define VER_FILETYPE                VFT_DRV
#define VER_FILESUBTYPE             VFT2_DRV_SYSTEM
#define VER_FILEDESCRIPTION_STR     "WDM Driver for Intel 8255x Ethernet Adapters"
#define VER_INTERNALNAME_STR        "PCIDRV.sys"
#define VER_ORIGINALFILENAME_STR    "PCIDRV.sys"

#define VER_FILEVERSION             1,00,00,0000
#define VER_FILEVERSION_STR         "1.00.00.0000"

#undef VER_PRODUCTVERSION
#define VER_PRODUCTVERSION          VER_FILEVERSION

#undef VER_PRODUCTVERSION_STR
#define VER_PRODUCTVERSION_STR      VER_FILEVERSION_STR

#define VER_LEGALCOPYRIGHT_STR      "Copyright (C) 2003 Microsoft Corporation"
#ifdef VER_COMPANYNAME_STR

#undef VER_COMPANYNAME_STR
#define VER_COMPANYNAME_STR         "Microsoft Corporation"
#endif

#undef VER_PRODUCTNAME_STR
#define VER_PRODUCTNAME_STR         "Microsoft Sample Driver for PCI Device"

#include "common.ver"

PciDrvWMI MOFDATA pcidrv.bmf
```

Figure 3—WDK Sample RC File

Win7 Crash Redux

In the last issue of *The NT Insider* we published a write-up describing analysis of a crash dump in filter manager. A couple of readers commented about the analysis and had some solid points that needed to be considered.

This also underscores an important aspect of analysis, namely that it can be helpful to obtain a second opinion on one's analysis, precisely because it is easy to miss some important point that aids in the analysis.

Now on to the specific issues raised by the readers:

- The value of the **RSI** register is in fact not null (**0000009d00000000**)
- There is a backwards jump in the instruction stream that impacts on the code flow analysis
- There is some useful information related to determining the *type* of trap frame that in turn tells us more about which registers are valid.

x64 Trap Frame Observations

On the x64 platform, a kernel trap frame does not capture all register state:

- For a processor exception or trap, the OS only captures the **volatile** registers (**RAX, RCX, RDX, R8-R11 and XMM0-XMM5**) and the **RBP** register.
- For a system call entry, the OS only captures **RBP, RSI** and **RDI**. No other registers are preserved.

By using the **ExceptionActive** field, we can determine which type of trap frame this is (a 0 or 1 value indicates this is an exception or trap and the volatile registers plus RBP are stored).

Further, the reader observed:

You can very often get reliable nonvolatile registers on x64 from ".frame /r (frame number)". There is also the newly-documented (but long-present) ".frame /c (frame number)" command that sets your effective context to the values obtained from .frame /r. This works using the unwind metadata generated by the x64 compiler that the debugger stackwalker uses (it also works for Itanium, if you should be debugging that, but not x86). It should _always_ give you correct nonvolatile registers if you start from the context obtained by .cxr, .thread, or .exptr.

This was useful information in general, and hopefully will help our readers further hone their debugging skills into the future.

Code flow analysis

This reader had some valid observations here:

The first constructive point is that if your debugging takes you into the game of back-tracing, you need to study whole functions. This means unassembling not just at addresses before the faulting instruction, nor after, but at all places that fragments of the function have got scattered by optimisation. Basically, you need to raise your debugging to the foothills of reverse engineering. The reverse engineer will see that the faulting instruction, "mov rax,qword ptr [rsi+20h]" at ...F141 is picked up for TreeUnlinkMulti by inlining TreeUnlinkMultiDoWalk, which in turn inlines TreeLookup, which in turn inlines TreeFindNodeOrParent. The loop that the analyst has missed is actually from the start of this last subroutine. The code's overall intention is to walk a given tree, remove the nodes that match a given pair of keys, and return these nodes as a list (linked through the RightChild members only).

The reality is that with x64 it is proving to be far more often the case that we need to back track through the code flow in order to find local variables and reconstruct the stack. When doing a thorough analysis, it is indeed important to look at the entire function (the **uf** function is good for this) but this is a bit more time-consuming (and certainly more daunting to those just approaching kernel debugging).

But these are valid points.

So with this said, let's go back and revisit our analysis (see *Figure 1*).

(Continued on page 9)

```
2: kd> .cxr fffff88005f45960
rax=fffffaf7072f96b0 rbx=0000000000000000 rcx=fffffa8008e13318
rdx=fffffa8007e5b550 rsi=0000009d00000000 rdi=0000000000000000
rip=fffff8800106f141 rsp=fffff88005f46330 rbp=fffffa8008e13318
 r8=ffffffffffffffff  r9=ffffffffffffffff r10=fffffffffffffe4a
r11=0000000000000001 r12=fffffa8007e5b550 r13=fffffa8007e34684
r14=0000000000004000 r15=0000000000000000
iopl=0         nv up ei pl nz na po nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00010206
fltmgr!TreeUnlinkMulti+0x51:
fffff880`0106f141 488b4620        mov     rax,qword ptr [rsi+20h] ds:002b:0000009d`00000020=????????????????
```

Figure 1— Context Record

Win7 Crash Redux...

(Continued from page 8)

And of course the registers are valid here (they are all captured in the context record,) as our reader noted.

This gives us the stack shown in *Figure 2*.

Then let's look at the invalid address (*Figure 3*).

This decodes the address, finds the relevant page table entries and decodes each of them. From this, we can tell there is nothing within this 512GB memory region (since each PXE entry corresponds to a 512GB region of the address space).

Thus, while not the null pointer indicated previously, this is still an invalid address – within a large, undefined region of the address space.

As you so choose, you can look at the first function from the stack *in its entirety* as provided at http://www.osronline.com/article.cfm?id=584.

(Continued on page 25)

```
2: kd> kv

  *** Stack trace for last set context - .thread/.cxr resets it

Child-SP          RetAddr           : Args to
Child                                                     : Call Site

fffff880`05f46330 fffff880`0106c460 : fffffa80`07a96920 fffffa80`07e5b550 fffffa80`07a96920
00000000`00000000 : fltmgr!TreeUnlinkMulti+0x51

fffff880`05f46380 fffff880`0106cbe9 : fffff880`05f48000 00000000`00000002 00000000`00000000
00000000`00000000 : fltmgr!FltpPerformPreCallbacks+0x730

fffff880`05f46480 fffff880`0106b6c7 : fffffa80`08b93c10 fffffa80`07ca8de0 fffffa80`07b402c0
00000000`00000000 : fltmgr!FltpPassThrough+0x2d9

fffff880`05f46500 fffff800`02da278e : fffffa80`07e5b550 fffffa80`07dfa8e0 fffffa80`07e5b550
fffffa80`07ca8de0 : fltmgr!FltpDispatch+0xb7

fffff880`05f46560 fffff800`02a918b4 : fffffa80`07e34010 fffff800`02d8f260 fffffa80`06d17c90
00000000`ff060001 : nt!IopDeleteFile+0x11e

fffff880`05f465f0 fffff800`02d900e6 : fffff800`02d8f260 00000000`00000000 fffff880`05f469e0
fffffa80`08b93c10 : nt!ObfDereferenceObject+0xd4

fffff880`05f46650 fffff800`02d85e84 : fffffa80`07c3fcd0 00000000`00000000 fffffa80`07a17b10
fffffa80`0a31e701 : nt!IopParseDevice+0xe86

fffff880`05f467e0 fffff800`02d8ae4d : fffffa80`07a17b10 fffff880`05f46940 0067006e`00000040
fffffa80`06d17c90 : nt!ObpLookupObjectName+0x585

fffff880`05f468e0 fffff800`02d1ee3c : fffffa80`08cf07e0 00000000`00000007 fffffa80`00001f01
00001f80`00f40200 : nt!ObOpenObjectByName+0x1cd

fffff880`05f46990 fffff800`02a8b993 : fffffa80`0a31e7e0 00000000`00000000 fffffa80`0a31e7e0
00000000`7ef95000 : nt!NtQueryFullAttributesFile+0x14f

fffff880`05f46c20 00000000`77320eba : 00000000`00000000 00000000`00000000 00000000`00000000
00000000`00000000 : nt!KiSystemServiceCopyEnd+0x13 (TrapFrame @ fffff880`05f46c20)

