ISSUE VOLUME
01/19
MARCH—APRIL
2013

# The NT Insider

A publication of OSR Open Systems Resources, Inc.

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

http://www.osr.com

**Consulting Partners**
W. Anthony Mason
Peter G. Viscarola

**Executive Editor**
Daniel D. Root

**Contributing Editors**
Scott J. Noone
OSR Associate Staff

**Send Stuff To Us:**
NtInsider@osr.com

*Single Issue Price: $15.00*

## We're Baaaaaack!
## The NT Insider

First and foremost, our apologies for the extended vacation we've taken from *The NT Insider*. We'd like for you to believe we simply fell asleep at our keyboards, but the reality is that we just got busy. Maybe that's the same thing...

Suffice it to say, we have plenty to tell you about, as much has happened in our absence, and it's high time we bring our faithful readership up to speed.

In this issue, we start to smother you with some additional detail on the new, integrated development environment for driver development, in *MSBuild 101.*

We're not afraid to revisit topics from the past, either, and coverage of *The Inverted Call Model in KMDF* is a good example of this, as well as a guest article offering *Another Look at Lint* .

Toss in a topic WE found interesting ourselves, in *Introducing Reader/Writer Spin Locks*, and something esoteric such as *Fixing Kernel Debugger Extensions*, PLUS the Pontifications of our fearless leader, and you've got the March-April 2013 issue of *The NT Insider*—enjoy!

*How are we doing? Are these article topics interesting? What else do you want to hear/know about? Maybe you 've been yearning to write up something yourself that you'd like us to consider publishing in this journal?*

*We DO enjoy hearing from you and appreciate your patronage. Drop us an email at ntinsider@osr.com.*

### 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. We'll quite literally fall asleep at our keyboards to make sure you are happy.

So, the only question WE have is, "How can we help you?"

Contact: sales@osr.com

# Peter Pontificates: Coming to the Surface

About two years ago in my Pontification here in *The NT Insider*, I was musing about some of the things I would do if I were president of the Windows division at Microsoft. I wrote, in part:

*The other thing I'd do is I would start designing and manufacturing Microsoft-branded hardware. Yes, yes, I know all about how important the OEMs (Dell, HP, and the like) are to Microsoft, and how Microsoft needs to be careful about treading on their turf. Those relationships can be managed.*

*What Microsoft sorely needs right now is a BIG win in the tablet space. They cannot settle for releasing a solution that's just OK. They need something that overtakes, and not merely imitates, the iPad. The only way they can control the end-to-end quality and experience, and get a device to market sufficiently quickly is by having total control of the solution.*

Given the above, you can imagine how excited I was about the Surface when it was announced. In fact, on the day the Surface RT was released, I slogged down to the Bellevue Mall after work and actually **stood in line** to get one. I'm not sure if you understand how significant that last statement is. I wouldn't stand in line to go "around the world" with Beyoncé, never mind to buy a freaking computer. But stand in line to be one of the first to get my paws on a Surface RT I did.

It was impressive: the only time I've actually seen a Microsoft store packed with people. Ouch. Sorry.

I got my Surface RT and – after playing with it for about an hour -- came to the same conclusion as just about everyone who bought one. Great hardware, but in general pretty useless. It doesn't run "normal" Windows software (duh… it's an ARM processor). It doesn't come with Outlook. And I'm convinced the guy who invented the email client that the Surface RT **does** have, spends time in his office quietly snickering while people try to actually use the software he invented.

And when it comes to apps? Not only are there very few apps, but the apps that do exist for the Surface RT are super terrible compared to their iPad counterparts. For example, consider the app for reading The Wall Street Journal (Yes… I read The Wall Street Journal. Cut me some slack, OK? I have the responsibility of actually **running** a small consulting company… though Dan would tell you I appear to be doing that mostly in my spare time these days, as I've generally been sticking to writing code, doing code reviews, writing articles, or playing with my dogs… or, you know, anyone's dog that happens to be around, really). Anyhow, the app for reading the Wall Street Journal on the iPad is a tour de force. Fabulous. So good, in fact, that whoever designed and wrote it should get some kind of award and a raise. It has become my preferred way of reading The Journal each morning. But that same app on the Surface bears no visual or functional resemblance to either the print edition of the newspaper or the iPad app. It's terrible. So terrible, in fact, that I find it completely unusable. Sadly, this is not an exception. Rather, it's the norm when it comes to apps for the Surface.

It would be wrong for me to not acknowledge that there are some people who absolutely **love** the Surface RT. I know this because I have actually met one such person. I had the pleasure of sitting next to her on a flight from Boston to Seattle. She was charming, intelligent, and a real pleasure to talk with. And she **loved** the Surface RT. She also happened to be the wife of a very senior and long tenured Microsoft software engineer. She loved the Surface RT so much she bought one for each of her kids in college. Perhaps they love it as much as she does. I just hope they don't try to find the Facebook or Twitter app. But, what college kid would want either of those? After all, they get a swell version of Excel for free. They shouldn't need more than that. Well, that and a copy of Word. And that super email client. So they should be set.

My Surface RT now sits at home, alone and unloved, stacked on an unused desk between a flyer asking for donations to the Obama campaign and a chart that shows the correlation between the colored dots on various Motorola antennas and the frequency range they're designed to support. It's not the sort of pile I access every day. Or, apparently, even every

(CONTINUED ON PAGE 26)

**Yes, I waited in THIS line...**

# If You Build It
## MSBuild 101

For years, the driver development community clamored for integration with Visual Studio. While the application developers lived in the lap of luxury with slick, integrated development and debugging environments, we languished with SOURCES files, command line builds, and a wonky debugger. With the Windows 8 Driver Kit, our cries have finally been answered and driver development has been promoted to first class citizenship by being fully integrated with Visual Studio 2012.

### Death to SOURCES!

Being integrated with Visual Studio means many things, from integrated static analysis, to automated deployment, to debugging. For this article, the change that we're most interested in is the switch from the compilation and linking of drivers being driven by Build, which used SOURCES, DIRS, and MAKEFILEs, to the compilation and linking process being driven by MSBuild and Project Files.

The fact that drivers are now built using MSBuild puts them in good company. This is the current, modern build system provided by Microsoft, thus MSBuild is currently used for all application development, including native, managed, and Windows Store applications. While this may just seem like a burden at first (who wants to learn a new build system?), you'll quickly get to love the amount of community support available over the old system.

Under the covers, Visual Studio projects are really just MSBuild projects with some custom information used to drive the GUI. For example, Visual Studio adds some information to the project file about which debugger to use while debugging the project. When it comes to learning the new build environment, the thing you really want to know about is MSBuild, not Visual Studio. In fact, MSBuild has a command line interface, which means that you could entirely skip the Visual Studio GUI if you were so inclined!

### Anatomy of an MSBuild Project File

At the end of this article, we provide the entire contents of a simple MSBuild project file for a KMDF driver. Our hope is that by the time you reach the example you'll be able to understand it and feel comfortable with its structure.

This driver project contains one C module, nothing.c, and a header file located in a sibling directory of the source. In the interest of clarity, this project file does not work properly from within Visual Studio. Instead, we'll be processing this by directly running MSBuild from within a Visual Studio Tools Command Prompt. This allows us to avoid discussing a lot of things that you can figure out for yourself once you've mastered the basics. MSBuild project files are really just globs of XML. Thus, before we get to the example we'll describe the various elements commonly found in a project file and some of their uses.

#### *Project Element*

All project files start off with a Project element. This indicates the version of MSBuild required as well as the XML schema to use for validation. We're using MSBuild version 4.0 and the standard schema:

```
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/
developer/msbuild/2003">

</Project>
```

Pretty simple so far, right?

#### *Property and PropertyGroup Elements*

Property elements are an important concept in project files. They are basically a name and value pair, where the name of the property is provided by naming the element and the value is the element contents. Due to the fact that the element name is, in fact, the property, all Property elements must be defined within a PropertyGroup element. There is, however, no restriction on the number of PropertyGroup elements you may have, thus (if you wanted to) you could declare each Property in its own PropertyGroup.

Property elements may have custom names that you create as part of your project, or they may have pre-defined names that are used to pass values to the build procedure. The next step in our project file will be to set some pre-defined Property elements to indicate that we are targeting Windows 8 and that we are building a KMDF driver:

```
<PropertyGroup>

  <TargetVersion>Windows8</TargetVersion>

  <PlatformToolset>WindowsKernelModeDriver8.0</PlatformToolset>

  <ConfigurationType>Driver</ConfigurationType>

  <DriverType>KMDF</DriverType>

</PropertyGroup>
```

Property elements can be referenced in the project file using the *$(PropertyName)* syntax. Thus, if we wanted to reference the target version later in the project file we would use *$(TargetVersion)*.

When MSBuild is invoked, a Property element can be specified on the command line via the **/p:** switch.

#### *ItemGroup Element*

Item elements are similar to Property elements, but are much more powerful. Instead of being a one to one concept, where a single name corresponds to a single value, Item elements are one to many, with a single name including multiple values. In addition, Item elements can (and often do) have metadata

(CONTINUED ON PAGE 5)

## MSBuild 101 (Cont.)

(CONTINUED FROM PAGE 4)

associated with them. As with Property elements, Item elements may have custom names that you create as part of your project or they may have pre-defined names used to pass values to the build procedure. Again, like Property elements are always defined in PropertyGroups, Item elements are always defined within an ItemGroup.

Given that Item elements can have multiple values as well as metadata associated with them, the simple element and value pair that we use to define a Property element is not sufficient to describe an Item. Instead, we'll use the Include attribute of the element to specify a semicolon delimited list of values to assign to the Item. It is also possible to have multiple definitions of the Item element with separate Include properties, in which case the subsequent values are appended to the previous values. To create metadata for the Item, we simply add child elements in the form of element and value pairs.

In the following example, we define an ItemGroup containing the *ProjectConfiguration* element. This is another pre-defined element with a pre-defined value for each build configuration that we support.  In this project, we'll only support building Debug Win32 and Release x64 binaries:

```
<ItemGroup>

    <ProjectConfiguration Include="Debug|Win32">
    </ProjectConfiguration>

    <ProjectConfiguration Include="Release|x64">
        <MyMetaDescription>This is an x64 build
        </MyMetaDescription>
    </ProjectConfiguration>

</ItemGroup>
```

In this example, we see two forms of defining an Item element. In the first element, "Debug|Win32", we do not explicitly provide any metadata. However, for the second element, "Release|x64", we provide a custom metadata element *MyMetaDescription* that contains a description of the project configuration.

All Item elements have built in metadata elements, which can be extracted and used as part of expressions. In the final example, we'll also see that the element value can be something more complex than just a simple string.

Item elements can be referenced within the project file in one of two ways. By using the *@(Item)* syntax, you can retrieve a semi-colon delimited list of all of the Item values. To access individual metadata elements, you use the *@(Item->'%(MetadataElement)')* syntax. We'll see examples of each of these in the upcoming section.

### *Target and Task Elements*

Target elements contain one or more Task elements, which use the previously set Property and Item elements to actually do some work. Many Task elements are already provided by MSBuild, Visual Studio, and the WDK and it's possible to create your own if necessary. When MSBuild is invoked, the Target element to execute is chosen on the command line via the **/t:** switch.

Let's add a few Target elements to our project file that invoke the Message Task element, which is a Task provided by MSBuild for printing messages to the console:

```
<Target Name="PrintProperty">
    <Message Text="Driver Target Version is $(TargetVersion)"/>
</Target>

<Target Name="PrintItems">
    <Message Text="@(ProjectConfiguration)"/>
</Target>

<Target Name="PrintItemMetadata">
    <Message Text="@(ProjectConfiguration->'%
        (MyMetaDescription)')"/>
</Target>
```

The following three command lines result in the following outputs:

```
msbuild nothing.vcxproj /t:PrintProperty
```

"Driver Target Version is Windows8"

```
msbuild nothing.vcxproj /t:PrintItems
```

"Debug|Win32;Release|x64"

```
msbuild nothing.vcxproj /t:PrintItemMetadata
```

";This is an x64 build"

(CONTINUED ON PAGE 16)

# Sharing is Caring: Introducing Reader/Writer Spin Locks

I regularly enjoy teasing my file system developer colleagues over the pain they suffer about whether some structure should be stored in paged pool or nonpaged pool or exactly which type of locking primitive they should use for a particular task. "You should switch to writing device drivers," I tell them. "It's much easier. In the land of device drivers, all pool is nonpaged and all locks are spin."

It's plain fact that the life of a device driver writer is easy in some ways. However, I do admit that sometimes, just **sometimes**, I get a bit jealous over the variety of locking choices available to devs who never have to be concerned about running at IRQL DISPATCH_LEVEL or higher. Over the years, I've particularly lusted over ERESOURCES.

ERESOURCES are locks that can be acquired in either shared mode or exclusive mode. When acquired shared, the lock allows multiple threads to simultaneously read from a data area. When acquired exclusive, the lock ensures that there's only one accessor to a data area and thus allows the data to be updated atomically. Because ERESOURCES allow multiple simultaneous readers of a given data area but only a single writer, this category of lock is most often referred to as a reader/writer lock.

*"It should really go without saying but I'm going to say it anyway: Reader/Writer Spin Locks and traditional kernel Spin Locks are different data types and cannot be used interchangeably."*

One problem with ERESOURCES is that they can only be used in code running at less than IRQL DISPATCH_LEVEL. That pretty much leaves them out of contention as a possible solution for most device driver work. And there hasn't been any sort of reader/writer lock documented as being available for use at IRQL DISPATCH_LEVEL. That is, there hasn't been one until the introduction of the WDK for Windows 8.