00000000`0121e778 00000000`00000000 : 00000000`00000000 00000000`00000000 00000000`00000000
00000000`00000000 : 0x77320eba
```

Figure 2— And, the Resulting Stack

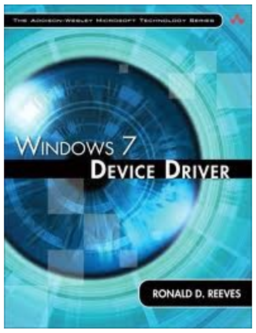
```
2: kd> !pte 0000009d00000000
                                           VA 0000009d00000000
PXE at FFFFF6FB7DBED008    PPE at FFFFF6FB7DA013A0    PDE at FFFFF6FB40274000    PTE at FFFFF6804E800000
contains 0000000000000000
not valid
```

Figure 3— Reviewing the Invalid Address

One Book, Four Reviews

Windows 7 Device Driver, by Ronald D. Reeves

As our first book review in some time, we though it appropriate to bring in a couple of vocal and recognized names in the Windows driver development community to help us out. We're pretty sure that the following reviews will raise awareness of this particular book, and provide you insight so that you can determine if it is deserving of a place on your shelf.



Review #1: "Save Your Money"

By Don Burn

The preface of the *Windows 7 Device Driver* book states, "This book provides the technical guidance and understanding needed to write device drivers…". Unfortunately it fails in this goal. The preface is also notable in that it acknowledges no technical reviewers, which as you read the book becomes obvious from its many flaws.

The book is split into three sections: an introductory section, a section on UMDF, and a section on KMDF. The introductory section clearly shows the failings of this book. The first chapter of the introduction covers standard object oriented programming which, while nice, has little applicability to the rest of the book. The second chapter is an extremely poor overview of Windows driver concepts. What is missing from the introductory section is an overview of WDF concepts. This is obvious later in the book. The author duplicates explanations of these concepts in both the UMDF and KMDF section. Neither explanation is complete so at times you need to read both to get what the author is saying.

Once you get into the UMDF and KMDF sections, things do not improve. The author repeats diagrams and text multiple times, including cases where the same text is repeated on two consecutive pages. If this were a large book this might be excusable, but with only 86 pages dedicated to UMDF and 192 pages to KMDF every page should count.

When the author is giving useful information, it corresponds directly with the WDK documentation or various Microsoft papers on WDF. The examples in the book are all WDK

(Continued on page 13)

Review #2: "Full of Inaccuracies"

By Scott Noone

When I first heard of an upcoming book published by Addison-Wesley on writing Windows drivers I was intrigued. Students are *always* asking me for book recommendations and, quite frankly, there just aren't that many books out there to be recommended. This new book might be the salvation that those students were looking for and might provide people the information they need to get up and running with the Windows Driver Foundation.

Then, unfortunately, I received my copy of the 350 page book. I can honestly say I'm not quite sure who this book is targeting or the process that was used to create it, but I will definitely not ever be recommending this to anyone.

Editing, Anyone?

I generally try not to harsh on people too bad for typos, inconsistencies, and obvious grammar problems. I mean, we all make mistakes, right? However, to review this book and not mention the many, many editing issues would be criminal. Heck, I'm pretty sure that *Dan* could do a better job editing, and that's saying a lot.

In order to see an example of what I mean, you need look no further than the preface of the book. This book alleges to teach the reader about WDF, however the author can't even decide what WDF stands for. First we're told:

(Continued on page 14)

Review #3: "Based on Existing Docs"

By Martin O'Brien

If you're looking for a terrible summary of a random collection of WHDC whitepapers, thoughtfully and extensively annotated with a whole bunch of errors courtesy of the author, with no complete examples, or original examples, and really nothing specific at all on any subject, written by someone with a PhD who knows nothing about Windows driver development, then this is the (wholly unedited) book for you.

To be fair, other than that, I have to say that I really did enjoy the play.

You know, with a book this crappy, it's very tempting to just skim. I mean, really, what are you going to miss? I ask you this, fair reader.

Alas, I did read the whole thing. And now I'm angry.

There are three things that strike me most about this book, other than its crappiness. The first is that is essentially a rehash of existing material. The second is its almost complete lack of detail. The third is that its rehash nature notwithstanding, it is full of errors of all types.

The degree to which this book is a rehash of WHDC papers is striking. Seriously, were it not for the fact that Penny Orwick writes good stuff, I would think that she was getting kickbacks. The most obvious source of information, however, is the WDK toaster sample, which is what the author terms the *Featured Toaster* driver. The book is mostly about said *Featured Toaster* driver, but the question of which version of the WDK Dr. Ron used is a good one. The title of the book is of course, *Windows 7 Device Driver*, but the good doctor clearly did not use the Win 7 WDK. In some places, he refers to the earlier of the Vista RTM WDK's (6000), but there, as in most places, there are so many errors and contradictory pieces of information, it's hard to say really (Page 190).

Launch the Windows 7 Checked Build Environment console window. It opens in the c:\WinDDK\6000 folder.

Well, no, Ron, it does not, because there is no 'Windows 7 Checked Build Environment' console window in the Vista WDK, oddly enough. Damn Softies.

To my way of thinking, there's so many errors of this type – referencing existing material and still managing to introduce errors – that there's really not much point in detailing them, not to mention that it wouldn't be possible to do this in the space provided. They are everywhere. Seriously, just open to pretty much any old page and start reading.

For the same reasons, I think it somewhat a fool's errand to break down how the author handles the finer points of driver development, not to mention what he doesn't attempt to do so.

What I *do* think is a laudable goal for a book like this is to take a bite out of the fairly terrible learning curve of getting a driver built, installed and running under a KD session. Admittedly, the last of those is not by a long shot an easy one to document, but the first ones are certainly all addressable.

To that end, the good doctor actually had a pretty good approach using the *Featured Toaster* driver (of course). That is, use an existing driver that everyone has access to as a starting point and detail the finer points of the development process from there. Had he followed through on this, I think that would have been great. To date, to the best of my knowledge, there really isn't any document out there that collects all the bits and pieces you need to know to get up and running quickly. That would be well worth $50, I think.

(Continued on page 12)

Review #4: "Have a Nice Day"

By Peter Viscarola

Don, Martin, and SNoone *had* to read all of this book to review it properly. Because I knew they were reading it, and would write thorough, intelligent, complete, reviews, I stopped reading and started skimming about mid-way through the first chapter. I did this because I knew my opinion wouldn't change. This book sucks. It's useless. The hideously bad editing and the obvious lack of practical insight into the topic offend me as professional driver developer and author. How any publisher could let this piece of shit out the door is beyond my understanding. So, I guess that means I didn't like it, and I recommend you don't buy the book. Have a nice day.

Peter Viscarola is a Consulting Partner with OSR.

Book Reviews (O'Brien)...

(Continued from page 11)

The opposite side of this coin, however, is that if you're going to blatantly reproduce existing material, you need to (a) do so correctly and (b) provide said bits and pieces that tie it all together.

This is where the breakdown occurs.

Using another build example (Pages 187 - 188), although he mentions 'makefile.inc' many times, and provides a SOURCES file that uses it, he never explains how it really gets used. Further both his description and the SOURCES files have errors. For example, he describes 'NTTARGETFILES' as being used to 'specify additional targets and dependencies not covered by makefile.def.' This is incorrect, of course – it causes the inclusion of 'makefile.inc.' – and makefile.def has exactly nothing to do with it. If you want to go that route, you'd need to say 'makefile.new,' (makefile.def defines no targets, dependencies or much of anything else) but that too isn't really relevant. What would be relevant is describing how it works, which he fails to do. He also has included both NTTARGETFILES and NTTARGETFILE0 in his SOURCES, though has misspelled the latter as 'NTTARGETDILE0,' and he used 'KMDF_VERSION' instead of 'KMDF_VERSION_ MAJOR,' with the former having been marked as deprecated for a few WDK's.

Just to state the obvious, all he had to do was look at the existing SOURCES file in the Toaster sample.

The lamest example of conspicuous reproduction of existing material, by far, is his reproduction of the BUILD command line options. He lists five of them – including '?' – and provides no explanation for what they do, other than his mangled version of what typing 'build /?' produces (page 186).

The thing that I find most disappointing about the author's treatment of this topic is that as any regular on ntdev/ntfsd/ windbg (nttalk, not so much, probably) knows, both BUILD and WINDBG are *filled* with 'subtleties' that collectively provide a veritable kaleidoscope of low hanging fruit for a treatment such as this, and were they provided, they would be of legitimate benefit to the beginner.

Another area which would be a good candidate for an introductory book, in my opinion, is background theory/ architecture of Windows. Here too, the results aren't good, though I'll give the author credit for originality. It's not every book on Windows drivers that starts off with a discussion on the 'Nature of an Object' (page 7), let alone one that references Kant, among others, and his indirect contributions to Object Oriented Programming due to his involvement in 'explaining the meaning of existence in general.'

In addition to it being, say, wholly irrelevant, Ron, buddy, you're like five time zones behind your own ass – describing everything under the sun and the Windows kernel in particular as 'object oriented' was in fashion maybe five years ago, at best, and probably more like ten. Or at least pretending to think it important was.

As far as the accuracy of the introductory material goes, it's more of the same.

There are several confusing/incorrect diagrams. The most bewildering to me, at least, is *Table 2.1 WDF Device Support for Windows 7* (Page 16). It has four columns that not many would normally group together, in my opinion – 'KMDF, 'UMDF,' 'SDV' and 'PREfast.' Using this table, you can determine, for example, that neither KMDF nor UMDF nor SDV support 'Video Capture Devices,' but PREfast in fact DOES. Well, I guess I'll just write my driver using the PREfast model then.

Finally, while I suppose that this might be seen by some as a strange point to focus on, I think that one of the most admirable qualities in a book is a good index. This is darn rare these days, with everything being auto generated. Call me a geek, but I find a good index to be ones of life's true pleasures. This book's index is not good, but that's mostly because there's no material to reference.

What is positively *shameless* is the *Driver Information Web Sites* appendix (page 323). Ever find yourself looking for, say, 142 permutations of *go.microsoft.com*? Well, here you go! In a nutshell, the author – literally – provides links (ever so helpful in hardcopy) to various pages in the WDK and calls each a 'web site.' Whether the links even work due to the MSFT link permutator, who knows. There's also almost no third party 'web sites' referenced, though he does manage to get *two* OSR references in there. Unfortunately, they happen to be for the *same* page (numbers 1 & 137, something to do with WinUSB).

I guess my summary would be that this book was written by someone who would appear to know nothing about developing (let alone testing) Windows drivers, and instead based his entire book on an academic reading of existing documents without trying any of it.

Martin O'Brien is a research scientist/business developer at a company that is owned by another company that might not care for the saltiness he expressed in this review. They're really quite narrow. He can be reached at: martin.matthew.obrien@gmail.com

Book Reviews (Burn)...

(Continued from page 10)

example code. There is not a single line of code by the author. In fact, in reading the book you wonder if the author has ever written a Windows device driver; in many places he puts emphasis on items not required to write a driver, then skims over important concepts. You also have to wonder if the author has ever attended a conference such as DDC and WinHEC, or for that matter actively monitored the various online discussion groups covering Windows driver writing; because as you read the book, various approaches that have appeared in the conferences or discussion groups are never mentioned.

The book has way too many flaws to mention all of them, but a few lowlights that struck me include:

- The book is based on the Vista WDK; things like KMDF_VERSION are still specified when they have been replaced and in the Windows 7 WDK produce a warning
- The author at one point references IRQL, but never explains what it is, and rarely refers to it again for the rest of the book. In fact, he waits until forty pages before the end of the book to mention that you should not execute code that can cause a page fault at DISPATCH_LEVEL.
- The book gives no examples of how to access hardware. Instead, in a review of the PCIDRV driver he shows the calls to the wrapper routines the driver uses, but nothing about required macros and functions like READ_PORT_XXX, READ_REGISTER_XXX, WRITE_PORT_XXX, etc.

The text has a short description of reading from the registry, but no explanation of what the registry is, Further, it only shows an example of reading the device parameters under the hardware key and nothing about reading parameters under the services key for the whole driver.

The book's cover claims, "The First Authoritative Guide to Writing Robust, High-Performance Windows 7 Device Drivers". The book is far from authoritative, does not provide the information to allow creation of a robust driver, presents no information on performance, and does not use the Windows 7 WDK, so it cannot claim to specialize in Windows 7 drivers.

New developers could use a good introductory book on WDF to get them going. This book is not it. For experienced developers, this book is a total waste of time, since it covers things at a very high level with no new information. Get the Microsoft WDF book if you do not have it, but save your money and don't waste your time and dollars on this book.

Don Burn is a consultant with 40 years of system software development experience. Don has architected multiple system software products, including operating systems, tools, and compilers. For the last 17 years, he has been developing Windows device drivers and file systems.

As a consultant, Don has developed system software for a number of clients, and has been engaged for architecture definition, assistance in hiring a development team, and review of development processes. Don can be reached at http://www.windrvr.com

Book Reviews (Noone)...

(Continued from page 10)

Much of the previous grunt work, thank goodness, is now being handled by the latest device development framework Windows Driver Foundation (WDF).

While I do have other problems with that sentence, I'm going to ignore those and instead focus on the correct definition of WDF as, *Windows Driver Foundation*. Unfortunately, three paragraphs later the initials are redefined to be something else: *This section also covers…which are part of the Windows Driver Framework (WDF).*

While this may seem like a minor nit, it is demonstrative of two things that become very clear when reading this book:

- There's a general lack of attention to detail. For example, page 19 calls the WDFDMAENABLER object a, "WDFDMANENABLE" object and page 189 has the SOURCES file INCLUDE directive listed as, "INNLUDES". Note that these are not isolated instances, but examples found while flipping through the book as I write this review.

- Before becoming the *Windows Driver Foundation*, WDF was initially introduced to the community as the *Windows Driver Framework*. I suspect that the author's confusion comes from ancient WinHEC slides and other material that introduced it as such. This might be considered an honest mistake. However, there are other cases where the author has used old white paper and WinHEC material that do not represent the current state of the framework. For example, the block diagram shown on page 25 is an exact copy of a block diagram from a whitepaper published in 2006 using antiquated, "package" terminology for portions of the framework. Instances like this give the book a, "cobbled together" feeling that is never good from a technical work.

With all of that said, we can begin looking at the content of the book as a whole.

Where to begin?

The book starts off with what can best be described as a bizarre commentary on the "nature" of objects. I find that this section is best summed up by the following sentence from the second paragraph:

> *Ancient philosophers, such as Plato and Aristotle, as well as modern philosophers like Immanuel Kant have been involved in explaining the meaning of existence in general and determining the essential characteristics of concepts and objects (Rand 1990).*

Huh? I'm not sure what, if anything, this has to do with writing drivers. While this might be an interesting mental exercise for a college text on OO programming, this is supposed to be a practical book about writing drivers for Windows. You would think that the author would have taken this time to even briefly explain concepts such as IRQL, execution context, driver objects, device objects, etc. Instead, we're left with twelve pages discussing UML and .NET.

Who is the Target Audience?

As I made my way through this book, I began wondering who exactly the target audience would be. While the second chapter, *WDF Object Model*, starts off fairly benign with a high level overview of why we need a replacement for WDM, the author then immediately starts using very specific WDM details that would totally confuse anyone that didn't have prior driver writing experience. For example, in section 2.5.1 the author states:

> *KMDF objects are unique to the framework. They are not managed by the Windows object manager and therefore cannot be manipulated by using the system's* ***ObXxx*** *functions.*

Thus, there is an assumption that the reader has an understanding of the Object Manager in Windows and the available Object Manager APIs. As another example, Section 2.7.2 assumes that the reader is familiar with the I/O Manager and IRP processing:

In Windows, the I/O request packet (IRP) does more than just present traditional I/O requests (read, write, create, and so forth) to drivers.

Note the lack of context for these terms; they seem to come out of nowhere. Also, what good does this information provide the reader? While initially learning the framework it can often be a *disadvantage* to try to think of the framework in terms of the underlying WDM objects. This can cause the student to lose focus on the framework itself and instead start worrying about the implementation of the framework.

Unless we are to expect newbie driver writers to identify and ignore WDM references that they don't understand, I can't recommend this book to an inexperienced driver writer. It lacks the fundamentals necessary to give a solid understanding on the O/S environment and will mostly just serve to confuse with its mixing of terms.

What about the Experienced Driver Writer?

So, if it's not for newbies, is this book for those experienced WDM driver devs out there looking to quickly come up to speed on WDF for their next project? Unfortunately, no, this book isn't for the experienced driver dev either. As I mentioned previously, this book is mostly a collection of the various whitepapers currently available and thus does not provide much content that you haven't seen before. Did I also mention that this book is brief? For example, the author

(Continued on page 15)

Book Reviews (Noone)...

(Continued from page 14)

spends a whopping 31 pages describing the KMDF object and I/O model.

In addition, the KMDF sections do a poor job of highlighting the typically troublesome concepts of synchronization scopes, queue dispatching, execution level constraints, and the implications that these can have when combined. For example, while explaining **WdfExecutionLevelDispatch** the author states:

> *This setting does not force all callbacks to occur at **DISPATCH_LEVEL**. However, if a callback requires synchronization, KMDF uses a spin lock, which raises **IRQL** to **DISPATCH_LEVEL**.*

Clearly the author is trying to make some kind of point here about the impact of synchronization scope and execution level constraints, though the point is poorly formed and does little to clear up any confusion the reader might have.

The back of the book alleges that this book has a section on how to write secure kernel code, which I thought would be an excellent addition to any driver writing book and was excited to read about as an experience dev. Unfortunately, the security section is all of one page and has useless information such as:

> ***7.5.3 Counted UNICODE Strings***
> *To help prevent string handling errors, KMDF DDIs use only counted **PUNICCODE_STRING** (sic) values. To aid drivers in using and formatting **UNICODE_STRING** values, the safe string routines in **ntstrsafe.h** have been updated to take **PUNICODE_STRING** parameters.*

That's the entire section on safe string handling. Seriously.

Downhill from Here...

The rest of the book consists of dramatic readings of several of the KMDF samples from the Vista RTM WDK. I'm not sure what value is being added here, especially with sentences such as (page 230):

> *The variable **fdoData**, of type **PFDO_DATA**, is defined to hold a pointer to the context area.*

The End (Finally)

The book ends as strangely as it began, with an appendix that contains a list of 142, "driver information web sites." In reality, it's mostly an eclectic collection of random MSDN pages. Things such as ACCESS_MASK and SECURITY_IMPERSONATE_LEVEL (sic) are listed here, which don't really appear to match up with anything in the book. Though we do have the same article from *The NT Insider* listed twice in the appendix, so that must make it extra important.

In case it hasn't been clear from this review, *do not buy this book.* There just isn't anything here that you haven't seen before and what *is* here is full of inaccuracies.

Scott Noone is a Consulting Associate with OSR.

Custom Software Development—Experience, Expertise ...and a Guarantee

In times like these, you can't afford to hire a fly-by-night Windows driver developer. The money you think you'll save in hiring inexpensive help by-the-hour, will disappear once you realize this trial and error method of development has turned your time and materials project into a lengthy "mopping up" exercise...long after your contract programmer is gone.

Consider the advantages of working with OSR. If we can be of value-add to your project, we'll tell you. If we can't, we'll tell you that too. You deserve (and should demand) definitive expertise. You shouldn't pay for inexperienced devs to attempt to develop your solution. What you need is fixed-price solutions with guaranteed results. Contact the OSR Sales team at sales@osr.com to discuss your next project.

WDF Queues...

(Continued from page 1)

Queue can be in progress in your driver at a time, and how those Requests are presented. The possible Queue Dispatch Types are:

- **Sequential Dispatch Type**
  In Sequential Dispatching, only one Request from the Queue may be in-progress in your driver at one time. A new Request is not presented to your driver until your driver:

  - Completes the current Request (regardless of the completion status)
  - Forwards the current Request to another Queue
  - Sends the current Request to an I/O Target, using the Send and Forget option.

- **Parallel Dispatch Type**
  When you specify Parallel Dispatching for a Queue, multiple Requests can be in progress in your driver from that Queue simultaneously. Instead of waiting for one Request to finish before presenting your driver with another Request, as in Sequential Dispatching, in Parallel Dispatching the Framework will continue to present your driver with Requests from the Queue until some maximum number of Requests are in progress from that Queue. You specify the maximum number of Requests that can be in progress at one time when you configure the Queue. By default, the maximum number of Requests is "unlimited."

  If you're clever, you'll note that using Parallel Dispatching and setting the maximum number of in-progress Requests to one gets you exactly the same behavior as what you get from Sequential Dispatching. Good catch!

- **Manual Dispatch Type**
  Manual Dispatching is very different from other Queue Dispatch Types. When you specify Manual Dispatching, Requests are explicitly retrieved from the Queue by your driver; the framework never calls your driver to present Requests from the Queue.

  Because the Manual Dispatch Type requires a driver to actively "pull" Requests from the Queue, Queues with this Dispatch Type are almost never used for delivering Requests from the Framework to a driver. Consider why this is the case: If you use a Queue with the Manual Dispatch Type to receive Requests from the Framework, your driver would have to poll the Queue – periodically attempting to remove entries – in order to get work. When compared to the other Dispatch Types, which present Requests to your driver when appropriate, this isn't a very appealing alternative.

  Queues with the Manual Dispatch Type are almost always used to hold Requests that have been previously presented to a driver via another Queue. Thus, Queues with the Manual Dispatch Type are most like ordinary queues or linked lists. They're used to hold items that originated elsewhere until needed.

So, a Queue's Dispatch Type controls both how Requests are delivered to your driver (by presenting Requests to your driver, or by your driver manually calling a function to remove a Request from the Queue) and how many Requests may be in progress in your driver at a time (for Sequential Dispatching one Request can be in progress at a time, for Parallel Dispatching multiple requests – up to a limit defined by your driver – can be in progress at a time). Queues with Sequential Dispatching or Parallel Dispatching are most often used to control the flow of Requests from the Framework to a driver. Manual Dispatching are typically used to temporarily hold Requests that a driver has received via another method.

(Continued on page 17)

WDF Queues...

(Continued from page 16)

Queue States

Another way in which WDF Queues differ from ordinary linked lists is that WDF Queues are actively controlled by the Framework. Each WDF Queue may be in one of several different states:

- **Started**: If a Queue is in the *Started* state, when a Request arrives the Queue will immediately consider it for presentation to the driver according to the rules for its Dispatch Type. For example, if a new Request arrives at an empty Queue in *Started* state, and that Queue uses the Sequential Dispatch Type, the Request will be presented to the driver if there are currently no other Requests active in the driver from that Queue.

- **Stopped**: When a Queue is in the *Stopped* state, newly arriving Requests will be inserted on the Queue and held on the Queue indefinitely. If the Queue's state changes to *Started*, Requests pending on the Queue will be evaluated for presentation according to the rules for the Queue's Dispatch Type.

  Therefore, if the previously described empty Sequential Queue is in the *Stopped* state when a Request arrives, that Request will be placed on the Queue. Later, if the Queue's state changes to *Started* the first Request that arrived to the Queue will be presented to the driver. Because the Queue in this example uses the Sequential Dispatch type, additional Requests will not be presented until the previously presented Request is completed or forwarded as previously described.