## Reader/Writer Spin Locks

The WDK for Windows 8 tells us that starting in Vista SP1, Windows includes support for Reader/Writer Spin Locks. These locks are just what their name implies: they are spin locks that can be acquired in either shared mode (for reading, but not modifying, shared data) or exclusive mode (for reading and modifying shared data). Because they're spin locks, they can be acquired at IRQLs less than or equal to DISPATCH_LEVEL.

Reader/Writer Spin Locks are wonderful additions to the driver writer's tool chest. Here at OSR, we've been using them on OS versions as early as Windows 7 without any problem. Using Reader/Writer Spin Locks keeps us from having to choose between (a) acquiring a "regular" Spin Lock when you want to read some data, thereby guaranteeing data integrity but blocking anyone else who might simultaneously want to read that same data, and (b) acquiring no lock at all to read the data, thus risking data inconsistency due to the possibility that the data could be changed while you're in the process of reading it.

Reader/Writer Spin Locks can be useful in all sorts of situations. Consider the case when you have a block of statistics. Prior to the introduction of Reader/Writer Spin Locks you would guard the structure holding those statistics with a traditional Spin Lock. That means that only one thread could access the statistics at a time, regardless of whether that thread wanted to read or update those statistics. With Reader/Writer Spin Locks, however, when you just want to read the statistics you would acquire the lock shared by calling **ExAcquireSpinLockShared**. While you're holding the lock shared, other threads can also acquire the lock shared and simultaneously read the data. When you need to update the statistics, you acquire the lock exclusive by calling **ExAcquireSpinLockExclusive**. This blocks all other acquisitions of the lock, and once the lock is granted allows you exclusive access to the statistics so you can perform your update.

## Acquiring and Releasing Reader/Writer Spin Locks

There are a few interesting things about the implementation of Reader/Writer Spin Locks that are worthy of note. First, notice that the functions are part of the Executive and not the Kernel. So, instead of calling, for example, **KeAcquireSpinLock** as we've been used to, the functions for Reader/Writer Spin Locks are **ExAcquireSpinLockXxxx**. Also, note that the data type for Reader/Writer Spin Locks is an EX_SPIN_LOCK, as opposed to the KSPIN_LOCK used for traditional Spin Locks. Yay! Let's hear it for strong data typing. You must allocate space for the EX_SPIN_LOCK structure in nonpaged memory. Prior to the first use of a Reader/Writer Spin Lock you're required to initialize the EX_SPIN_LOCK by setting it to zero.

Here's an example of the definition of a Reader/Writer Spin Lock in a driver's device context:

```
typedef struct _MY_DEVICE_CONTEXT {

    WDFDEVICE       Device;

    //
    // Statistics area
    //
    ULONG           ReadCount;
    ULONG           WriteCount;
    EX_SPIN_LOCK    StatisticsLock;

    //...

} MY_DEVICE_CONTEXT, *PMY_DEVICE_CONTEXT;
```

(CONTINUED ON PAGE 7)

## Reader/Writer Spin Locks (Cont.)

(CONTINUED FROM PAGE 6)

Note that this satisfies the requirement for the lock being in nonpaged memory because a driver's device context is always stored in nonpaged memory. You would initialize that lock prior to its first use by simply setting its value to zero:

```
    // ...
    devContext = MyGetContextFromDevice(device);
    devContext->StatisticsLock = 0;
```

The function call you use to acquire the Reader/Writer Spin Lock indicates whether you want to acquire that lock in shared or exclusive mode. To acquire a Reader/Writer Spin Lock in shared mode call the following function:

```
KIRQL ExAcquireSpinLockShared(
_Inout_ PEX_SPIN_LOCK SpinLock
);
```

The prototype for the function to acquire a Reader/Writer Spin Lock in exclusive mode is similar:

```
KIRQL ExAcquireSpinLockExclusive(
_Inout_ PEX_SPIN_LOCK SpinLock
);
```

As with traditional Spin Locks, there are **ExAcquireSpinLock SharedAtDpcLevel** and **ExAcquireSpinLockExclusiveAtDpcLevel** functions that you can optionally use if you know in advance that you're code will always be running at IRQL DISPATCH_LEVEL when you attempt to acquire the lock. This would be the case, for example, when you acquire a Reader/Writer Spin Lock from a DPC. These functions provide the small optimization of assuming the current IRQL is already set and not calling **KeRaiseIRQL** to raise the IRQL to DISPATCH_LEVEL before attempting to acquire the lock.

The function you call to release a Reader/Writer Spin Lock depends on the mode in which you acquired the lock. If you acquired the lock shared, you would call **ExReleaseSpinLock Shared**, which has the following prototype:

```
VOID ExReleaseSpinLockShared (
    _Inout_  PEX_SPIN_LOCK SpinLock,
    _In_     KIRQL OldIrql
    );
```

Similarly, to release a lock you've acquired in exclusive mode, you call the following function:

```
VOID ExReleaseSpinLockExclusive (
    _Inout_  PEX_SPIN_LOCK SpinLock,
    _In_     KIRQL OldIrql
    );
```

If you acquired the lock with one of the **ExAcquireSpinLockXxxAt DpcLevel** calls, you must release the lock by calling the matching **ExReleaseSpinLockXxxxFromDpcLevel** function. Note the acquire call is "AtDpcLevel" and the release call is "FromDpcLevel", thus preserving the naming pattern established by the traditional Spin Lock functions.

Finally, there's a function that lets you attempt to "promote" to exclusive mode a Reader/Writer Spin Lock that you're holding in shared mode. The prototype for this function is:

```
BOOLEAN ExTryConvertSharedSpinLockExclusive(
    _Inout_  PEX_SPIN_LOCK  SpinLock
    );
```

This function returns TRUE if the lock was successfully promoted from shared mode to exclusive mode, and FALSE otherwise. The promotion from shared to exclusive will only work if no other threads are holding the lock shared and there are no threads waiting to acquire the lock exclusive. One thing that's easy to overlook whenever you use **ExTryToConvertSharedSpinLock**

(CONTINUED ON PAGE 20)

# Calling User Mode Functions from Kernel Mode
## The Inverted Call Model in KMDF

One of the most common questions we see from students, clients, and new Windows driver writing colleagues is, "How can a driver perform a callback to a user-mode program?" The answer to this is simple: it can't. There is no architected way in Windows for a driver to call a given callback in a user program. It's simply not something you can do.

If it's that simple, should we just end this article here? Probably not. Because when we investigate further, people who ask this question typically don't **actually** want to know how a driver can call a user-mode callback. Rather, what they really want to know is, "How can my driver notify a user-mode program that some event has occurred in the driver." It seems that devs who mostly work in user mode just naturally think "perform a callback" when they have the requirement "notify about an event."

### Enter the Inverted Call Model

So, what **is** the best way for a driver to notify a user-mode program that a given event has occurred? The answer to **that** question is simple: The driver should use what is referred to as the Inverted Call Model. We first described this technique in *The NT Insider* back in January of 2002: http://www.osronline.com/article.cfm?id=94.

Using this technique is much easier than its fancy name makes it sound. The application sends one or more requests (typically device controls, also known as IOCTLs) to the driver, with a pre-determine control code. The driver keeps these requests pending. When the driver needs to notify the application of the occurrence of an event, the driver simply completes one of the pending IOCTLs. The application discovers that the event has occurred by noticing that the previously issued IOCTL has been completed by the driver. Once notified, the application sends the IOCTL back to the driver where the driver once again holds the request pending until it needs to notify the application of another event.

That's all there is to it. I did tell you this was simple, didn't I?

One objection that new Windows driver writers often raise to this approach is that it seems like "an awful lot of overhead" to use an I/O completion as an event notification mechanism. But, in fact, this isn't true at all. The Windows I/O Subsystem and its companion Win32 are both highly optimized for handling I/O completions. If you're using asynchronous I/O (supplying an OVERLAPPED structure) to send multiple IOCTLs to the driver to hold for event notification purposes, using Completion Ports for handling the notification is particularly efficient. And, when you think about it, using something like a shared event between user-mode and kernel-mode has at least as much overhead as, and many more disadvantages than, the Inverted Call Model.

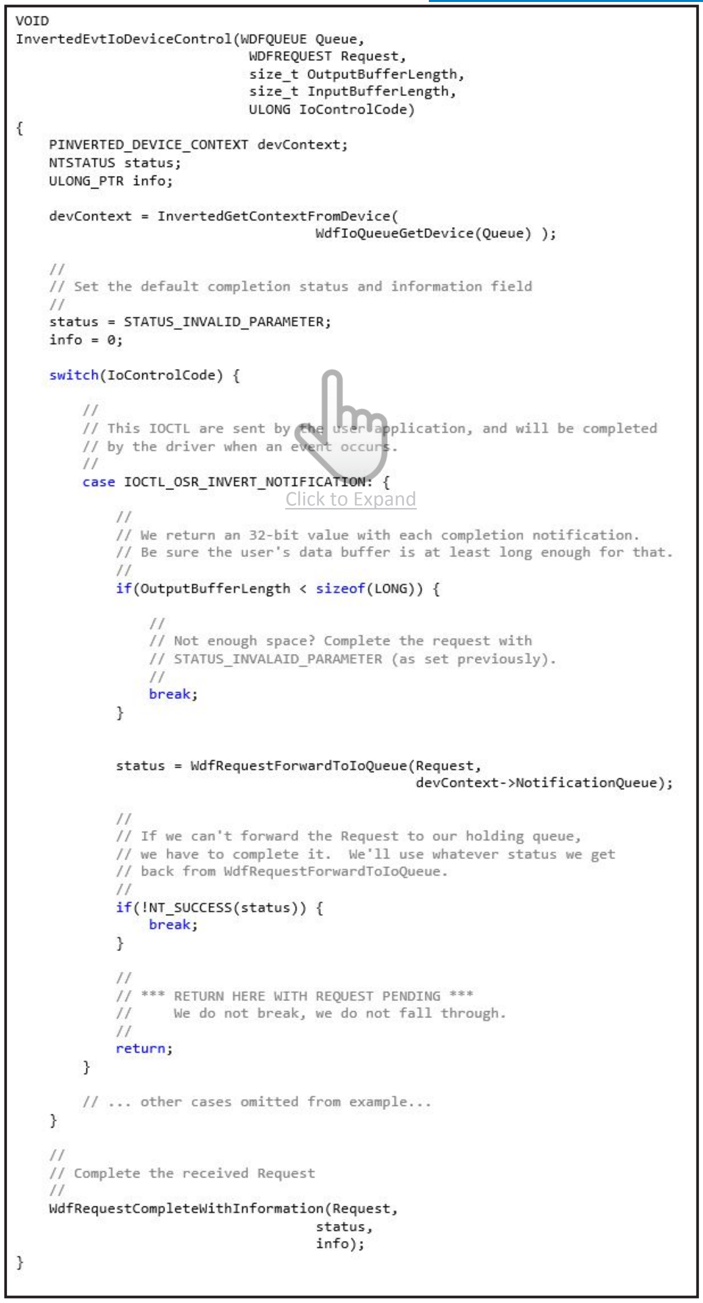
### Implementing the Driver

The pattern for implementing the Inverted Call Model in your driver couldn't be simpler. You define an IOCTL control code that the driver and the application will use for notification purposes:

```
#define IOCTL_OSR_INVERT_NOTIFICATION \
        CTL_CODE (FILE_DEVICE_INVERTED, 2049, METHOD_BUFFERED, FILE_ANY_ACCESS)
```

In the driver's *EvtIoDeviceControl* Event Processing Callback, the driver takes each IOCTL Request it received with the designated control code, and forwards it to a Queue with manual dispatching. The code for doing this is shown in **Figure 1**.

(CONTINUED ON PAGE 9)



```
VOID
InvertedEvtIoDeviceControl(WDFQUEUE Queue,
                            WDFREQUEST Request,
                            size_t OutputBufferLength,
                            size_t InputBufferLength,
                            ULONG IoControlCode)
{
    PINVERTED_DEVICE_CONTEXT devContext;
    NTSTATUS status;
    ULONG_PTR info;

    devContext = InvertedGetContextFromDevice(
                                WdfIoQueueGetDevice(Queue) );

    //
    // Set the default completion status and information field
    //
    status = STATUS_INVALID_PARAMETER;
    info = 0;

    switch(IoControlCode) {

        //
        // This IOCTL are sent by the user application, and will be completed
        // by the driver when an event occurs.
        //
        case IOCTL_OSR_INVERT_NOTIFICATION: {

            //
            // We return an 32-bit value with each completion notification.
            // Be sure the user's data buffer is at least long enough for that.
            //
            if(OutputBufferLength < sizeof(LONG)) {

                //
                // Not enough space? Complete the request with
                // STATUS_INVALID_PARAMETER (as set previously).
                //
                break;
            }

            status = WdfRequestForwardToIoQueue(Request,
                                        devContext->NotificationQueue);

            //
            // If we can't forward the Request to our holding queue,
            // we have to complete it.  We'll use whatever status we get
            // back from WdfRequestForwardToIoQueue.
            //
            if(!NT_SUCCESS(status)) {
                break;
            }

            //
            // *** RETURN HERE WITH REQUEST PENDING ***
            //     We do not break, we do not fall through.
            //
            return;
        }

        // ... other cases omitted from example...
    }

    //
    // Complete the received Request
    //
    WdfRequestCompleteWithInformation(Request,
                                      status,
                                      info);
}
```

Figure 1

## The Inverted Call Model in KMDF (Cont.)

(CONTINUED FROM PAGE 8)

In the example in **Figure 1**, when the driver is presented with a Request that has a control code value of IOCTL_OSR_INVERT_NOTIFICATION, it checks to see if the length of the output buffer associated with the Request is at least the size of a LONG integer (4 bytes). It does this because in this example when the driver notifies the application of an event, it will also return a 32-bit value containing additional information about the event. If the output buffer length is less than the size of a LONG integer, the driver completes the Request with STATUS_INVALID_PARAMETER.

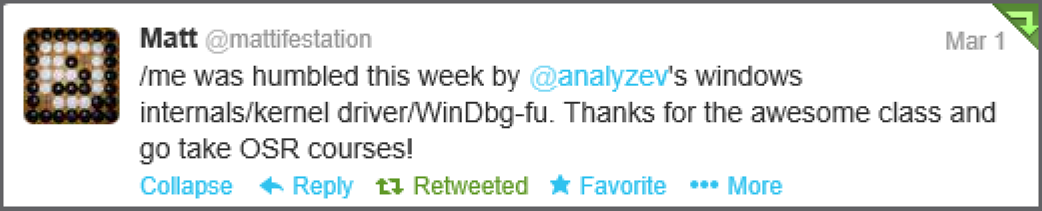
If the output buffer accompanying the Request is of sufficient length, the driver calls **WdfRequestForwardToIoQueue** to ask the Framework to place the Request on the driver's notification queue. This Queue is only used for holding IOCTL_OSR_INVERT_NOTIFICATION Requests until they are completed to notify the application of an event, and is configured with manual dispatching. The driver created the Queue in its *EvtDriverDeviceAdd* Event Processing Callback, and the handle for the Queue was saved in the WDFDEVICE's context area in the field named **NotificationQueue**. If the driver's attempt to forward the Request to its notification queue is not successful, the Request is completed with STATUS_INVALID_PARAMETER.

If the driver successfully forwards the Request to its notification queue, it simply returns. The Request *is not completed* in the EvtIoDeviceControl Event Processing Callback, and thus remains in progress (i.e. pending) until it is completed later by the driver. When an event occurs, and the driver wants to notify the application of the occurrence of that event, the driver simply removes the first pending IOCTL Request that it finds on its notification queue and completes that Request. Prior to completing the Request, the driver can optionally return

(CONTINUED ON PAGE 22)

## WHAT SHOULD I EXPECT FROM AN OSR SEMINAR?

Our students routinely tell us that their experience at an OSR seminar is THE most positive one they've had in technical training. Seriously. After a recent *Windows Internals & Software Drivers* seminar from OSR engineer Scott Noone, we were happy to see this tweet:



Matt @mattifestation Mar 1

/me was humbled this week by @analyzev's windows internals/kernel driver/WinDbg-fu. Thanks for the awesome class and go take OSR courses!

Training the Windows driver developers, debugging gurus and file system experts of the future is a particular joy for us. And, while we can't guarantee a training room with wallpaper as "attractive" as you see in the picture, we can guarantee you will learn a lot of good stuff.

Want to know more? Drop us an email, or better yet, call and speak live and in-person to an OSR seminar coordinator. We want you to be confident in choosing a topic and format that suits you or your team, and comfortable in that you'll be maximally prepared and attended to, from your initial contact, all the way through (and AFTER) your seminar experience is over.

Phone: +1.603.595.6500
Email: seminars@osr.com

# ! You're Dead
# Fixing Broken Debugger Extensions

As I'm sure you're all aware at this point, there are two different sets of PDB files generated for Microsoft provided components: private and public. The private symbols are used internally for source level debugging of the operating system. The public symbols are the private symbols with interesting bits stripped from them. These interesting bits include:

- Data structure types
- Local variable names
- Function parameters
- Global variable data types
- Source line information

Around the XP timeframe, it was determined that the public PDBs weren't as useful as they could be. Notably, the absence of the data structure type information made it difficult to debug OS level issues and created a maintenance burden for the public debugger release (anyone else remember when the debugger extensions had hard coded versions of the internal data structures?).

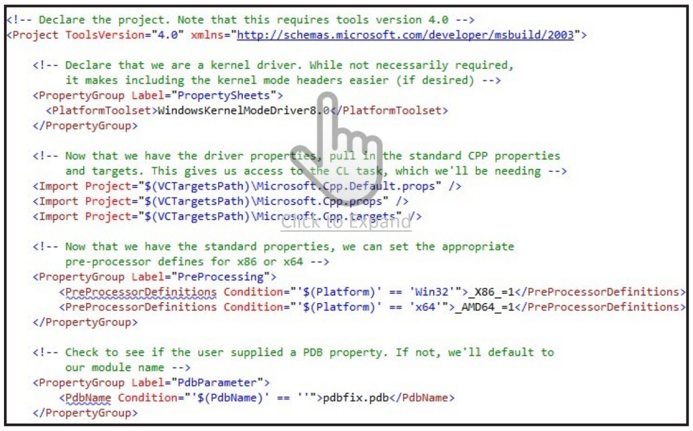
It was then decided that some of the interesting information lost during the stripping process should be put back into the public PDBs. Note that I did say **some** of the interesting information, not all. Thus, only data type information is added back into the public PDBs. To restrict this even further, only those data types considered to be necessary or appropriate for public use are added back into the PDBs.

Inevitably, this results in problems with the debugging tools. These tools are developed internally using private PDBs, therefore they don't always function properly when run against the public PDBs due to missing types. Luckily there **is** a way to fix this in some cases, as it is possible to add type information to an existing PDB.

The trick is to simply compile a C source file containing the types of interest and specify an existing PDB as the location of the debug information. Instead of overwriting this PDB, the compiler will add the missing types to the file. What's pretty neat about this solution is that you can even do this to override existing types in the PDB, which you could use to fix broken data structures present in the PDB (and, yes, this does happen too!).

## Invoking CL through MSBuild

To do this, we'll need to invoke the Microsoft C/C++ compiler directly via CL.EXE and carefully supply the debug parameters of interest. But, you didn't think that we'd wimp out and invoke CL directly from a command window, did you? Of course not! It's a new year and we have a new build system, so why **wouldn't** we do this through an MSBuild project file? The beginning of PDBFix.proj can be seen in **Figure 1**.



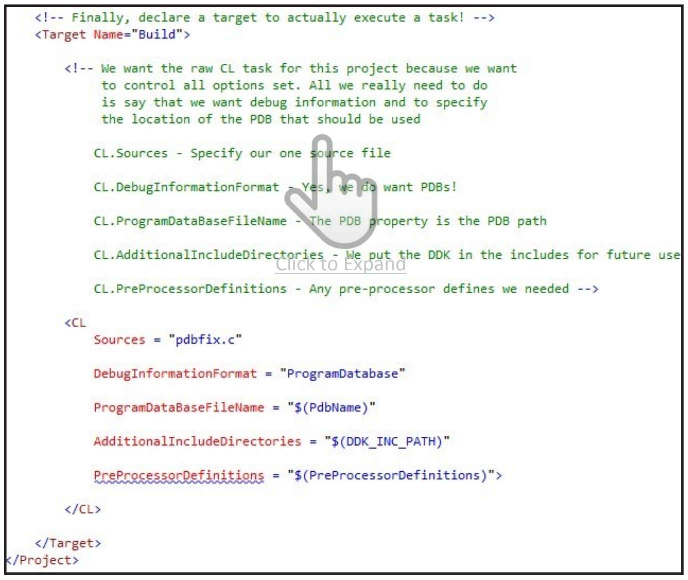
```
<!-- Declare the project. Note that this requires tools version 4.0 -->
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <!-- Declare that we are a kernel driver. While not necessarily required,
         it makes including the kernel mode headers easier (if desired) -->
    <PropertyGroup Label="PropertySheets">
      <PlatformToolset>WindowsKernelModeDriver8.0</PlatformToolset>
    </PropertyGroup>

    <!-- Now that we have the driver properties, pull in the standard CPP properties
         and targets. This gives us access to the CL task, which we'll be needing -->
    <Import Project="$(VCTargetsPath)\Microsoft.Cpp.Default.props" />
    <Import Project="$(VCTargetsPath)\Microsoft.Cpp.props" />
    <Import Project="$(VCTargetsPath)\Microsoft.Cpp.targets" />

    <!-- Now that we have the standard properties, we can set the appropriate
         pre-processor defines for x86 or x64 -->
    <PropertyGroup Label="PreProcessing">
        <PreProcessorDefinitions Condition="'$(Platform)' == 'Win32'">_X86_=1</PreProcessorDefinitions>
        <PreProcessorDefinitions Condition="'$(Platform)' == 'x64'">_AMD64_=1</PreProcessorDefinitions>
    </PropertyGroup>

    <!-- Check to see if the user supplied a PDB property. If not, we'll default to
         our module name -->
    <PropertyGroup Label="PdbParameter">
        <PdbName Condition="'$(PdbName)' == ''">pdbfix.pdb</PdbName>
    </PropertyGroup>
```

**Figure 1**

In our project file, we include the properties necessary for a kernel mode driver. This allows us to easily add the WDK include path to our compilation step. For this project, we want complete control over the parameters passed to CL.EXE, thus we import the CPP properties and targets to obtain access to the CL Task.

Minimally, compilation requires that we set the pre-processor define to indicate the processor architecture that we are compiling for. In this project, we rely on the *Platform* property to indicate which pre-processor define should be set. Lastly, we use the custom *PdbParameter* property to determine the location of the PDB file. This allows the user to supply the path to the PDB on the command line via the **/property (/p)** switch.

**Figure 2** shows the remainder of the project file, which is responsible for executing the CL Task.



```
    <!-- Finally, declare a target to actually execute a task! -->
    <Target Name="Build">

        <!-- We want the raw CL task for this project because we want
             to control all options set. All we really need to do
             is say that we want debug information and to specify
             the location of the PDB that should be used

            CL.Sources - Specify our one source file

            CL.DebugInformationFormat - Yes, we do want PDBs!

            CL.ProgramDataBaseFileName - The PDB property is the PDB path

            CL.AdditionalIncludeDirectories - We put the DDK in the includes for future use

            CL.PreProcessorDefinitions - Any pre-processor defines we needed -->

        <CL
            Sources = "pdbfix.c"

            DebugInformationFormat = "ProgramDatabase"

            ProgramDataBaseFileName = "$(PdbName)"

            AdditionalIncludeDirectories = "$(DDK_INC_PATH)"

            PreProcessorDefinitions = "$(PreProcessorDefinitions)">

        </CL>

    </Target>
</Project>
```

**Figure 2**

(CONTINUED ON PAGE 11)

## Fixing Broken Debugger Extensions (Cont.)

(CONTINUED FROM PAGE 10)

### Adding Types to the Source File

Now that we have our project file, the next step is to add data types to our source file, PDBFix.c. This is pretty straightforward; just add the data structure definition as you would in any other driver. The only thing to remember is **unused types do not go in the PDB**, thus you'll need to create a global variable that actually uses the type you define. An example PDBFix.c is shown in its entirety in **Figure 3**.

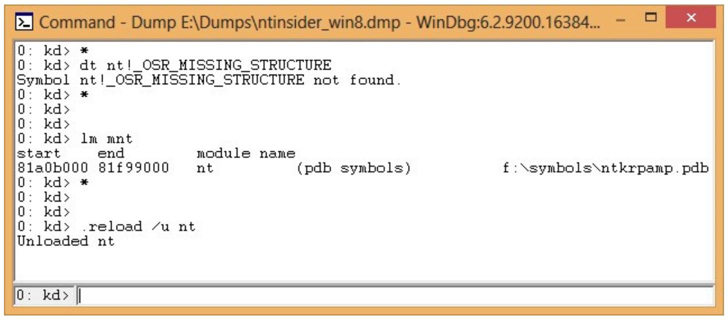
```
#include <ntddk.h>

typedef struct _OSR_MISSING_STRUCTURE {
         ULONG Field1;
         ULONG Field2;
}OSR_MISSING_STRUCTURE, *POSR_MISSING_STRUCTURE;

OSR_MISSING_STRUCTURE OsrMissingStructureDummy;
```

**Figure 3**

### Compiling the Code and Adding the Types

Let's now finally put this together and see how we can add types to an existing PDB using MSBuild. All we need to do is to build our PDBFix project, pointing the output PDB path of that project to an existing PDB file. This results in the linker **adding** the symbol definitions we create in our PDBFix project to the already existing PDB file. Pretty cool, eh?

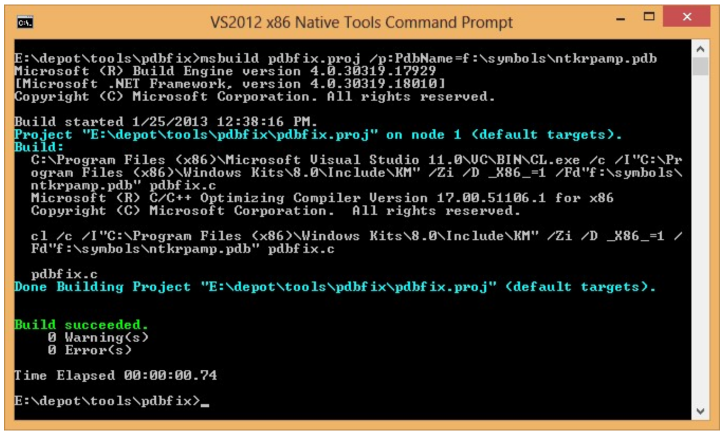
Let's walk through the steps: First, we need to find the location of the PDB that has the missing type information. Once you have the path of the PDB of interest, you'll need to unload the symbols for the module so that CL can modify the PDB. You can see an example of the steps necessary in **Figure 4**.