- **Purged**: When a Queue is in the *Purged* state any Requests that are on the Queue or that arrive for that Queue are immediately completed by the Framework with an error status. Therefore, if a Queue that was in the *Stopped* state is changed (by command from either the driver or the Framework) to the *Purged* state, any Requests that happened to be waiting on the Queue are completed by the Framework with an error status. Any new Requests that arrive for this Queue are also completed by the Framework with an error status; they are not inserted onto the Queue.

Power Managed

An extremely handy feature of WDF Queues is that they are "power managed" by default. This meaning that the Framework will automatically change the Queue's state according to the D-State of the device with which the Queue is associated.

When a device is in the fully powered, working (D0), state the Framework sets that device's Queues to the *Started* state. This results in arriving Requests being presented to the driver according to the rules for the Queue's Dispatch Type, as previously described. When the device transitions to a non-working power state (that is, any device power state other than D0), the Framework will automatically put the Queue into the *Stopped* state, resulting in Requests arriving at the Queue being held. In addition, if a Request arrives at a power managed Queue while the device is idling in a low power state (any state other than D0, as a result of the device putting itself to sleep to save power) the Framework will automatically initiate the process to return the device to the working (D0) power state so it can resume processing Requests.

This is a terrific feature, because it frees drivers from having to be concerned about what power state the device is in when Requests arrive. You certainly don't want read or write requests to be presented to your driver before your driver has had the chance to power-up your device to handle them, right? Right! And, with power managed Queues, the Framework keeps that from happening.

Of course, you can override the Framework's default behavior by setting the Queue to be non-power managed. In this case, the state of the Queue isn't changed automatically based on the associated device's D-state.

Automatic Cancellation

A final feature of WDF Queues that makes WDF driver development convenient is that Queues handle cancellation of pending Requests automatically. This means that if a Request is cancelled when it is pending on a Queue the Framework will automatically remove that Request from the Queue and

(Continued on page 18)

WDF Queues...

(Continued from page 17)

complete it with a cancelled status. A driver can choose to be informed about the cancellation by specifying an appropriate Event Processing Callback (discussed later). Reasons that a Request might be cancelled when it's on a Queue include the thread that initiated the Request exiting, or the issuing thread calling CancelIo or the issuing process calling CancelIoEx.

Queue Events and Event Processing Callbacks

So how exactly are Requests presented to your driver? It's simple: Like all WDF Objects, Queues are capable of raising a given set of events. A driver may choose to handle a subset of these events by providing appropriate Event Processing Callbacks.

The most important Queue-based Event Processing Callbacks available to your driver are the I/O Event Processing Callbacks. These callbacks are invoked by the Framework to present Requests from a Queue to your driver. On each of these callbacks, the Framework passes your driver a handle to the Queue from which the Request originated, and a handle to the Request itself. Other parameters vary, depending on the specific callback.

Note that I/O Event Processing Callbacks are only used for Queues that have been configured with the Sequential or Parallel Dispatch Type. Queues that are configured with the Manual Dispatch Type do not raise I/O events, and thus I/O Event Processing Callbacks are not allowed for these Queues. In fact, the Framework returns an error if you try to specify an I/O Event Processing Callback for a Queue with the Manual Dispatch Type.

The I/O Event Processing Callbacks that your driver can handle are:

- **EvtIoRead**: This callback is invoked by the Framework when it has a read Request to be presented to the driver from the Queue.
- **EvtIoWrite**: This callback is invoked by the Framework when it has a write Request to be presented to the driver from the Queue.
- **EvtIoDeviceControl**: This callback is invoked by the Framework when it has a Device I/O Control (IOCTL) Request to be presented to the driver from the Queue.
- **EvtIoInternalDeviceControl**: This callback is invoked by the Framework when it has an Internal Device Control Request to be presented to the driver from the Queue.
- **EvtIoDefault**: This callback is invoked by the Framework when it has a Request to be presented to the driver from the Queue, and one of the more specific I/O Event Processing Callbacks has not been specified by the driver.

If you think about it a bit, I expect you'll understand how the above I/O Event Processing Callbacks are used. Consider, for example, a driver that has a single Queue using Parallel Dispatching for which EvtIoRead, EvtIoWrite and EvtIoDefault Event Processing Callbacks have been supplied. If a read operation is sent to the device, the Framework will present that Request to the driver by calling the EvtIoRead Event Processing callback. Likewise, if a write operation is sent to the device, the Framework will present that Request by calling the driver's EvtIoWrite Event Processing Callback. However, if a Device Control Request arrives at the Queue, the Framework will present that Request via the driver's EvtIoDefault Event Processing Callback, because the driver did not configure the EvtIoDeviceControl Event Processing Callback to handle this specific type of Request.

There are also Event Processing Callbacks that allow your driver to be informed when a Queue transitions into and out of the *Stopped* state. These callbacks are only needed in rare cases, and are thus much less frequently used than the I/O Event Processing Callbacks.

Before leaving our discussion of Event Processing Callbacks, we should note one more type of Event Processing Callback that your driver can specify: That's the Event Processing Callback for Request cancellation. If a Request is currently on a Queue and is aborted (either as a result of the user attempting to cancel it or the thread that issued it attempting to exit), the Framework will call a driver's EvtIoCanceledOnQueue Event Processing Callback. Note that this callback is only invoked when a Request is canceled while on a Queue (or immediately before being queued), thus it's not likely to provide much value in the case of a parallel Queue.

Incoming Request Queues

A driver is free to create as many Queues as it likes. However, at least one of these Queues will need to be an Incoming Request Queue. An Incoming Request Queue is a Queue through which the Framework presents Requests to the driver. The most common way for a driver to specify an Incoming Request Queue is by marking a Queue it creates as the Default Queue for a device. A driver can create exactly one default Queue for a given device. In addition (or, in fact, even in place of) a default Queue, a driver may choose to create one or more additional non-default Queues and indicate to the Framework that these Queues are Incoming Request Queues for specific Request types. A driver does this by configuring Request dispatching, indicating the Queue and the particular request type (such as read, write, or device control) that will be routed to the Queue from the Framework.

(Continued on page 19)

WDF Queues...

(Continued from page 18)

Most WDF drivers utilize a single, default, Incoming Request Queue for receiving Requests from the Framework. However, the ability to use multiple WDF Queues to sort and organize Requests is one of the best features of WDF Queues!

Using Multiple Queues

It might not be immediately apparent why using multiple WDF Queues can be handy. Let's look at a few examples to illustrate some common uses for this feature.

As an example, let's consider a driver for a simple device that processes read and write operations, and that also has a set of IOCTL control codes that can be used to enable, disable, and get statistics for the device. Maybe the device is a simple point-to-point communications link. The exact function of the device doesn't matter. What does matter is that it supports the previously specified three types of I/O requests.

Like most devices, our example device can only handle read and write operations when the device is fully powered on (in D0). This won't present any problems for us because WDF Queues are, by default, power managed. That means that if we use a power managed Queue to handle incoming read and write operations, those Requests will only be presented to the driver when the device is in its fully powered state.

However, note the IOCTLs that that driver must support. These control codes, to enable and disable the device and to gather statistics from the device, can be handled by the driver without regard to the device's power state. If we configure our driver to send all arriving I/O requests to a single Queue that is Power Managed, the IOCTLs won't be delivered when the device is in a lower power state. And, if we choose to have all I/O requests delivered to a single queue that is *not* Power Managed, the Framework will deliver us read and write requests to the driver when the device is in low power state and they can't be handled!

What do we do? Well, to support this device, a WDF driver can choose to configure *two* Incoming Request Queues: One default Power Managed Queue that handles the read and write requests, and another Queue that is **not** power managed, that that driver configures to handle only Device Control requests. In this way, read and write operations are handled by the Framework according to the power management state of the device, and IOCTLs are delivered to the driver without regard to the device's power state. Problem Solved!

How about another example: Did you ever have to write a driver for a device that can handle one read plus one write simultaneously, but not two reads or two writes? This is a pretty common requirement, and if you've been writing drivers for a while you've probably encountered a device like this. This requirement is easily handled by configuring two Incoming Request Queues, each with the Sequential Dispatching Type. You configure one Queue to handle incoming read requests, and the other to handle incoming write requests.

Extending that last example a bit further, maybe you want to modify this driver to support device control operations. Adding this support is simple! Just configure another Incoming Request Queue and tell the Framework to route any received device control Requests to that Queue. Note that when you configure that Queue, you can easily choose whether the queue is Power Managed, and you can also choose how many device control Requests your driver will have in progress at a time by specifying the Dispatch Type. And you can do this easily, without disturbing the conditions under which your driver already handles read and write operations.

(Continued on page 20)

WDF Queues...

(Continued from page 19)

How About Those Manual Queues

So far, we've primarily focused on Queues that use the Sequential and Parallel Dispatch Types. How do Queues with the Manual Dispatch Type fit in?

As mentioned previously, you're allowed to create as many Queues as you want, and not all of them need be Incoming Requests Queues. You might, for example, create a Queue with the Manual Dispatch Type that your driver uses to hold Requests that are waiting for some event to occur. In this case, your driver would *forward* Requests presented through one of your Incoming Request Queues to your manual Queue.

Drivers often need the ability to "park" Requests within the driver to be completed only when some asynchronous event occurs. For example, in the OSR USB FX2 device there is a switch pack that generates an interrupt with the state of the switches when they are toggled. Instead of having the application poll the driver to determine if something has changed, it would be nice to let the application send asynchronous IOCTLs that get completed when the switch pack changes.

This is a perfect fit for a manual Queue. Requests to read the switch pack arrive at one of the FX2 driver's Incoming Request Queues and are then promptly forwarded to a manual Queue. When the device interrupts, the driver simply drains its manual Queue and completes each of the Requests it retrieves from that Queue with the state of the switch pack. Clean and simple, with the added benefit of cancellation of the Requests being completely handled for you!

Queue Gotchas

Of course, there are going to be some gotchas that you might run into sooner or later. In order to give your later the benefit of our sooner, here are some things to note when writing your driver:

- If a Request is presented to your driver from a Sequential Queue and you forward that Request to a secondary Queue, *another request may be presented to your driver from the Queue*. Thus, if your device only supports one I/O Request at a time, parking the in progress Request on a manual Queue is not an option.

- Don't use the dispatch type as a cheap means of serialization. While it might at first seem tempting to set your Incoming Request Queues to sequential and never worry about locking, that's not really the spirit of the sequential Queue. Sequential Queues should only be used when your device can only support one operation at a time, WDF's Synchronization Scope (not covered in this article) should be used in all other cases.

- You cannot complete Requests while they are on a Queue.

Queue Power!

So, that's a brief introduction to WDF Queues. We hope you agree that WDF Queues are one of the most powerful, interesting, and useful features of WDF.

Being Resourceful...

(Continued from page 7)

So, given that, let's talk about what *is* in a VERSIONINFO resource.

VI resource

The VI resource that we mention above is defined in the RC file using the VERSIONINFO statement. The VI resource itself can be defined using one of two following formats:

```
versionID VERSIONINFO fixed-info { block-statements …}
```

Or,

```
versionID VERSIONINFO
fixed-info
BEGIN
block-statement
….
END
```

Let's describe the fields used in each of these formats.

versionID

This field is the version resource identifier and must be 1. As an alternative to specifying 1 in this field, devs often specify **VS_VERSION_INFO**. The value for VS_VERSION_INFO is defined in the include file "verrsrc.h", but is referenced in the include file "winver.h" (typically referenced in an RC file).

fixed-info

The fixed-info section of a VI resource contains version information such as the file version and intended operating system. It is composed of the following statements:

- **FILEVERSION** *version* – This statement defines the binary version number of the executable. It is made up of two 32-bit integers which are broken up into four 16-bit fields. Given a the statement "FILEVERSION 2,5,6,7" this would produce the integers 0x00020005 and 0x00060007 or the 4 16-bit fields of 0x0002, 0x0005, 0x0006, 0x0007 respectively.

- **PRODUCTVERSION** *version-* This statement defines the binary version number of a product associated with the executable. Its format follows the same format as defined in the **FILEVERSION** statement. One thing that devs often find confusing is that this version number does not need to match the version number in the **DriverVer** field of the driver's INF file. In fact, in most cases, these two values are unrelated. Crazy, I know… but that's the way it is.
- **FILEFLAGSMASK** *fileflagsmask* – This statement is used to give some information about the state of the executable image associated with this VI. The state of the image is defined by OR'ing together one or more of the following flags (example: "FILEFLAGS VS_FF_DEBUG | VS_FF_PRERELEASE"):

  - VS_FF_DEBUG – the image contains debugging information or is compiled with debugging features enabled.
  - VS_FF_PATCHED – This image has been patched and is not identical to the original image containing the same **FILEVERSION** number.
  - VS_FF_PRERELEASE – This image is a development version of the image and is not a released product
  - VS_FF_PRIVATEBUILD – This image was not built using the standard release procedures. If this flag is set the **StringFileInfo** block must contain a **PrivateBuild** string.
  - VS_FF_SPECIALBUILD – This image was built using the standard release procedures but varies from the original image with the same **FILEVERSION** number. If this flag is set the **StringFileInfo** block must contain a **SpecialBuild** string.
  - VS_FF_FILEFLAGSMASK – this flag is a combination of all preceding values.

- **FILEOS** *fileos* – This statement is used to define the operating system that this image is valid for. This field is a 32-bit integer that is made up of two 16-bit fields. The high 16 bits indicate the OS that the image is valid for, while the low 16-bits indicates' the windowing system. Most driver writers only care about Windows

(Continued on page 22)

OSR: Just Ask

Ask us to cogently explain the Windows I/O Manager to a couple dozen Windows developers of varied background and experience. Ask us how to address latency issues in a given design of a driver. Ask us to look at a post-mortem system crash, determine its root cause, and suggest a fix. Ask us to design and implement a solution that plays well with Windows, even if it has no business being a Windows solution in the first place.

Ask us to perform any of the above activities for your company, and you will be pleased with the definitive answer or result we provide.

So, the only question WE have is, "How can we help you?" Contact: sales@osr.com

Being Resourceful...

(Continued from page 21)

NT, so the only value that makes sense is (If you care to know what the other values are check out "verrsrc.h"):

- VOS_NT – This value indicates that this image was designed for 32-bit windows. The reader should note that there is no value for 64-bit windows, setting VOS_NT will suffice

- **FILETYPE** *filetype* –This statement is used to define the type image this VI resource describes. This field can contain one of a number of values, but as stated above we are driver writers and only care about drivers. Thus for us, the only value that make sense is:

  - VFT_DRV – This value indicates that the image contains a device driver. The **FILESUBTYPE** value described below contains a more specific description of the contained driver.

- **FILESUBTYPE** *filesubtype* –This statement is used to further refine the driver image that was indicated in the **FILETYPE** state. It can have one of the following values:

  - VFT2_UNKNOWN – this value indicates the driver type is unknown
  - VFT2_DRV_COMM – image contains a communications driver
  - VFT2_DRV_PRINTER – image contains a printer driver
  - VFT2_DRV_KEYBOARD – image contains a keyboard driver
  - VFT2_DRV_LANGUAGE – image contains a language driver
  - VFT2_DRV_DISPLAY – image contains a display driver
  - VFT2_DRV_MOUSE – image contains a mouse driver
  - VFT2_DRV_NETWORK – image contains a network driver
  - VFT2_DRV_SYSTEM – image contains a system driver
  - VFT2_DRV_INSTALLABLE – image contains an installable driver
  - VFT2_DRV_SOUND – image contains a sound driver
  - VFT2_DRV_VERSIONED_PRINTER – image contains a versioned printer driver

The reader should note that any field listed above that is not specified in your RC file, will default to 0. In addition, if you reexamine *Figure 1* you will notice that none of the information from the *fixed-info* section is visible via Explorer. You should keep in mind that the information contained in the VI is really only there for informational purposes, the *real* information about the image is contained within the Portable Executable header (PE Header) of the image itself.

block-statement

The "block-statement" section defines one or more version-information blocks. The version-information blocks can contain multiple **StringFileInfo** blocks and accompanying **VarFileInfo** blocks as defined below.

VarFileInfo Block

VarFileInfo defines a Variable File information block. This block describes the language and character set used to encode the strings contained within the associated **StringFileInfo** block discussed in the next section. The **VarFileInfo** block is defined as follows:

```
BLOCK "VarFileInfo" {VALUE "Translation", langID,
charsetID ….}
```

Where:

- langId – This field contains a language code in hexadecimal format. All the possible values for this field are contained within the documentation, but an example would be a value like "0x0409" which indicates a language of U.S English.
- charsetID - This field contains a character set code in hexadecimal format. All the possible values for this field are contained within the documentation, but an example would be a value like "1200" which indicates a character set of Unicode.

With this "Translation" statement you can specify multiple langId and charsetID pairs which will show the user that that are multiple translations for the VI in the image. So as an example let's assume that in our RC file we wanted to have a **StringFileInfo** information block in U.S. English using the Unicode character set. Our **VarFileInfo** block would be defined as in *Figure 4* below:

```
BLOCK "VarFileInfo"
BEGIN
VALUE "Translation", 0x409, 1200
END
END
```

Figure 4—VarFileInfo for U.S. English with Unicode

So now that we've indicated the language and character set to be used for our information, we need to create a **StringFileInfo** block in U.S. English using Unicode with the appropriate information. Let us see how it is done in the next section.

(Continued on page 23)

Being Resourceful...

(Continued from page 22)

StringFileInfo Block

StringFileInfo defines an information block that contains a number of string-name parameters that describe the contained image in human-readable format. This block describes the language and character set used to encode the strings contained within the block. The **StringFileInfo** block is defined as follows:

```
BLOCK “StringFileInfo” {BLOCK “lang-charset” {VALUE
“string-name”, “value” …}}
```

Where:

- lang-charset – this field contains a language and character-set pair. All the possible values for this field are contained within the documentation, but an example would be a string value like “040904B0” which indicates a language of U.S English (0409) and a character set of Unicode (04B0).
- string-name –this field contains one or more predefined names that describe the image in human readable format (in the language and character set defined by the lang-charset statement). The predefined names are:
  - Comments – a optional comment string
  - CompanyName – a required string indicating the company that created this image
  - FileDescription – a required string that contains a string describing the image
  - FileVersion – a required string that contains the version number of the file in human-readable format, for example “V1.0.3.2” or “A1.0.0.0 RC2”
  - InternalName – a required string that contains the name of the image, for example “osrdrv.sys”
  - LegalCopyright – an optional string that contains the copyright notices that apply to this image. If used, this should contain all copyright notices, legal symbols, copyright dates, and other information that apply.
  - LegalTrademarks – an optional string that contains all trademarks and registered trademarks that apply to the image. If used, this should contain all notices, legal symbols, trademark numbers, and other information that apply,
  - OriginalFileName – a required string that contains the original name of the file. This allows an application to determine whether the image has been renamed by the user.
  - PrivateBuild – this string should only be present if the user specified VS_FF_PRIVATEBUILD in the FILEFLAGMASK defined in the fixed-info section. It should describe something about the private build.
  - ProductName – This is a required string that describes the name of the product which this image is associated with.
  - ProductVersion – This is a required string that contains the version number of the product which this image is associated with. For example, “OSRUSB Learning Kit V2.0.1.0”
  - SpecialBuild – this string should only be present if the user specified VS_FF_SPECIALBUILD in the FILEFLAGMASK defined in the fixed-info section. It should describe something about the special build.

It is worth noting that you can add your own string-name fields to this block. Unfortunately you will not be able to see them via Explorer. To see them you would have to write a program that uses the Win32 Version Information functions to retrieve them. In addition if you don’t specify a pre-defined name, its information will not be shown via Explorer.

So those are the fields that are contained within a **StringFileInfo** block. Now, as we showed in *Figure 4* we defined a **VarFileInfo** block that indicated that our **StringFileInfo** block is in U.S. English using the Unicode character set. Therefore we have to have a **StringFileInfo** block with a *lang-charset* that corresponds to what we specified in the **VarFileInfo** block. Since we put 0x409 (U.S. English) and 1200 (Unicode) then the setting for our *lang-charset* in our **StringFileInfo** block will be “040904b0”, where 0409 corresponds to our specified language and 04b0 corresponds to our specified character set (1200 == 0x04b0). Simple! *Figure 5* below shows the **StringFileInfo** block for U.S English in Unicode.

```
BEGIN
BLOCK "StringFileInfo"
BEGIN
BLOCK "040904b0"
BEGIN
VALUE "Comments",          "OSR Driver"
VALUE "CompanyName",       "OSR Open Systems
Resources, Inc."

VALUE "FileDescription",   "OSR Driver"
VALUE "FileVersion",       “V1.0.0.0”
VALUE "InternalName",      "A OSR Written Driver"
VALUE "LegalCopyright",    "2011 OSR Open Systems
Resources, Inc."

VALUE "OriginalFilename", "OSRDRV.sys"
VALUE "ProductName",       "OSR Driver"
VALUE "ProductVersion",    “V1.0.0.0”
END
END
```

Figure 5—StringFileInfo for U.S. English in Unicode

So as you can see, the VERSIONINFO block can be used to thoroughly describe the image that is embedded in and this allows the user and the developer to quickly identify the software installed on the users’ machine.

(Continued on page 24)

Being Resourceful...

(Continued from page 23)

Note: RC files and VERSIONINFO resources can be internationalized. All you need to do is create additional VERSIONINFO resources in your RC file in the languages you require.

Now that we've discussed what fields are in a **VERSIONINFO** resource, let's talk about creating one for our image.

Creating your own VI Resource

As mentioned earlier, an RC file is just a text script file. While it can be created with a resource editor, for example within Visual Studio, it can just as well be created by hand using your favorite text editor. Instead of starting from scratch and writing your RC file from scratch, the easiest thing to do is find an existing RC file with the required information and replace the fields that are important to your driver. We would suggest you copy the information shown in *Figure 2* into your WDK project's RC file and edit it appropriately.

Now when we say edit it appropriately, what we mean is that you modify the fields listed in *Figure 6* with settings that are appropriate for your image (you should note that if you want your Version Information to be visible in multiple languages you are going to be required to have a StringFileInfo/ VarFileInfo section for each supported language).

Once you have completed that you need to add the name of the ".RC" file that you created to your sources file. So in an example "Sources" file, for OSR's "nothing" driver shown in *Figure 7,* you can see how we added "nothingver.rc" to the "SOURCES" line.

- FILEVERSION
- PRODUCTVERSION
- FILEFLAGSMASK
- FILETYPE
- FILESUBTYPE
- CompanyName
- FileDescription
- InternalName
- LegalCopyRight
- LegalTradeMarks
- OriginalFileName
- ProductName
- ProductVersion

Figure 6—Modify These Fields as Appropriate

```
MAJORCOMP=ntos
MINORCOMP=osr

TARGETNAME=nothing
TARGETPATH=obj
TARGETTYPE=DRIVER

INCLUDES=$(DDK_INC_PATH)\inc;..\inc

SOURCES=nothing.c nothingver.rc
```

Figure 7—Sample Sources File with .RC File

When we use the WDK "build" utility to compile our nothing driver, it will see the "nothingver.rc" file on the "SOURCES" line in our "SOURCES" file and automatically compile it and link it to the "nothing.sys" file that it builds.

After you have successfully built the driver, make sure that you use Explorer to examine the information that you specified to ensure that all the information you want the user to see is there. In addition you want to make sure that there is no information present that you did not intend to be there. Remember that sometimes (depending on how the VI got created) Microsoft supplied information can sneak in, so you have to make sure that you're supplying all the correct fields.

Summary

With the information contained within this article you can now create and add a comprehensive RC file to your driver without the risk of getting unexpected values in the various fields. This information will allow your users and support people to immediately know what version of your product and image is being run. Now that is being resourceful!

Windows Internals & Software Driver Development

Attention security researchers, government contractors and engineers involved in security and threat analysis modeling! The next offering of our Windows Internals & Software Drivers seminar has been scheduled.

17-21 October, Waltham, MA

For a look at the outline, pricing and registration information, visit www.osr.com/swdrivers.html.

Win7 Crash Redux...

(Continued from page 9)

The current block starts at:

```
fltmgr!TreeUnlinkMulti+0x4e:
fffff880`0106f13e 488bdf          mov     rbx,rdi
```

The mistake in the earlier analysis was to miss the jump backwards several instructions afterwards:

```
fffff880`0106f158 ebe7            jmp     fltmgr!
TreeUnlinkMulti+0x51 (fffff880`0106f141)
```

Thus, we really do have a small block of code under analysis, as shown in *Figure 4* below.

The reader that pointed out the loop here also pointed out the intent of this code fragment:

> *The code's overall intention is to walk a given tree, remove the nodes that match a given pair of keys, and return these nodes as a list (linked through the RightChild members only).*
>
> *The analyst has identified the given tree and has in Figure 7 dumped for us the root node, as the TreeLink member of a _NAME_CACHE_NODE. See there that the LeftChild member is corrupt but not with the value of RSI at the time of the fault. Execution will have worked some distance into the RightChild subtree until reaching a node that has the faulting RSI as either its LeftChild or RightChild member. Most plausibly, this tree was already corrupt when TreeUnlinkMulti was entered. A race condition, whether inside TreeUnlinkMulti or out, is just one of many ways that links in a tree might get corrupted.*

Thus, at this point we're pretty much at a similar conclusion of the analysis: we have a data corruption; it doesn't seem likely the corruption occurred *here* but it is clear there is a data corruption.

As noted previously, we've seen similar data corruption – on a different computer system, but on Windows 7 x64. In the first dump, we observed what appears to be a single bit error in memory. By itself it led us to suspect the machine. Seeing this on a *different* computer in similar circumstances makes us suspect there is some source of data corruption in the code. While a race condition is a potential data corruption source, it's not the *only* possibility.

Data corruption issues are often the most difficult to track down. Frequently the source of the corruption shows up from a pattern that materializes after reviewing a number of crash dumps, not a single crash dump. While we still do not know the actual issue here, we'll be on the look-out for it in the future and invite our readers to share their own observations if they see it as well.