Command - Dump E:\Dumps\ntinsider_win8.dmp - WinDbg:6.2.9200.16384...

```
0: kd> *
0: kd> dt nt!_OSR_MISSING_STRUCTURE
Symbol nt!_OSR_MISSING_STRUCTURE not found.
0: kd> *
0: kd>
0: kd>
0: kd> lm mnt
start    end        module name
81a0b000 81f99000   nt         (pdb symbols)          f:\symbols\ntkrpamp.pdb
0: kd> *
0: kd>
0: kd>
0: kd> .reload /u nt
Unloaded nt

0: kd>
```

**Figure 4**

The next step is to open a VS2012 Tools Command Prompt of the appropriate architecture for the target, in this case x86. Once there, navigate to the location of the PDBFix project and execute MSBuild, passing the full path to the PDB as the *PdbName* property. If all goes well, you should see output resembling that in **Figure 5**.



```
E:\depot\tools\pdbfix>msbuild pdbfix.proj /p:PdbName=f:\symbols\ntkrpamp.pdb
Microsoft (R) Build Engine version 4.0.30319.17929
[Microsoft .NET Framework, version 4.0.30319.18010]
Copyright (C) Microsoft Corporation. All rights reserved.

Build started 1/25/2013 12:38:16 PM.
Project "E:\depot\tools\pdbfix\pdbfix.proj" on node 1 (default targets).
Build:
  C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC\BIN\CL.exe /c /I"C:\Pr
  ogram Files (x86)\Windows Kits\8.0\Include\KM" /Zi /D _X86_=1 /Fd"f:\symbols\
  ntkrpamp.pdb" pdbfix.c
  Microsoft (R) C/C++ Optimizing Compiler Version 17.00.51106.1 for x86
  Copyright (C) Microsoft Corporation.  All rights reserved.

  cl /c /I"C:\Program Files (x86)\Windows Kits\8.0\Include\KM" /Zi /D _X86_=1 /
  Fd"f:\symbols\ntkrpamp.pdb" pdbfix.c

  pdbfix.c
Done Building Project "E:\depot\tools\pdbfix\pdbfix.proj" (default targets).


Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:00.74

E:\depot\tools\pdbfix>_
```

**Figure 5**

(CONTINUED ON PAGE 24)

## Guest Article

# Another Look at Lint

**By Don Burn, community contributor**

It's been ages since *The NT Insider* published All About Lint - PC Lint and Windows Drivers which introduced driver writers to compile time checking tools. The original article pointed out that PC-lint was already fully integrated into the build process used by the DDK at that time, and included a configuration file to tailor the errors and warnings that PC-lint reported. Thanks to changes in the WDK development and build environments, the original setup no longer works well and when it does work it produces a large number of false positives. The new tools that accompany this article easily work with all versions of the Windows Driver Kit (WDK) and at the same time significantly reduce the spew that PC-Lint produces.

*"Like all good tools, PC-lint provides ways to easily disable any of its checks that you don't consider valuable. So don't be put off if a small number of the checks PC-lint performs are not to your liking."*

A lot of developers figure that once they have their driver building successfully at /W4, and passing PREfast and SDV without errors, they don't need to do any further checking. While some folks believe those tools are sufficient, I personally don't. My background is in fault-tolerant computing, and there **you can never check too much**.

This article is going to take a look at what PC-lint can do for you, and give you some tools and pointers to make the experience of using PC-lint easier. The original article referenced above has a lot of good examples of using PC-lint with drivers.

A number of the errors PC-lint will report are of categories that the Microsoft tools also check. Even here I find the tool to be of significant value. For example, if you fix a problem that PREfast identifies, PC-lint may flag the problem again because Lint's algorithms are different enough that the tool can identify an edge condition you missed in your original fix. For a code review you are likely to ask more than one person to review your code, right? Well, I think the same should be true of using tools to find potential bugs in your code.

Like all good tools, PC-lint provides ways to easily disable any of its checks that you don't consider valuable. So don't be put off if a small number of the checks PC-lint performs are not to your liking. Similarly, options can be used to enable a specific check that you want that the accompanying configuration file disables. In fact, Gimpel provides configuration files for some of the popular coding guidelines such as Scott Meyer's books and the MISRA consortium.

To me, the most valuable thing that PC-lint does is highlight code that is likely to contain omission errors. Omission errors are where you forget to do something, and tend to be some of the hardest category of errors to find. Thus, any assistance in locating this category of problem is likely to be very worthwhile. So let's take a look at some of the things that PC-lint can do for you.

PC-lint is exceptionally good at identifying unused fields and variables. Before you think the Microsoft tools identify unused variables sufficiently well, consider that Visual Studio with /W4 only catches unused **local** variables. If you have an unused static variable, global variable or structure member, neither the MS VC compiler nor PREfast will flag them; But PC-lint will identify them for you. In many cases I find that an unreferenced variable means that a piece of code is missing. There had to be a reason you declared the item in the first place, right? So what was it?

Related to unused variables are unreferenced variables and values. A simple case is initializing a variable but then never using it is the following:

```
ULONG index = 0;
```

If you never use the variable *index*, the compiler won't complain – but, once again, PC-lint will. Before you think, "who cares", I once encountered a driver that initialized a structure it allocated for each channel with over 128KB of data through several thousand lines of code, when only 256 bytes of the structure were ever used! That particular driver consumed over 4MB of space where 8KB was needed.

In the same category, if a function does not have the PREfast annotation *__checkreturn*, no tool except PC-lint will complain about:

```
status = Func1(Param1, Param2);
return;
```

If you don't care about the returned status, it is clearer to future maintainers reading your code if you cast the function call to void indicating that the return value is being ignored, like this:

```
(void)Func1(Param1, Param2);
return;
```

How about data and functions that are only used in one module? The original author may know what was intended, but the next person who maintains your code may appreciate if you put a static qualifier on the variable or function. It definitely

(CONTINUED ON PAGE 13)

## Another Look at Lint (Cont.)

(CONTINUED FROM PAGE 12)

makes it clear this is only used in this source file. PC-lint will flag these in its global wrap-up of the analysis.

PC-lint also does a better job of expression checking than any of the Microsoft tools. This can be a great help when you wonder why a complex piece of code in a conditional statement is never executed. A simple example of this can be found in a number of WDK driver samples, like the following:

```
ULONG ux;
...
if ( ux < 0 ) {
    // do something
}
```

Of course *ux* can never be less than zero because it's an nsigned variable! There were an amazing number of cases like this in the samples I used for testing, with a significant amount of code that will never be executed.

PC-lint tracks values and reports questionable actions. For example:

```
#define FLAG     0   // definition in some include
...
x = x & FLAG;
```

While this may be the intent, it does not hurt to be reminded that FLAG is zero, and it may confuse the next developer who looks at the code. In my testing, I found some cases where expressions evaluated to zero without any constants hiding the result.

Another area that PC-lint checks is compatibility of variables in expressions. A number of these can show up as integer overflow or underflow problems and lead to security holes or crashes. The Microsoft tools do some of this, but PC-lint is outstanding in finding potential problems such as mixing signed and unsigned variables in the same expression. PC-lint applies a stricter set of tests for loss of precision and loss of sign than Microsoft does. Think of these tests as the equivalent to PREfast's requirement for using safe string functions. Microsoft does provide safe integer runtimes, but few of us use them.

A number of things that PC-lint checks for could be best described as coding style verification. As previously mentioned, you can disable **any** of these checks (or any PC-lint check at all) if you wish. Take C preprocessor macro handling:

```
#define TEST_MACRO(a,b) (a + (b & 0x111)  + (2 * a) )
```

The above may look good but consider what happens when:

```
x = TEST_MACRO( i++, y + 2 );
```

The double increment of *i* and the calculation *y + 2 & 0x111* are probably not what you expected. PC-lint will flag these errors. [ED: This is also a great example of why macros are inherently evil and should be avoided in favor of inline functions.]

PC-lint will complain about multiple definitions of a global symbol. If the symbols are different types or qualifiers, this can cause some nasty problems even if the compiler accepts it. When the definitions are the same, having the redundant definition still makes changing the code harder, and is more likely to lead to confusion for the next developer who looks at the code.

(CONTINUED ON PAGE 14)

## Another Look at Lint (Cont.)

(CONTINUED FROM PAGE 13)

And then there are issues with being overly clever with Boolean expressions. For example, consider the following:

```
if (NdisEqualMemory(pStation->Config.DesiredBSSIDList[index],
                    StaEntry->Dot11BSSID,
                    sizeof(DOT11_MAC_ADDRESS)) == 1)
```

This example from the WDK may be obvious after a little reflection, but it could be written a lot more clearly, especially given that not everyone remembers the semantics of *memcmp* with a length of zero!

Another area that PC-lint checks is for bugs in switch statements. The tool will flag the lack of a default case and cases not ending in a break statement.

PC-lint will check for strange indentation that could indicate you messed up on closing braces. The Win7 WDK has an example of code where the indentation goes down in the middle of a compound statement for no reason; luckily that one is correct – the indentation is just broken. A similar case in a demo driver from elsewhere was actually a logic bug that would impact correct execution.

The tool also flags questionable semi-colons:

```
        while ( Hw->PhyState.RadioAccessRef != 0 ) ;
```

The above example from the WDK is correct, with the code being used to ensure all references by other threads have exited. It could easily have been a mistake, replacing the semi-colon with braces is preferable here.

And those are just a few things that PC-lint does **today**. One of the nicest things I like about PC-lint is that it keeps evolving. At present they have developed a minimal checking capability for multi-threaded programs. The current support does not provide much in the way of useful data for a Windows driver, but as the tool evolves this and other features will keep PC-lint in my mix of tools for a long time.

### Using PC-lint with the WDK

Since the original *All About Lint* article, a lot has changed in the Windows driver development environment so I've developed a new package called *Lint Driver Extensions (LDX).* You can download a zip archive containing this tool from the link provided at the conclusion of this article.

LDX is packaged with an installer. On your development system run the install package after you have installed the WDK and Gimpel's PC-lint. For PC-lint you should install all the patches from the Gimpel website and have either the co-msc110.* or co-msc100.* files installed. While the environment is optimized for PC-lint version 9.0 it works fine with the earlier version 8.0 (aside: Version 8.0 will both miss some errors and have more false positives.) . If you installed the package before the required software or if you add an additional WDK then just go into Add/Remove programs and select *Change* for the Lint Driver Extension package. This will update the software for all WDK's.

To use PC-lint with the WDK you have several options:

- If you are using an older WDK, the tool installs a command in the WDK's Bin directory called Lint. Simply preface your build command with lint e.g., *lint build <options>* or *lint prefast build <options>.* The Lint output is placed in a log file following the conventions of the build log with the name lint replacing build. For example, for a lint in the Windows 7 Checked build environment *lintchk_win7_x86.log* will be generated.
- For the Windows 8 WDK that is integrated with Visual Studio, you will find a new external tool on the tools menu. This is *LDX Solution* which will rebuild and lint the current solution, placing the results in the Output pane.
- If you are using a command line build with the Windows 8 WDK, then *LdxMsb* from the LDX install directory will run MSBuild to rebuild the solution with Lint and display the output to stdout.
- Finally the ultimate tool for using PC-Lint with the WDK is Riverblade's Visual Lint. This is a third party tool providing an integrated package that works inside VS2012. The tool is an add-on to PC-Lint which you must still purchase. The capabilities include background analysis of the project, coded display listings that like Visual Studio clicking on the error takes you to the line to edit and provides easy lookup of the description of the errors. The latest version of Visual Lint (4.0.2.198) is required for use with the WDK. The tool has a minor bug that if there are two subprojects with the same name, such as filter in the Toaster sample, one needs to be renamed for analysis to work. A fix is in the works.

  To use Visual Lint with the WDK choose *LintLdx.lnt* as the standard lint configuration file for the tool. There is a 30-day free trial of Visual Lint available so if you are considering PC-Lint, take a look at what Visual Lint can add to the experience. I expect to be using it for much of my work.

One of the biggest problems with the configuration from original article was the large number of spurious errors that were reported. As stated earlier my LDX tool contains a much richer lint options file that significantly reduces the errors. The file is far from a complete coverage of the entire set of kernel API's, and I'll gladly accept additions. The options **do** disable a number of errors that PC-lint complains about but which are false positives.

(CONTINUED ON PAGE 15)

## Another Look at Lint (Cont.)

(CONTINUED FROM PAGE 14)

Choosing errors to suppress is always something where people have strong opinions, so the new package provides the ability to customize the lint options. When the package is installed the first time, an empty file *LdxCustom.lnt* is placed in the PC-lint installation directory. Edit this file to add the options you wish to use for lint. Re-installing the tool will not change the LdxCustom.lnt file.

If you find problems with the package please let me know. My email address is at the end of the article. Hopefully I've encouraged you to grab PC-lint and give it a try. Using my LDX tool makes it easy. Enjoy writing better, more reliable, drivers using PC-lint!

Code associated with this article:

http://insider.osr.com/2013/code/ldx.zip

*Don Burn is a Windows system software architect, specializing in drivers and file systems, with 35 years of industry experience. He provides consulting services for Windows drivers, specializing in challenging problems for the Windows architecture. Don can be reached at burn@windrvr.com*

## MSBuild 101 (Cont).

(CONTINUED FROM PAGE 5)

### *ItemDefinitionGroup Element*

You'll note in the previous section that when we printed the metadata of *ProjectConfiguration*, we received a blank string for the first entry in the element. This is due to the fact that we did not provide a *MyMetaDescription* metadata element for the first entry.  Here's the definition of that *ProjectConfiguration* item group that we saw previously:

```
<ItemGroup>

    <ProjectConfiguration Include="Debug|Win32">
    </ProjectConfiguration>

    <ProjectConfiguration Include="Release|x64">
        <MyMetaDescription>This is an x64 build
        </MyMetaDescription>
    </ProjectConfiguration>

</ItemGroup>
```

Notice that, as we just mentioned, there's no *MyMetaDescription* for one of the values.  If we wanted to provide a default value for this metadata element in all *ProjectConfiguration* items, we could define an ItemDefinitionGroup element and create somewhat of a template for any *ProjectConfiguration* items defined:

```
<ItemDefinitionGroup>

    <ProjectConfiguration>
        <MyMetaDescription>No Description Available
        </MyMetaDescription>
    </ProjectConfiguration>

</ItemDefinitionGroup>
```

And we run our *PrintItemMetadata* task again and get the following results:

```
msbuild nothing.vcxproj /t:printitemmetadata
```
"No Description Available;This is an x64 build"

### *Import Element*

The Import element is an easy one, it simply allows you to include other MSBuild project files that define more Property, Item, and Target elements. Most MSBuild project files are going to have Import elements for the standard MSBuild files:

```
<Import Project="$(VCTargetsPath)\Microsoft.Cpp.Default.props" />

<Import Project="$(VCTargetsPath)\Microsoft.Cpp.props" />

<Import Project="$(VCTargetsPath)\Microsoft.Cpp.targets" />
```

### MSBuild Project File Processing

When MSBuild processes a project file, it evaluates all Property and Item elements, including imports, before evaluating the chosen Target. If a Property or Item element is declared multiple times, the last element wins. For example:

```
<PropertyGroup>
    <IncludePath>..\inc</IncludePath>
</PropertyGroup>

<PropertyGroup>
    <IncludePath>..\inc_other</IncludePath>
</PropertyGroup>
```

Results in *$(IncludePath)* evaluating to, "..\inc_other". However, Property and Item elements may be self-referential. For example, the following syntax is valid for a Property:

```
<PropertyGroup>
    <IncludePath>..\inc</IncludePath>
</PropertyGroup>

<PropertyGroup>
    <IncludePath>$(IncludePath);..\inc_other</IncludePath>
</PropertyGroup>
```

And provides the desired result of, "..\inc;..\inc_other".

To extend our ItemDefinitionGroup element, we simply need to use the metadata syntax:

```
<ItemDefinitionGroup>

  <ProjectConfiguration>
    <MyMetaDescription>No Description Available
    </MyMetaDescription>
  </ProjectConfiguration>

</ItemDefinitionGroup>

<ItemDefinitionGroup>

  <ProjectConfiguration>
    <MyMetaDescription>%(MyMetaDescription), seriously!
    </MyMetaDescription>
  </ProjectConfiguration>

</ItemDefinitionGroup>
```

And the result:

```
msbuild nothing.vcxproj /t:printitemmetadata
```
"No Description Available, seriously!;This is an x64 build"

### Build Related Property and Target Elements

Now that we've had our fun, we'd like to actually build some code! In order to drive the build procedure, we need to be sure to supply three bits of interesting information. First is the configuration, meaning whether or not we're building Debug or Release. This is done by providing a value for the *Configuration* Property element, typically on the command line.

Next is the platform to build for: x86 or x64. This is done using the *Platform* Property element, again typically passed on the command line. Annoyingly, MSBuild refers to the x86 platform as, "Win32", thus you'll need to use this name when you're building 32-bit versions of your driver.

Lastly, we need to specify a Target element that will actually do some useful work. The most common Target elements used are *Clean*, *Build*, or *Rebuild* and, again, this is typically specified on the command line.

(CONTINUED ON PAGE 17)

## MSBuild 101 (Cont).

(CONTINUED FROM PAGE 16)

**Project File Walkthrough**

Now we've (finally!) covered everything that we need to know to allow us to walk through a functional MSBuild project file. Note that some sections have been abbreviated from what we've shown earlier for clarity, though it is 100% functional as shown. In addition, none of the Property or Item elements or values shown here are custom, they are all well-known names that are used by the build procedure.

Example command lines to build this code would be:

```
msbuild nothing.vcxproj /p:platform=Win32 /
p:configuration=Debug /t:rebuild

msbuild nothing.vcxproj /p:platform=x64 /
p:configuration=Release /t:build
```

Let's talk a bit about the sections in the example shown in **Figure 1**. The numbers 1-8 in the example correspond to the numbered items described below.

1. The example starts with a PropertyGroup element that defines four properties:

    a. *TargetVersion* – Set to Windows8
    b. *PlatformToolset* – Set to WindowsKernelModeDriver8.0
    c. *ConfigurationType* – Set to Driver
    d. *DriverType* – Set to KMDF

    These properties are all "known" to the WDK build environment and cause it to build a KMDF driver for Windows 8. This makes sure that we link to the appropriate libraries, such as the KMDF wrapper, and have the proper include paths set for compilation.

(CONTINUED ON PAGE 18)

```
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/
developer/msbuild/2003">

    1.
    <PropertyGroup>
      <TargetVersion>Windows8</TargetVersion>
      <PlatformToolset>WindowsKernelModeDriver8.0</PlatformToolset>
      <ConfigurationType>Driver</ConfigurationType>
      <DriverType>KMDF</DriverType>
    </PropertyGroup>

    2.
    <ItemGroup>
        <ProjectConfiguration Include="Debug|Win32"/>
        <ProjectConfiguration Include="Release|x64"/>
    </ItemGroup>

    3.
    <PropertyGroup Condition="'$(Configuration)'=='Debug'">
        <UseDebugLibraries>true</UseDebugLibraries>
    </PropertyGroup>

    <PropertyGroup Condition="'$(Configuration)'=='Release'">
        <UseDebugLibraries>false</UseDebugLibraries>
    </PropertyGroup>

    4.
    <Import Project="$(VCTargetsPath)\Microsoft.Cpp.Default.props" /
>
    <Import Project="$(VCTargetsPath)\Microsoft.Cpp.props" />

    5.
    <ItemDefinitionGroup>
        <Link>
            <AdditionalDependencies>
                $(DDK_LIB_PATH)\wdmsec.lib;%(AdditionalDependencies)
            </AdditionalDependencies>
        </Link>
    </ItemDefinitionGroup>

    6.
    <ItemDefinitionGroup>
        <ClCompile>
            <AdditionalIncludeDirectories>
                ..\inc;%(AdditionalIncludeDirectories)
            </AdditionalIncludeDirectories>
        </ClCompile>
    </ItemDefinitionGroup>

    <PropertyGroup>
        <IncludePath>$(IncludePath);..\inc</IncludePath>
    </PropertyGroup>

    7.
    <ItemGroup>
        <ClCompile Include="nothing.c" />
    </ItemGroup>

    8.
    <Import Project="$(VCTargetsPath)\Microsoft.Cpp.targets" />

</Project>
```

**Figure 1—Fully Functional Project File**

## MSBuild 101 (Cont).

(CONTINUED FROM PAGE 17)

2. We specify our possible configurations as Item elements. In this example, we have two possible configurations: Win32 Debug and x64 Release. Again, this Item element and its values are known to the build environment. Thus, for example, specifying, "Debug|x86" or, "Release|Win64" would result in a build error indicating an unknown project configuration.

3. *UseDebugLibraries* is another known Property element to indicate whether this is a debug or release build. In the example, we show the optional *Conditional* attribute used on the PropertyGroup. This is pretty self-explanatory and allows you to conditionally set Property elements, though note that this syntax is not exclusive to PropertyGroup elements.

4. Here we import the standard Microsoft C++ compiler properties. Note that the placement of these statements is important as it will use the previously defined Item and Property elements to construct further elements. Also, we do this before our subsequent statements so that we can pass further information to the compiler and linker by modifying the elements defined by the props files.

5. Passing additional libraries to the linker can only be achieved via an ItemDefinitionGroup that overrides the *Link* item's metadata. Our need to link to wdmsec.lib here is fabricated, but we just wanted to show the technique.

6. We show two ways to modify the include path: one via an ItemDefinitionGroup element and the other via a Property element. There's no apparent advantage between the two, but we wanted to demonstrate that both methods were viable. Note that the *IncludePath* property only works because it is handled especially within *Microsoft.Cpp.Targets* (search, "$(IncludePath)" to see how!).

7. The *ClCompile* Item element is how we actually tell the compiler which files to compile. It makes sense for this to be an Item instead of a property because we are likely compiling multiple files as part of a project. In our simple project, we have a one value in the item: nothing.c. While there's nothing special about the item as it is declared here, because nothing.c exists as a file on disk the item takes on special properties. For example, we could use the built in *AccessedTime* metadata element to get the accessed time of the file.

8. Lastly, we include the Microsoft C++ targets definition file. This is going to define the targets that we use to build, rebuild, or clean our project.

### SOURCES Files are Dead! Long Live…er, SOURCES Files?

It's really too bad that the Windows 8 Driver Kit didn't provide XP support. While it takes some getting used to, once you know the basics the new build system is **far** more flexible (do some research on, "Property Functions" to see some crazy stuff you can do!) and sane than the previous one. Also, trust me when I say that having a wealth of documentation, books, and forums to reference is a nice change. However, with no ability to build drivers for XP, I'm sure we'll all be suffering with SOURCES files for many years to come.

*Code associated with this article:*

*http://insider.osr.com/2013/code/msbuild101.zip*

## To Switch, or Not to Switch?

After reading the MSBuild related articles in this issue, you're probably ready to dive in and start using the new build environment for upcoming projects. But, wait! What about all of those existing projects using SOURCES files? Should you switch **those** to the new build environment?

The short answer is yes, absolutely! This is the build environment of the future, so if you're planning on supporting the project through Windows 8 and beyond you'll need to make the switch eventually. However, don't forget that Windows XP targets are **not** supported in the new kit, thus you're still stuck with SOURCES and the old environment if you care about XP.

Once you settle on the fact that you want to switch, you have to ask yourself **how** you want to switch. As you may know, the WDK ships with a nifty utility to automatically convert your SOURCES based project into an MSBuild one: nmake2msbuild.exe. This utility is even easily accessible via the Visual Studio GUI, just navigate to File->Open->Convert Sources/Dirs. This means that you have two distinct paths available to switch your existing projects: you can have the WDK automatically convert them for you or cast away your SOURCES files and manually convert from scratch.

### To Automatically Convert?

Conversion using the WDK-provided conversion utility is a breeze; it should take only a few seconds and even provides a detailed log of the conversion steps that were performed. As

(CONTINUED ON PAGE 19)

## To Switch, or Not to Switch? (Cont.)

(CONTINUED FROM PAGE 18)

you can imagine, writing a utility that is able to convert **any** possible SOURCES file ever written in history to a new build system is a massive undertaking. For the most part, we found the utility did an admirable job of converting the various projects that we threw at it. It particularly did well on standard driver projects that didn't attempt to do anything exotic in the old build system. We were able to convert, compile, and load the resulting binaries of these projects without issue.

The conversion didn't do nearly as great of a job, however, when applied to really complex projects involving both user and kernel mode components. We had problems ranging from the conversion failing, to the conversion succeeding but the compile or link steps failing when we tried to build the converted project. In one case, a static kernel library was improperly identified as a static user library, which resulted in the wrong include paths being used. In another, the resulting project failed to specify an entry point to the linker. In one case we were able to massage the project files into the correct state to work, in the other we simply gave up.

In the case where the conversion does succeed, the end product of the conversion is a bit of an oddity when compared to other MSBuild project files. Your SOURCES file stays mostly intact, though your directives are converted into property elements and stored in a sources.props file:

```
<PropertyGroup>
  <TARGETNAME Condition="'$(OVERRIDE_TARGETNAME)'!='true'">NOTHING</TARGETNAME>
  <TARGETTYPE Condition="'$(OVERRIDE_TARGETTYPE)'!='true'">DRIVER</TARGETTYPE>
  <INCLUDES Condition="'$(OVERRIDE_INCLUDES)'!='true'">..\inc</INCLUDES>
  <C_DEFINES Condition="'$(OVERRIDE_C_DEFINES)'!='true'">-DFOODEF=1</C_DEFINES>
  <SOURCES Condition="'$(OVERRIDE_SOURCES)'!='true'">nothing.c</SOURCES>
</PropertyGroup>
```

As part of being a driver, your project file imports WDK properties files. These files import the sources.props file and convert the property elements into other item and property elements, which are then used to drive the compilation process. For example, within the WDK's PostToolsetRules.props the *SOURCES* property is converted into the *ConvertedSources* property:

```
<ConvertedSources>$(SOURCES.ToLower())</ConvertedSources>
```

Which is then converted into the *ClSourceFiles* item using a regular expression match looking for C and C++ files:

```
<ClSourceFiles   Include="$([System.Text.RegularExpressions.Regex]::Matches
($(ConvertedSources), '(?%3C=^|;)[^;]+\.(c|cpp|cxx)(?=;|$)'))"/>
```

Now, we jump back to our project file and see the *ClSourceFiles* item included in the *ClCompile* item, which actually indicates the files we want to compile:

```
<ClCompile Include="@(ClSourceFiles)" Exclude="@(ClCompile)" />
```

Phew! Quite the trip that information has to take, isn't it? To be fair, this is all compile time overhead so it's not **that** big of a deal in terms of performance. However, it does increase the complexity of the build files when it comes time to maintain or debug them.