```
fltmgr!TreeUnlinkMulti+0x4e:
fffff880`0106f13e 488bdf          mov     rbx,rdi

fltmgr!TreeUnlinkMulti+0x51:
fffff880`0106f141 488b4620        mov     rax,qword ptr [rsi+20h]
fffff880`0106f145 483bd0          cmp     rdx,rax
fffff880`0106f148 741b            je      fltmgr!TreeUnlinkMulti+0x75 (fffff880`0106f165)

fltmgr!TreeUnlinkMulti+0x5a:
fffff880`0106f14a 483bd0          cmp     rdx,rax
fffff880`0106f14d 720b            jb      fltmgr!TreeUnlinkMulti+0x6a (fffff880`0106f15a)

fltmgr!TreeUnlinkMulti+0x5f:
fffff880`0106f14f 488b7610        mov     rsi,qword ptr [rsi+10h]
fffff880`0106f153 4885f6          test    rsi,rsi
fffff880`0106f156 74ce            je      fltmgr!TreeUnlinkMulti+0x36 (fffff880`0106f126)

fltmgr!TreeUnlinkMulti+0x68:
fffff880`0106f158 ebe7            jmp     fltmgr!TreeUnlinkMulti+0x51 (fffff880`0106f141)
```

Figure 4

Analyst's Perspective

10 WinDBG Commands You Might Not Know (But Should)

Can you count the number of WinDBG commands you know on one hand? Been meaning to learn some commands *other* than **!analyze –v** but been too busy to crack the docs open? Well then, this article is for you! I'm going to break down ten WinDBG commands that I couldn't live without.

System Information Commands

Sometimes as part of your analysis, you'd like a bit more detailed information about the target system that generated the crash dump. The commands in this section are going to let you find out critical details about your system that just might be the clues you need to perform your analysis.

!vm

Don't be fooled by the name, the **!vm** command gives you a great quick view into the virtual *and* physical memory usage on a system. When I run **!vm** I like to use a flags value of 0x21, which will omit some process specific memory usage information and add in some extra info about the kernel address space on platforms that support it (See *Figure 1*).

NOTE: The !vm output currently has a bug where the non-paged pool usage will always be listed as zero. The actual non-paged pool usage is listed as, "NonPagedPoolNx Usage" in the output.

Note here that we see the amount of physical memory in the system as well as how much memory is currently free. We then get to note the current usage of the system PTEs as well as the pools. If we suspect some sort of resource exhaustion going on in the system, we can use this command to quickly pinpoint which resource is being consumed.

```
kd> !vm 0x21

*** Virtual Memory Usage ***
        Physical Memory:      261886 (   1047544 Kb)
        Page File: \??\C:\pagefile.sys
          Current:   1572864 Kb  Free Space:   1571132 Kb
          Minimum:   1572864 Kb  Maximum:      3145728 Kb
        Available Pages:      211575 (    846300 Kb)
...
        Free System PTEs:     231247 (    924988 Kb)
...
        NonPagedPool Usage:        0 (         0 Kb)
        NonPagedPoolNx Usage:   2969 (     11876 Kb)
        NonPagedPool Max:      52691 (    210764 Kb)
...
        PagedPool Usage:        4904 (     19616 Kb)
        PagedPool Maximum:     51200 (    204800 Kb)
…
```

Figure 1

!sysinfo

Do you have a customer that can repeatedly reproduce a problem but you just can't reproduce it with the exact same procedure? Maybe you're not using a fast enough processor or the right BIOS version, but in any event, how can you tell what system configuration the customer is using from just a dump file? Enter **!sysinfo**, a command that can tell you just about anything you'd want to know about your system using information cached on the target. For example, let's see what kind of processor is in this system (See *Figure 2*, page 27).

There's more here as well if you go exploring the documentation for the command. For example, you can even query information about which RAM slots are populated using the **smbios** switch (e.g. **!sysinfo smbios –memory**).

Suspected Race Condition Commands

Race conditions are the worst. They're difficult to track, difficult to reproduce, and when you get a crash it may be too late. The race has already happened and when the system crashes you're dealing with the secondary failure, so there's nothing that can be done, right? Wrong! WinDBG has a couple of commands that can make you feel like you've won the lottery and pinpoint the racing thread with ease.

!running

If you're lucky, the thread that is racing with your crashing thread is still running on another processor. This is where **!running** comes in, which will show you information about each thread that is currently running on a processor in the system. Whenever I run this command I like to specify the **–ti** switch, to include **t**hread stacks in the output as well as **i**dle threads:

```
1: kd> !running -ti

..
  0    f7857120
85ed2da8          ................

ChildEBP RetAddr
ba9be270 804f961f nt!KeBugCheckEx+0x19
ba9be62c 805310dd nt!KiDispatchException+0x307
ba9be694 8053108e nt!CommonDispatchException+0x4d
ba9be6a4 f6cd768d nt!Kei386EoiHelper+0x18e
ba9be6b4 f6c0675a ks!
KsReleaseIrpOnCancelableQueue+0x5b
ba9be758 f6c15264 portcls!
CIrpStream::ReleaseUnmappingIrp+0xd0
ba9be780 f6c21760 portcls!UpdateActivePinCount+0xb
f6cd7553 10c2c95e portcls!
CPortPinWavePci::DistributeDeviceState+0x4d

  1    f7867120  86fb5b30          ..............
```

(Continued on page 27)

Analyst's Perspective...

(Continued from page 26)

```
ChildEBP RetAddr
f7a1eba0 f6c0d445 portcls!
CIrpStream::GetMapping+0x17
f7a1ebc8 f6c31ce1 portcls!
CPortPinWavePci::GetMapping+0x2a
…
```

!ready

If the thread isn't actively running, you might think that you would have to go the long way and try finding a racing thread with **!process 0 7**. However, WinDBG also provides us a way to look at threads that are *ready* to run, with the **!ready** command. Maybe the current thread pre-empted another thread and that's the reason for the race, in which case the other thread will be in the ready state. Whenever using **!ready**, I like to use the 0xF flags value so that I can see the call stacks of the threads, though I won't do that here just to keep the output short (see *Figure 3*).

Memory Analysis

Have an address and want to know what it is? Is it a pool allocation? Is it paged out? Here are a couple of commands that will get you the information that you need.

!pool

!pool is a standard command for any toolbox, so I suspect that most of you know it and love it already. However, for those that might not be aware, **!pool** will take an arbitrary virtual address and let you know if it is a pool allocation or not. If it is indeed a pool allocation, you'll be told some details about it, such as whether it's allocated or freed, the length of the allocation, the tag, etc. When I use **!pool**, I like to specify a flags value of 2 to suppress information about other allocations surrounding the address (See *Figure 4*).

Before moving on, I'd like to note something in the output here that often confuses people. The **previous size** value mentioned here is *not* the, "previous size of this allocation." Instead, what it is telling you is the size of the allocation preceding this entry in the pool page. This is used as part of a consistency check by the Memory Manager to validate that

(Continued on page 28)

```
kd> !sysinfo cpuinfo
[CPU Information]
~MHz = REG_DWORD 1779
Component Information = REG_BINARY 0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0
Configuration Data = REG_FULL_RESOURCE_DESCRIPTOR ff,ff,ff,ff,ff,ff,ff,ff,0,0,0,0,0,0,0,0
Identifier = REG_SZ x86 Family 15 Model 1 Stepping 2
ProcessorNameString = REG_SZ               Intel(R) Pentium(R) 4 CPU 1.80GHz
Update Signature = REG_BINARY 0,0,0,0,2d,0,0,0
Update Status = REG_DWORD 0
VendorIdentifier = REG_SZ GenuineIntel
MSR8B = REG_QWORD 2d00000000
CPUID1 = REG_BINARY 12,f,0,0,8,8,1,0,0,0,0,0,ff,fb,eb,3f

How about the BIOS version and other platform info?

kd> !sysinfo machineid
Machine ID Information [From Smbios 2.3, DMIVersion 35, Size=2982]
BiosVendor = Dell Computer Corporation
BiosVersion = A05
BiosReleaseDate = 10/05/2001
SystemManufacturer = Dell Computer Corporation
SystemProductName = OptiPlex GX400
BaseBoardManufacturer = Dell Computer Corporation
BaseBoardProduct = OptiPlex GX400
BaseBoardVersion =
```

Figure 2

```
kd> !ready
Processor 0: Ready Threads at priority 8
    THREAD 8543cd48  Cid 0004.0b58  Teb: 00000000 Win32Thread: 00000000 READY
Processor 0: Ready Threads at priority 1
    THREAD 85367020  Cid 0004.0008  Teb: 00000000 Win32Thread: 00000000 READY
```

Figure 3

```
kd> !pool 8539da40 2
Pool page 8539da40 region is Nonpaged pool
*8539da40 size:    8 previous size:  148  (Free)      Io
                Pooltag Io    : general IO allocations, Binary : nt!io
```

Figure 4

Analyst's Perspective...

(Continued from page 27)

the page of memory has not been corrupted by buffer overruns or underruns.

!pte

Sometimes you'd like to view the virtual memory structures for a given virtual address, such as the PDE and PTE. In that case, you can use the **!pte** command, which will provide decoded information about a virtual address. Here's some example output for a valid virtual address:

```
kd> !pte 9371a000
                        VA 9371a000
PDE at C0300934           PTE at C024DC68
contains 9B441863         contains 8B660121
pfn 9b441 ---DA--KWEV     pfn 8b660 -G--A--KREV
```

We can also see what happens if we specify a virtual address that isn't valid to the hardware, such as one with its backing page currently in transition:

```
kd> !pte 93726000
                        VA 93726000
PDE at C0300934           PTE at C024DC98
contains 9B441863         contains 8B5A0860
pfn 9b441 ---DA--KWEV     not valid
                           Transition: 8b5a0
                           Protect: 3 - ExecuteRead
```

Now we have some further details as to *why* the address is invalid, which may be invaluable to our investigation.

Viewing O/S Trace Information

The O/S has some built in trace facilities that you can turn on to collect data that might be useful during analysis. Unfortunately these facilities need to be turned on *before* the problem happens, but knowing that this information is available can be useful in some situations.

!verifier

We're *all* using Driver Verifier, right? Well, what you might not realize is that starting in Windows Vista Verifier has been enhanced to keep a log of interesting events that happen in your driver. Assuming that you've enabled Driver Verifier on your driver, you can now extract valuable information with the following **!verifier** commands:

- **!verifier 0x80** *Address* – This command dumps the allocate and free log, which logs each pool allocate and free made by your driver. Included in the output is the call stack of the operation, which can be invaluable when you're trying to track down use after free or double free bugs. Optionally, the command takes an address value that will limit the output to only include allocation ranges including that address.

- **!verifier 0x100** *Address* – This command dumps the IRP log, which logs each call to **IoAllocateIrp**, **IoCancelIrp**, and **IoCompleteRequest** made by your driver.

- **!verifier 0x200** – This command dumps the critical region log, which logs each call to **KeEnterCriticalRegion** and **KeLeaveCriticalRegion** made by your driver.

!htrace and !obtrace

Handle leaks and object reference leaks can be *very* tricky to track down, especially when working with a large code base. Luckily, the O/S has built in facilities for logging handle and reference count activities. All you need to do is enable them and be aware of the commands available for extracting the logs, which in this case are **!htrace** and **!obtrace**.

Handle tracing needs to be enabled on a per-process basis, which can be done by using Application Verifier. As driver writers, however, we're typically only interested in kernel handles. By implementation, kernel handles are actually just handles from the handle table of the System process. And, as luck would have it, if you enable Driver Verifier handle tracing is automatically turned on for the System process. Thus, as long as Driver Verifier is enabled on the target you can dump the handle tracing log for all kernel handles with **!htrace 0 *PEPROCESS***:

```
1: kd> !htrace 0 85e0a170
Process 0x847c6530
ObjectTable 0x85c01aa8

--------------------------------------
Handle 0x281C - CLOSE
Thread ID = 0x00000ab4, Process ID = 0x00000408

0x82a63f72: nt!ObpCloseHandle+0x7F
0x82a98bf0: nt!ObCloseHandle+0x40
0x828cef7b: nt!ExpWorkerFactoryCreateThread+0xFC
0x828bf02e: nt!NtSetInformationWorkerFactory+0x56D
...
--------------------------------------
Handle 0x281C - OPEN
Thread ID = 0x00000ab4, Process ID = 0x00000408

0x82a97fde: nt!ObOpenObjectByPointerWithTag+0xC1
0x82a98043: nt!ObOpenObjectByPointer+0x24
0x82a9cdf0: nt!PspCreateObjectHandle+0x2E
0x82a742ff: nt!PspInsertThread+0x685
0x82a9392e: nt!PspCreateThread+0x244
```

Object reference tracing, on the other hand, needs to be enabled on a system wide basis with GFlags. Due to the volume of tracing generated, when you enable tracing you must specify the pool tag of the object you want to trace (e.g. 'File') and you can also limit the tracing to only apply to a single process' objects. Once you have enabled tracing via GFlags, you can view the trace for a given object with **!obtrace** (shown in *Figure 5*, page 29).

(Continued on page 29)

Analyst's Perspective...

(Continued from page 28)

Plug and Play and Power Issues

Nothing is more annoying than when the system hangs during a plug and play or power operation. Luckily, the debugger provides a quick way to identify the threads participating in the operation so that you can get right to resolving the issue.

!pnptriage

!pnptriage is a nifty command that combines the output of several PnP related debugging commands. It will identify any of your devnodes with problems as well as dump out any PnP worker threads that are currently executing, which will give you the ability to quickly identify the threads in the system that might be of interest to you:

```
0: kd> !pnptriage

…

*****************************************************
Dumping devnodes with problems...
*****************************************************
Dumping IopRootDeviceNode (= 0x86c05c08)
DevNode 0x8a131e78 for PDO 0x8a1af6a8
  InstancePath is "USB\VID_0403&PID_6001
\7&2363c875&0&1"
  State = DeviceNodeInitialized (0x302)
  Previous State = DeviceNodeUninitialized (0x301)
  Problem = CM_PROB_FAILED_INSTALL
...
*****************************************************
Dumping currently active PnP thread (if any)...
*****************************************************

Dumping device action thread...

THREAD 847f8798  Cid 0004.0044  Teb: 00000000
Win32Thread: 00000000 WAIT: (Executive) KernelMode
Non-Alertable
    8712b944  NotificationEvent
...
nt!KiSwapContext+0x26
nt!KiSwapThread+0x266
nt!KiCommitThreadWait+0x1df
nt!KeWaitForSingleObject+0x393
nothing!NothingAddDevice+0xa9
nt!PpvUtilCallAddDevice+0x45
nt!PnpCallAddDevice+0xb9
nt!PipCallDriverAddDevice+0x565
nt!PipProcessDevNodeTree+0x15d
nt!PiRestartDevice+0x8a
nt!PnpDeviceActionWorker+0x1fb
nt!ExpWorkerThread+0x10d
nt!PspSystemThreadStartup+0x9e
nt!KiThreadStartup+0x19
```

(Continued on page 30)

```
0: kd> !obtrace 9f6aca50
Object: 9f6aca50
 Image: notepad.exe
Sequence   (+/-)   Tag   Stack
--------   -----   ----  --------------------------------------------------
      f3    +1     Dflt      nt!ObCreateObject+1c4
                             nt!IopAllocRealFileObject+50
                             nt!IopParseDevice+ac4
                             nt!ObpLookupObjectName+4fa
                             nt!ObOpenObjectByName+159
                             nt!IopCreateFile+673
                             nt!NtOpenFile+2a
                             nt!KiFastCallEntry+12a

      f4    +1     Dflt      nt!ObfReferenceObjectWithTag+27
                             nt!ObfReferenceObject+12
                             nt!IopParseDevice+1395
                             nt!ObpLookupObjectName+4fa
                             nt!ObOpenObjectByName+159
                             nt!IopCreateFile+673
                             nt!NtOpenFile+2a
                             nt!KiFastCallEntry+12a

      f5    -1     Dflt      nt!ObfDereferenceObjectWithTag+22
                             nt!NtOpenFile+2a
                             nt!KiFastCallEntry+12a

--------   -----   --------------------------------------------------
References: 2, Dereferences 1

Tag: Dflt References: 2 Dereferences: 1      Over reference by: 1
```

Figure 5

What OSR Students Say

"I learned so much more in the week spent here than trying to learn on my own these past 4 years. I only wish I took the class back then. OSR continues to provide the best training experience we developers could wish for."

Analyst's Perspective...

(Continued from page 29)

!poaction

!poaction is the essential command for debugging any of your power related issues. Most importantly, **!poaction** will show any outstanding query or set power operations and the driver to which they were sent, which can be used to quickly identify which devices are preventing the power operations from occurring. Great for getting insight into what's going on when the system will mysteriously refuse to enter or resume from a lower power state:

```
1: kd> !poaction
PopAction: 8296ea60
  State..........: 3 - Set System State
  Updates........: 0
  Action.........: Sleep
  Lightest State.: Hibernate
  Flags..........: 80000004 OverrideApps|Critical
  Irp minor......: SetPower
  System State...: Hibernate
  Hiber Context..: 89dd5978

Allocated power irps (PopIrpList - 82978480)
  IRP: 8e1d8f00 (set/D0,), PDO: 89c0a248, CURRENT:
89fde028
  IRP: 9d722e48 (set/D0,), PDO: 89c08818, CURRENT:
89f92620
  IRP: 9fe7ee70 (set/D0,), PDO: 89c08940, CURRENT:
89f917a0
...
```

Did I Miss Any?
Got your own favorite command that wasn't represented here? Send me an email at ap@osr.com and let me know!

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 seminar. Comments or suggestions for this or future Analyst's Perspective columns can be addressed to ap@osr.com.

Peter Pontificates...

(Continued from page 5)

own all the driver code that is produced. Almost nobody writes driver code in a clean room, starting with an empty buffer in the editor.

It is a sad commentary on the state of our society that we feel the need to attach the following to this article:

IMPORANT NOTE: Neither OSR, *The NT Insider*, nor Peter provide any legal advice. Do not rely on the contents of this publication, and certainly do not rely on the contents of this column, for definitive legal information or advice regarding any topic, particularly involving intellectual property law. Your mileage may vary. Professional drivers on a closed course. Do not attempt. No purchase necessary, void where prohibited. Silica Gel – Desiccant – Do Not Eat. Caution: contents may be hot. Offer not good after curfew in sectors R or N.

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 see your comments or a rebuttal in a future issue. Send your own comments, rants, or distortions of fact to: PeterPont@osr.com.

Windows File System Development

Whether developing file systems, file system mini-filters OSR's *Developing File Systems for Windows* seminar has proved year after year to be the most effective way to get up to speed.

NEXT OFFERING:

19-22 September, Vancouver, BC

For more information, visit www.osr.com/fsd.html

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

The NT Insider™ is a subscription-based publication

Subscribe to The NT Insider—Digital Edition

If you are 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)	25-29 July	Boston/Waltham, MA
Developing File Systems for Windows	19-22 September	Vancouver, BC Canada
Windows Internals for Forensic Analysts	26-29 September	Columbia, MD
Writing WDF Drivers (Lab)	3-7October	Seattle, WA
Internals and Software Drivers (Lab)	17-21 October	Boston/Waltham, MA
Kernel Debugging & Crash Analysis	14-18 November	Columbia, MD

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