### To Manually Convert?

Manual conversion means starting from complete scratch or with the new project wizard in Visual Studio. You'll need to import all of your source files and settings manually or via the GUI. So, manually converting your project will take a lot longer than using the WDK built-in tool before you actually get something working. In order to go this route, you'll likely need to understand not only your old build procedure but also the new one.

The **good** news is that even though driver writers are just getting around to using it, MSBuild has been around for quite a while. If you stumble along the way, you'll surely find support out there someplace to get you back on track.

### *The Answer…*

In our opinion, converted projects that are built using the WDK-provided conversion utility are simply not worth the extra complexity relative to real projects. Using the conversion tool is a great way to jump into the new kit and get something to play with. In many cases it will simply, "just work" and you'll be able to spend the afternoon playing with SDV and Code Analysis.

However, in terms of long term maintainability it just doesn't make sense to ship products using converted projects. Not converting also provides you the opportunity to take a critical eye to your build practices and get rid of some cruft you've accumulated over the years. It's also a good time to think about your static analysis policies. Should you pass Code Analysis clean on all warnings? What about SDV? /W4? Just remember, the short term pain may just pay off in the long run!

## Reader/Writer Spin Locks (Cont.)

(CONTINUED FROM PAGE 7)

**Exclusive** is that you **must call the correct function** based on the mode of the lock when you release it. In other words, if your call to **ExTryToConvertSharedSpinLockExclusive** succeeds, you hold the lock in exclusive mode, and therefore must release it by calling **ExReleaseSpinLockExclusive**. Likewise, if your call to **ExTryToConvertSharedSpinLockExclusive** fails you still own the lock in shared mode, and must call **ExReleaseSpinLockShared** to release it.

Having to know the mode in which you're holding the Reader/Writer Spin Lock so you can use the matching call to release it is annoying. Some people might claim it provides nice built-in documentation, but all I see is a built-in opportunity for a bug or a cut/paste error. When you release an ERESOURCE, you simply call a common release function and the function knows if you were holding the lock shared or exclusive. Perhaps that's not really a fair comparison, because ERESOURCES have **much** more overhead than Reader/Writer Spin Locks. However, even given the low cost algorithm Windows uses for Reader/Writer Spin Locks, the release function **could** in fact know whether the lock was currently held shared or exclusive and just "do the right thing." See the sidebar, Why Not a Single Function for Release at the conclusion of this article for an extended discussion of why there isn't a single release function.

### Usage Details

Enough about syntax and calling patterns. Let's now turn our attention to the details of how you use these locks.

You acquire Reader/Writer Spin Locks from code running at IRQL DISPATCH_LEVEL or lower. Regardless of the mode in which the lock is acquired, there is no timeout on the acquisition attempt and you don't return from the call until the lock has been acquired. When you do return, you're holding the lock and your thread is running at IRQL DISPATCH_LEVEL. There are no functions available that attempt to acquire the lock and return without waiting if the lock is not immediately available.

It should really go without saying but I'm going to say it anyways: Reader/Writer Spin Locks and traditional kernel Spin Locks are different data types and cannot be used interchangeably. You can't, for example, call **KeAcquireSpinLock** on an EX_SPIN_LOCK in one place, and call **ExAcquireSpinLockShared** on that same EX_SPIN_LOCK in another place. **KeAcquireXxx** and **KeReleaseXxx** can only be used with traditional KSPIN_LOCKs. And **ExAcquireXxx** and **ExReleaseXxx** can only be used on Reader/Writer Spin Locks that always have the EX_SPIN_LOCK data type.

Note that the calls to acquire Reader/Writer Spin Locks differ slightly from the pattern used by **KeAcquireSpinLock**, in that the **ExAcquireSpinLockXxxx** functions return the previous IRQL at which the calling thread was running as the function's return value. I think that's rather handy.

So, how do attempts to acquire the lock in various modes simultaneously interact? If you attempt to acquire the lock in exclusive mode and there are one or more threads that already hold the lock in shared mode, the exclusive mode acquisition attempt will:

- Block any subsequent requests to obtain the lock shared and
- Wait until all the current shared holders have released the lock.

This is done to ensure the lock is granted as promptly as possible to an exclusive requestor.

Obviously, only one thread can hold the lock in exclusive mode at a time. When multiple threads are waiting to acquire the lock in exclusive mode, the lock is granted to waiters in random order. The way the lock is granted to exclusive waiters very closely resembles the mechanism used for traditional (non-reader/writer) Spin Locks.

The overhead for acquiring a Reader/Writer Spin Lock exclusive is approximately the same as that for a traditional Spin Lock. It's important to recall what we said previously, that these are not queued locks. Thus, they can suffer from the same propensity as traditional Spin Locks to favor some waiters over others in certain systems.

The overhead of acquiring a Reader/Writer Spin Lock shared is also low and involves little more than an increment, an assign, a test, and an interlocked operation. Couldn't be simpler, could it?

We previously described the function **ExAcquireSpinLockXxxxAtDpcLevel** and noted that it avoids setting the IRQL to DISPATCH_LEVEL like the function **ExAcquireSpinLockXxxx** does. At that point we said that calling **ExAcquireSpinLockXxxxAtDpcLevel** can be used as a small optimization when you know in advance that you're already running at IRQL DISPATCH_LEVEL. But those of you who like to push beyond the limit of what's documented might like to know that **ExAcquireSpinLockSharedAtDpcLevel** preserves the tradition started by **KeAcquireSpinLockAtDpcLevel** by allowing calls at **any** IRQL greater than or equal to DISPATCH_LEVEL. Note the SAL notations (from the WDK for Win8):

```
_IRQL_requires_min_(DISPATCH_LEVEL)
VOID
ExAcquireSpinLockSharedAtDpcLevel (
    _Inout_ _Requires_lock_not_held_(*_Curr_)
    _Acquires_lock_ (*_Curr_)
    PEX_SPIN_LOCK SpinLock
    );
```

(CONTINUED ON PAGE 21)

## Reader/Writer Spin Locks (Cont.)

(CONTINUED FROM PAGE 20)

The IRQL notation doesn't require IRQL DISPATCH_LEVEL, it simply requires any IRQL greater than or equal to DISPATCH_LEVEL. Interesting, eh? What this does is allow you to construct your own Reader/Writer Spin Locks that work at specific IRQLs greater than IRQL DISPATCH_LEVEL. Just raise the IRQL to the IRQL of your new spin lock (say, DIRQL), and call **ExAcquireSpinLockXxxxAtDpcLevel**. Note that we're not recommending this. We're simply saying that if you need a feature like this, and you don't mind writing code using a feature that's neither supported nor documented, this is a possibility.

### Summary

Starting in Windows Vista SP1, Windows introduced a cool new set of synchronization primitives: Reader/Writer Spin Locks. Using these, you can properly support multiple simultaneous read accessors to shared data, and create code that's more scalable. Here at OSR, we've used these "new" primitives extensively in code that runs on Windows 7 and later. We recommend you check them out.

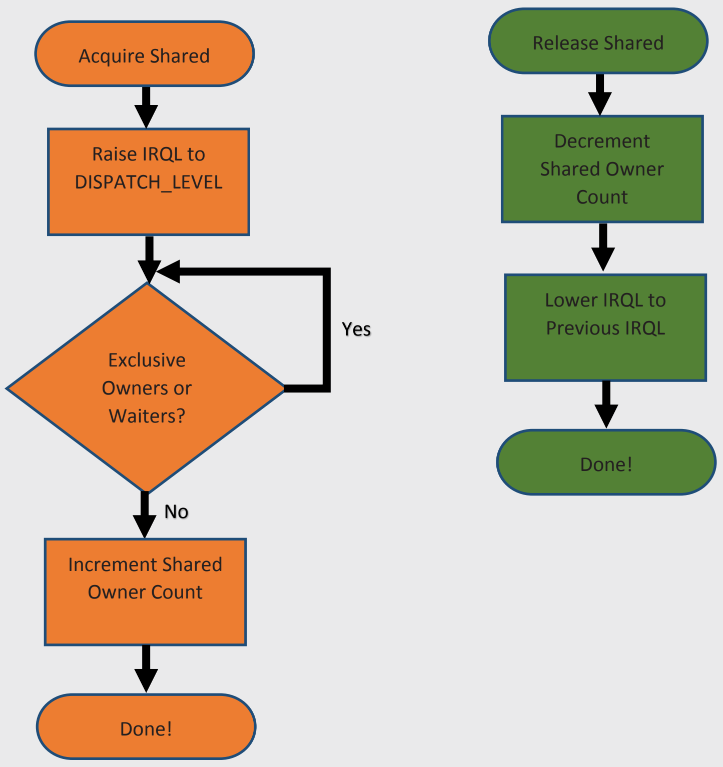
## Why Not a Single Function for Release?

When you want to release a Reader/Writer Spin Lock, you have to call the function that matches the mode in which you're holding the lock. If you're holding the lock shared, you need to call **ExReleaseSpinLockShared**. If you're holding it exclusive, you need to call **ExReleaseSpinLockExclusive**. Screw up and you screw up the lock. It's dumb, if you ask me. Just another unnecessary opportunity to create a bug.

So why don't we have a common release function for Reader/Writer Spin Locks? Well, one reason is almost certainly the desire to keep the overhead for releasing locks as absolutely low as possible. But I suspect the **real** reason is rooted in history. If you look at WDM.H, you'll see that the traditional (non-reader/writer) Spin Locks all have alternate names that start with **Ex** instead of **Ke**:

```
#define ExAcquireSpinLock(Lock, OldIrql) \
                     KeAcquireSpinLock((Lock), (OldIrql))
#define ExReleaseSpinLock(Lock, OldIrql) \
                     KeReleaseSpinLock((Lock), (OldIrql))
#define ExAcquireSpinLockAtDpcLevel(Lock) \
                     KeAcquireSpinLockAtDpcLevel(Lock)
#define ExReleaseSpinLockFromDpcLevel(Lock) \
                     KeReleaseSpinLockFromDpcLevel(Lock)
```

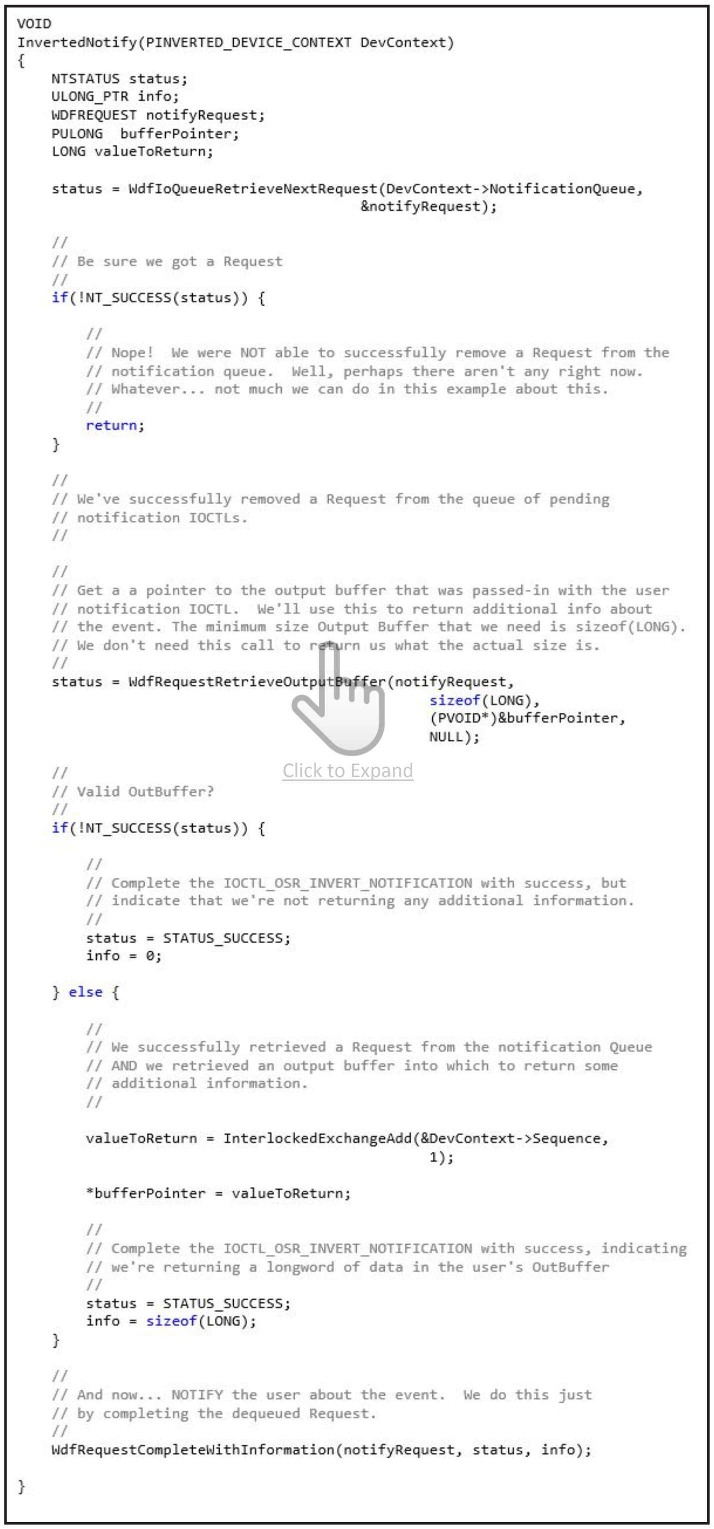
I bet you didn't know that! These executive names for the kernel spin lock functions have in fact been defined since the earliest days of Windows NT. And now that we have Reader/Writer Spin Locks, if we wanted to have a common function for releasing the lock without regard to whether we were holding it shared or exclusive, what would we name that function? Well, if we were to follow the current naming pattern you would probably want to name that function **ExReleaseSpinLock**. But, oooops! We already **have** a function with that name. And if we named it something out of the pattern such as **ExReleaseReaderWriterSpinLock**, that would just be an opportunity for developers to create a bug by accidentally coding **ExReleaseSpinLock** anyhow. Perhaps having to know the mode that you're holding the lock so you can call the appropriate function to release it isn't so bad after all. Oh well, whatever, never mind.


Acquire Shared
Raise IRQL to DISPATCH_LEVEL
Exclusive Owners or Waiters?
Yes
No
Increment Shared Owner Count
Done!
Release Shared
Decrement Shared Owner Count
Lower IRQL to Previous IRQL
Done!

## The Inverted Call Model in KMDF (Cont.)

(CONTINUED FROM PAGE 9)

additional information associated with the event such as an event code. This procedure is shown in **Figure 2**.



```
VOID
InvertedNotify(PINVERTED_DEVICE_CONTEXT DevContext)
{
    NTSTATUS status;
    ULONG_PTR info;
    WDFREQUEST notifyRequest;
    PULONG  bufferPointer;
    LONG valueToReturn;

    status = WdfIoQueueRetrieveNextRequest(DevContext->NotificationQueue,
                                           &notifyRequest);

    //
    // Be sure we got a Request
    //
    if(!NT_SUCCESS(status)) {

        //
        // Nope!  We were NOT able to successfully remove a Request from the
        // notification queue.  Well, perhaps there aren't any right now.
        // Whatever... not much we can do in this example about this.
        //
        return;
    }

    //
    // We've successfully removed a Request from the queue of pending
    // notification IOCTLs.
    //

    //
    // Get a a pointer to the output buffer that was passed-in with the user
    // notification IOCTL.  We'll use this to return additional info about
    // the event. The minimum size Output Buffer that we need is sizeof(LONG).
    // We don't need this call to return us what the actual size is.
    //
    status = WdfRequestRetrieveOutputBuffer(notifyRequest,
                                            sizeof(LONG),
                                            (PVOID*)&bufferPointer,
                                            NULL);

    //
    // Valid OutBuffer?
    //
    if(!NT_SUCCESS(status)) {

        //
        // Complete the IOCTL_OSR_INVERT_NOTIFICATION with success, but
        // indicate that we're not returning any additional information.
        //
        status = STATUS_SUCCESS;
        info = 0;

    } else {

        //
        // We successfully retrieved a Request from the notification Queue
        // AND we retrieved an output buffer into which to return some
        // additional information.
        //

        valueToReturn = InterlockedExchangeAdd(&DevContext->Sequence,
                                               1);

        *bufferPointer = valueToReturn;

        //
        // Complete the IOCTL_OSR_INVERT_NOTIFICATION with success, indicating
        // we're returning a longword of data in the user's OutBuffer
        //
        status = STATUS_SUCCESS;
        info = sizeof(LONG);
    }

    //
    // And now... NOTIFY the user about the event.  We do this just
    // by completing the dequeued Request.
    //
    WdfRequestCompleteWithInformation(notifyRequest, status, info);

}
```

**Figure 2**

As shown **Figure 2**, in the function *InvertedNotify*, the driver starts by attempting to remove a Request from its notification queue, by calling **WdfIoQueueRetrieveNextRequest**. The Request the driver is attempting to remove is an IOCTL with the control code IOCTL_OSR_INVERT_NOTIFICATION. This Request would have been forwarded to the notification queueby the driver's *EvtIoDeviceControl* Event Processing Callback that was described previously.

If the driver is not successful in getting a Request from its Notification Queue the driver just returns from this function. This would be the case when, for example, the notification queue is empty. In this case, the driver will either need to store the event in some driver-specific way, or the user application will simply not be notified of this event occurrence. If it's important for the application to not miss any event notifications, it will need to be sure to always keep one or more IOCTL_OSR_INVERT_NOTIFICATION Requests pending in the driver. The application would do this by calling **DeviceIoControl** multiple times, most likely using asynchronous completing with an OVERLAPPED structure for each request, and by immediately re-issuing the IOCTL each time it is completed by the driver to notify the application of an event occurrence.

If the driver successfully removes a Request from its notification queue, it calls **WdfRequestRetrieveOutputBuffer** to get a pointer to the output buffer associated with the Request. Recall that on receiving the IOCTL, the driver verified that an output buffer was supplied with the Request and that the output buffer was at least **sizeof**(LONG) bytes in length. If the driver is successful in getting a pointer to application's specified output buffer, it returns into that output buffer additional information associated with the event. In the example, this information is an event sequence number – a signed 32-bit value that increases by one for each event signaled back to the user.

Whether the driver was successfully able to get a pointer to the application's output buffer or not, the driver next completes the Request by calling **WdfRequestCompleteWithInformation**. The application interprets this completion as notification of the event occurring. If an output buffer was specified, the application can examine the additional information about the event returned by the driver into that buffer.

### Looks Too Simple

Well, of course it looks simple. Because it **is** simple. We're using KMDF, after all. Note that one very important, but perhaps easily overlooked, feature of the example is that it keeps the Requests to be completed when an event occurs on a WDFQUEUE. This isn't chance, and you couldn't (for example) easily substitute a WDFCOLLECTION or a vector of WDFREQUESTS for that WDFQUEUE. This is because the WDFQUEUE automatically handles request cancellation. When the application exits with Requests in progress on the notification queue, the Framework will automatically complete

(CONTINUED ON PAGE 23)

## The Inverted Call Model in KMDF (Cont.)

(CONTINUED FROM PAGE 22)

those Requests with STATUS_CANCELLED. The driver doesn't even have to get involved.

Another detail that you'll notice when you read the full example is the notification queue is setup so that it is not power-managed. That is, the **PowerManaged** field of the WDF_IO_QUEUE_CONFIG structure is set to FALSE. This is actually pretty important. If the notification queue was power managed, the driver's ability to remove Requests from that Queue would change based on the device's power state (D State). So if an event occurred that the driver wanted to notify the application about while the device was in a low power D State, that call to **WdfIoQueueRetrieveNextRequest** would fail with a return status of STATUS_WDF_PAUSED.

### The Application Side?

A gentleman was asking in the NTDEV forum recently how to implement the user side of this model. Frankly, we don't do user-mode here at OSR, so there's not a lot we can tell you that's both clever and definitive. However, we **can** repeat what we said earlier in this article: You probably want to implement this notification using asynchronous I/O and completion ports, and you almost certainly want to be careful in your application to send up enough notification requests to ensure that the driver always has one pending. That doesn't seem very difficult to us.

Just to give you an idea of how you might approach the user-mode side of things, we threw together an example application that you can find in the ZIP archive accompanying this article. But be warned: It's just test code. We don't care that it's sloppy or that it never calls ***free*** or ***CloseHandle***, or that it's got bugs. We already know its crap, so don't email us to tell us about how badly the code is written. Of course, please **do** feel free to write something that's proper, elegant, attractive, and well-documented... and then send it to us. We'll be happy to share it with the community.

### Natural Extensions

We've outlined just the basics in this article. There are several extensions to this model that should be relatively easy for you to implement. For example, on receiving the event notification you might want the user-mode application to do some processing and return some data back to kernel mode. To do this, the user-mode app could copy the sequence number that it received with the original event notification to a structure in its input buffer. Then the app would then add any data it wants to send to kernel mode to that structure, and sends the IOCTL_OSR_INVERT_NOTIFICATION back to the driver. When the driver receives this IOCTL it would process the information received from the application's input buffer and then forward the received IOCTL Request to the notification queue. Part of the processing done by the driver might be to match the supplied sequence number with an event the driver has buffered. Or something. I hope you get the idea.

### Summary

That's the Inverted Call Model. When you want your driver to be able to notify a user-mode application about the occurrence of an event, have the driver complete a request back to the application! It's an easy model to implement, it's easy to test, and it's remarkably efficient. Just be sure to have the application keep enough IOCTLs in progress at the driver, so there's always one available when the driver wants to notify it of an event.

*Code associated with this article:*

http://insider.osr.com/2013/code/inverted_call_example.zip

WINDOWS FILE SYSTEM DEVELOPMENT SEMINAR

A Must for Mini-Filter Developers

Whether developing file systems, or file system mini-filters, OSR's *Developing File Systems for Windows* seminar has proved year after year to be the most effective way to understand the fundamentals of the Windows file system "interface", and the subtle nuances of file system and mini-filter development on Windows.

Next presentation:

Boston/Waltham, MA
23-26 April

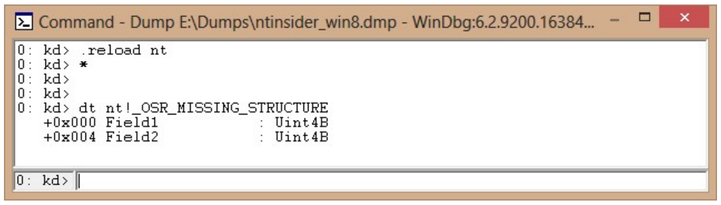
To communicate directly with our seminar coordinator:

Phone: +1.603.595.6500
Email: seminars@osr.com

## Fixing Broken Debugger Extensions (Cont.)

(CONTINUED FROM PAGE 11)

We can now return to WinDBG, reload our now fixed PDB, and see that the type information has indeed been added (**Figure 6**).



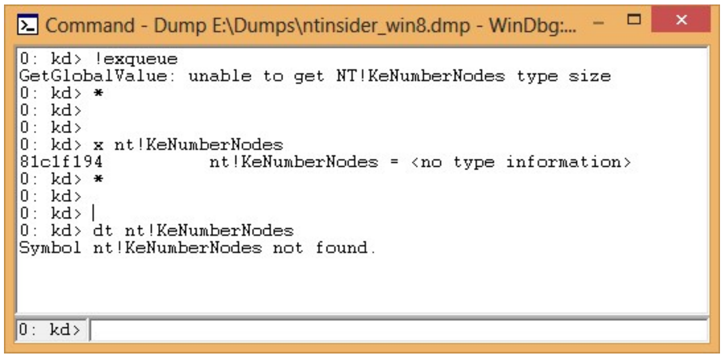
```
0: kd> .reload nt
0: kd> *
0: kd>
0: kd>
0: kd> dt nt!_OSR_MISSING_STRUCTURE
   +0x000 Field1           : Uint4B
   +0x004 Field2           : Uint4B

0: kd>
```

**Figure 6**

**Applying our Technique: Fixing !exqueue on Windows 8 x86 Targets**

Now that we know how to add information to PDBs, we can use what we've learned to fix the **!exqueue** command on Windows 8 x86 targets. This is a command that we at OSR use all the time, thus the fact that it's not working in some cases is an annoyance. We filed a bug report on the problem when we discovered it a while back, but we're impatient so we set out to fix it ourselves in the meantime.

First, let's see what happens when we run the command against the public symbols, as shown in **Figure 7**.



```
0: kd> !exqueue
GetGlobalValue: unable to get NT!KeNumberNodes type size
0: kd> *
0: kd>
0: kd>
0: kd> x nt!KeNumberNodes
81c1f194          nt!KeNumberNodes = <no type information>
0: kd> *
0: kd>
0: kd> |
0: kd> dt nt!KeNumberNodes
Symbol nt!KeNumberNodes not found.

0: kd>
```

**Figure 7**

As we discussed earlier, in creating the public PDBs, the data types associated with global variables are stripped out. Thus, as we see from **Figure 7**'s **x** and **dt** command outputs, we have no type information to go with the global and therefore the debugger cannot query its size.

Note that, up to this point, we have only discussed adding missing data types to an existing PDB and have said nothing about associating a global with a type. This isn't by accident, after several attempts we were unsuccessful in getting this information wired back into the PDB. This is unfortunate, as it would have been an elegant solution to an otherwise ugly problem. All is not lost, however, as we discovered an ugly solution instead.

(CONTINUED ON PAGE 25)

## Fixing Broken Debugger Extensions (Cont.)

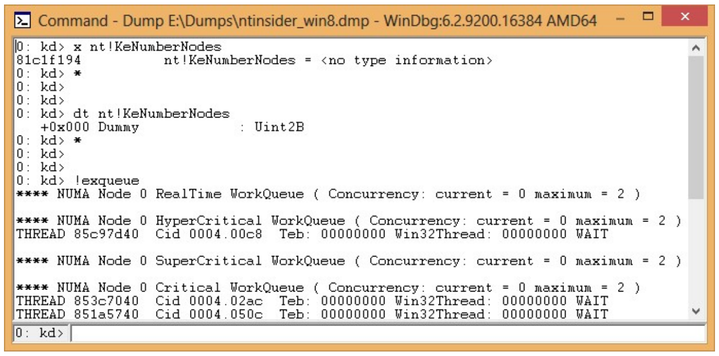
(CONTINUED FROM PAGE 24)

As it turns out, it's possible to trick the debugger into getting a size for the global by creating an appropriately sized data type of the same name. In the case of *KeNumberNodes*, dumping the memory location with the **d** command shows that the global is likely at most two bytes long. In order to account for this, we created a data type *KeNumberNodes* with a single USHORT member (**Figure 8**).

```
#include <pshpack1.h>
struct KeNumberNodes {
     //
     // Add a dummy member to get the size right.
     //
     USHORT Dummy;
};
#include <popack.h>

//
// Unused types are NOT in the PDB! So, make a dummy variable to
// get the type in the PDB.
//
struct  KeNumberNodes KeNumberNodesDummy;
```

**Figure 8**

While the **x** command will still not be able to resolve a type for the global variable, **dt** will pick up the type definition and correctly interpret the size of that structure as two bytes. This trick is enough to fool the **!exqueue** command into getting a size for the global, resulting in it happily dumping out the worker queues (**Figure 9**).


Command - Dump E:\Dumps\ntinsider_win8.dmp - WinDbg:6.2.9200.16384 AMD64

```
0: kd> x nt!KeNumberNodes
81c1f194          nt!KeNumberNodes = <no type information>
0: kd> *
0: kd>
0: kd>
0: kd> dt nt!KeNumberNodes
   +0x000 Dummy            : Uint2B
0: kd> *
0: kd>
0: kd>
0: kd> !exqueue
**** NUMA Node 0 RealTime WorkQueue ( Concurrency: current = 0 maximum = 2 )

**** NUMA Node 0 HyperCritical WorkQueue ( Concurrency: current = 0 maximum = 2 )
THREAD 85c97d40  Cid 0004.00c8  Teb: 00000000 Win32Thread: 00000000 WAIT

**** NUMA Node 0 SuperCritical WorkQueue ( Concurrency: current = 0 maximum = 2 )

**** NUMA Node 0 Critical WorkQueue ( Concurrency: current = 0 maximum = 2 )
THREAD 853c7040  Cid 0004.02ac  Teb: 00000000 Win32Thread: 00000000 WAIT
THREAD 851a5740  Cid 0004.050c  Teb: 00000000 Win32Thread: 00000000 WAIT
0: kd>
```

**Figure 9**

It's definitely a hack, but the usefulness of the extension makes it worth it until the command is fixed in a future release.

### Conclusion

Sources for this article are provided for download, giving you a method to fix most categories of public symbol issues you may come across. If you use it to resolve any problems let us know, one less broken command for us to stumble over when we need it!

*Code associated with this article:*

http://insider.osr.com/2013/code/fix.zip

## Peter Pontificates (Cont.)

(CONTINUED FROM PAGE 3)

quarter given that the US presidential election took place more than 4 months ago. I guess I could throw that flyer away now, huh? Lest there be any doubt, I am keeping the Motorola antenna color chart.

So, I think you can guess that when the Surface Pro was announced, I wasn't very interested. Fool me once, shame on you and all that. Sure, the Surface Pro has an x86 processor. And yes, you can run actual applications on it. But I had already lived the Surface experience (if you can fairly call using the device for a total of two hours "living") and I wasn't interested in going that route again.

Then the guy who has an office across the hall from me got one. In fact, he went so far as to pre-order his Surface Pro. The one with 128GB of storage. He ordered both the Touch Cover and the Type Cover. And some strange, Italian design influenced mouse that looks like a hunk of cheese. And when it finally came, he installed Microsoft Office, Visual Studio, the WDK (the one for Win7 **and** the one for Win8), WinDbg, VMWare Workstation, and even the fussy little application that we use for accessing the OSR VPN. It all installed without a problem. Why shouldn't it, right? The Surface Pro is running Windows 8, after all.

What's even **scarier** is that not only did all that stuff install, but **it ran really well**. In fact, I'd label the Surface Pro a little speed demon. Outlook fired-up and connected to our Exchange server without a hitch. He built a driver in Visual Studio using the Windows 8 WDK without any problems. And so it came to be that on his next trip, the Surface Pro owner left his corpulent (1/2 inch thick), excessively weighty (3.3 pounds) Lenovo laptop at home and just took (a) his iPad Mini, and (b) his Surface Pro. After carrying it around for a day, all he could say was "I'm pretty obsessed with it already, not sure how I'd ever go back."

That's not to say the Surface Pro doesn't have a few big ugly warts. He reports that the Bluetooth stack sucks, just like every other Bluetooth stack on the planet (WTF is it about Bluetooth that makes it suck so badly, huh? It doesn't work right in my car. It doesn't work right on my phone. And it sure doesn't work right on the Surface Pro, if "working right" means you don't regularly lose connectivity to your mouse… but I digress). The screen occasionally and randomly becomes unresponsive, requiring a reboot. And he discovered, as did I when using the Surface RT, that whoever decided that swiping in from the right side of the screen to bring up the charms is a good idea should be publically flogged. You try to turn the page of a document and ooops… a little over-swipe brings up the charms instead of your turned page. Arrrgh. Would whoever was responsible for approving this gesture please meet me for lunch over in The Commons so we can discuss how annoying it is? Thanks.

There are a couple of other issues as well. Where the WWAN? C'mon… even my Kindle talks on 3G. How can you not build this into this type of device? And, of course, there's the whole battery life thing. 4 hours of battery life? Really? Well, that wouldn't even come close to being an iPad replacement now, would it.

About the same time as the arrival of my colleague's Surface Pro, I started looking for a new laptop. One that would replace my gargantuan Lenovo X301 and that also had a touch screen to optimize its use with Windows 8. When I travel, I really only need a computing device to do three things: (a) keep me from losing my mind while in the boarding area of the plane and while on the plane, (b) project PowerPoint slides when I get where I'm going, and (c) maybe let me read and reply to an email that's too big to comfortably digest on my mobile phone. In addition, if the device can let me remotely access my desktop machine to do other stuff (look at a crash dump once or twice a year, for example) I'll be in heaven.

At this point, nothing's going to replace my iPad for keeping me amused while I travel, except maybe another iPad. I learned my lesson well with my Surface RT experience. I read books on my iPad, I play games on it, I surf the web with it. It lasts more than

(CONTINUED ON PAGE 27)

## Peter Pontificates (Cont.)

(CONTINUED FROM PAGE 26)

10 hours on one battery charge. It works in Delhi just as well as it works in Seattle. Basically, I'll give up my iPad when they pry my cold dead fingers from around it.

So, that leaves to the laptop the jobs of projecting PowerPoint slides and sometimes doing email. And sometimes maybe, rarely, perhaps accessing my desktop machine. For these things I definitely need a keyboard. Pecking with my fat fingers at a flat glass screen does not equal actual typing in my book.

In my search for a new lightweight replacement I looked at Lenovos. I looked at Dells. Heck, I even looked at Chromebooks. These devices all failed to meet my criteria in one or more ways. Most are fat and heavy. You'd think something with as little stuff in it as a Chromebook would weigh, I don't know, maybe 6 ounces or something. But no. All but one of the Chromebooks weigh more than 3 pounds. Dell has a cute tablet… but it only offers a Bluetooth keyboard. So now I have to get the tablet, get a case for it, get a keyboard, and get a case for the keyboard. Wait! That's more work than carrying the Thinkpad I already have. The Lenovo tablet is cute, but it has exactly the same problem. In that case, the Bluetooth keyboard (which is rigid plastic) weighs as much as the tablet!

At this point, I started thinking… maybe that Surface Pro would be a good idea after all. Maybe it would work as a replacement for my **laptop**, not my iPad. Sure the Surface RT sucked. But it sucked because it was neither an iPad replacement nor a laptop replacement.

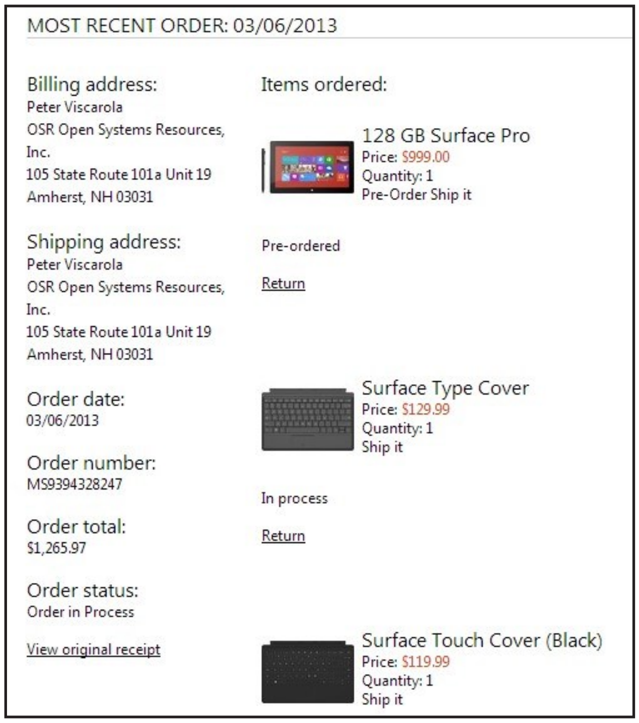
But the Surface Pro? Well…. As I said earlier, the Surface Pro runs office. If I need it to project PowerPoint slides, I could keep it plugged-in, so the battery life wouldn't be an issue. The Pro runs our fussy little VPN client, and therefore not only will Outlook be able to connect to our Exchange Server, but I could use remote desktop to access my office dev box if I needed to. It's got an integral cover with a keyboard with a touchpad (of course you have to pay extra for it, but you know… it's not my personal money that I'm spending). And it weighs less than 2.5 pounds **including the keyboard and cover**.

I searched and searched, but I could not find any device with similar capabilities at similar weight. When viewed as a laptop replacement, and not an iPad replacement, the damn kickstand and the integrated type and/or touch covers are all nothing short of brilliant.

In fact, I think Microsoft may be on to something. Just like Apple invented a new device category with the iPad, I don't think it's going too far at all to suggest that Microsoft has invented a new device category with the Surface Pro. It's a Super Ultra-Portable Laptop Replacement. It kicks the ass of any Ultrabook I've seen in terms of weight and portability. And so far the OEMs seem to be missing the boat with their tablets that need cumbersome add-on Bluetooth keyboards.

Good work, Microsoft. Way to go.

So I ordered a Surface Pro (see above). It just arrived. Wish me luck. After playing with it for a couple of months, I'll tell you how I like it.



MOST RECENT ORDER: 03/06/2013

Billing address:
Peter Viscarola
OSR Open Systems Resources, Inc.
105 State Route 101a Unit 19
Amherst, NH 03031

Shipping address:
Peter Viscarola
OSR Open Systems Resources, Inc.
105 State Route 101a Unit 19
Amherst, NH 03031

Order date:
03/06/2013

Order number:
MS9394328247

Order total:
$1,265.97

Order status:
Order in Process

View original receipt

Items ordered:

128 GB Surface Pro
Price: $999.00
Quantity: 1
Pre-Order Ship it

Pre-ordered

Return

Surface Type Cover
Price: $129.99
Quantity: 1
Ship it

In process

Return

Surface Touch Cover (Black)
Price: $119.99
Quantity: 1
Ship it

**See, OSR actually had to pay for it...**

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

## OSR Seminar Schedule

| Seminar | Dates | Location |
|---|---|---|
| Kernel Debugging & Crash Analysis | 25-29 March 2013 | Boston/Waltham, MA |
| Writing WDF Drivers | 22-26 April | Santa Clara, CA |
| Developing File Systems | 23-26 April | Boston/Waltham, MA |
| Internals & Software Drivers | 5-9 August | Santa Clara, CA |
| Kernel Debugging & Crash Analysis | 9-13 September | Santa Clara, CA |

## OSR Seminars
### We "Practice What We Teach" For a Reason

When we say "we practice what we teach", this mantra directly translates into the value we bring to our seminars. But don't take our word for it...this is what a recent survey of attendees of OSR's *Windows Internals & Software Drivers* seminar had to say:

- [Instructor] was simply awesome. I'd be eager to take any other seminar he teaches.
- [Instructor] is extremely knowledgeable regarding Windows internals. He has the communications skills to provide an informative, in-depth seminar with just the right amount of entertainment value.
- [Instructor] was amazing. He really knows his stuff and was able to handle off-topic questions very well.
- [Instructor] was great; very clear he knows the subject and even more importantly he can teach it.
- [Instructor] is an awesome instructor for these seminars. The venue was absolutely fantastic! The conference room where the seminar was held was less than 100 steps from my room. It was great not to have to drive there and back each day. The conference center staff seemed very eager to please. I really can't think of anything bad to say.
- I learned a lot of useful information that will get me working productively a lot sooner than if I had to pick this info up on the job. I also learned a lot that will be useful in the future.
- The course was very good and I will highly recommend OSR to colleagues in the future.
- It was a great course. I enjoyed the experience and learned a lot.
- One of the better training programs I have attended.

## Private Training

A private, on-site seminar format allows you to:

• **Get project specific questions answered.** OSR instructors have the expertise to help your group solve your toughest roadblocks.

• **Customize your seminar.** We know Windows drivers and file systems; take advantage of it. Customize your seminar to fit your group's specific needs.

• **Focus on specific topics.** Spend extra time on topics you really need and less time on topics you already know.

• **Provide an ideal experience.** For groups working on a project or looking to increase their knowledge of a particular topic, OSR's customized on-site seminars are ideal.

• **Save money.** The quote you receive from OSR includes everything you need. There are never additional charges for materials, shipping, or instructor travel.

• **Save more money.** Bringing OSR on-site to teach a seminar costs much less then sending several people to a public class. And you're not paying for your valuable developers to travel.

• **Save time.** Less time out of the office for developers is a good thing.

• **Save hassles.** If you don't have space or lab equipment available, no worries. An OSR seminar consultant can help make arrangements for you.

®