The NT Insider

The only publication dedicated entirely to Windows® system software development

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

September—October 2011 Digital Edition Volume 18 Issue 3

Epic Update: Win8 WDK Provides Visual Studio Integration

In case you haven't been paying attention to what's been going on in preparation for the next version of Windows, back on 13 September Microsoft held the long-awaited "Build" developer conference. As promised the conference gave the general public its first glimpse of Windows 8, dubbed the "Windows Developer Preview." For most folks who work in the world of Windows this was pretty big news.

What was even bigger news for those of us involved in writing Windows drivers was the introduction of the next version of the Windows Driver Kit (WDK). For the past many years, we've grown to expect a WDK that looks pretty much like the previous WDK. Maybe there are a few more driver models. Perhaps there are some more DDIs added to KMDF and UMDF. And we always expect the documentation to continue its slow progression from useable to useful.

However, if you were expecting "more of the same" for this release of the WDK you were in for a very, very, big – enormous – surprise. The new WDK is nothing like the old one. In fact, the new WDK incorporates *the number one feature* requested by the driver development community over the past ten years: The Windows 8 WDK has been changed to be fully integrated with Visual Studio.

No, I'm not kidding. Seriously. We've finally gotten full Visual Studio integration. And it's not just simple "invoke a command procedure as an external build step" integration, either. What we got was actual, real, integration including choices for different driver starter projects, built-in PREfast, the ability to fire-up SDV to run (asynchronously, thank goodness) from within the VS IDE, and even integration with the kernel debugger. And, if that's not enough, the Win8 WDK also includes support for automated deployment of your driver. If enabled, when you hit F5 your driver is rebuilt (if it's out of date), copied to the test system you've previously indicated, installed on that test machine, and started. You can choose to automatically enable Driver Verifier if you want. You can even configure a set of tests to start automatically.

Perhaps that's not enough to excite you. How about automatically signing your driver for you, if you choose? Yup, you can select from test signing or production signing. If you choose test signing you can optionally have a test certificate automatically generated for you, or you can choose an existing test cert. You can even choose a timestamp server to use during the signing process, without having to remember the URL (the paths to the Verisign and Globalsign time stamp servers are built in). And, yes… the WDK comes

(Continued on page 22)

Windows Developer Preview: Where to Get It

With the Build conference behind us finally and the Windows Developer Preview (i.e., Windows 8) now available, let's take a moment to make sure you all know where to go for it, and associated kits and tools.

You can access the Win8 Developer Preview at the Windows Dev Center—Hardware: http://msdn.microsoft.com/en-us/windows/hardware

In addition to the Win8 preview, you can gain access to the WDK, the Visual Studio 11 Developer Preview, and other associated test and deployment kits via MSDN Downloads (for MSDN subscribers).

Note also that there have been some great discussions about Win8 and the integration of the WDK and Visual Studio in the NTDEV newsgroup: http://www.osronline.com/cf.cfm?PageURL=showlists.cfm?list=NTDEV

The NT Insider™

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

http://www.osr.com

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

Single Issue Price: $15.00

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

Stuff Our Lawyers Make Us Say

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

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

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

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

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

Inside This Issue:

OSR Page: Of ARM, SoC and "Big Windows"

The movement of mobile devices (e.g., tablets, etc.) over to the Windows space has made for interesting work for the team here at OSR as of late. After all, it's not often we get to become involved in bringing large teams of architects and developers over to the world of Windows.

Surely, much of this is due to the overall interest in the mobile market, but it's hard to say that the single biggest impetus for this isn't the promise of Windows—and by "Windows" I don't mean, WinCE, Windows Mobile/Phone7 — increasing its breadth of support for mobile devices. I mean (as it was first described to us by a client) "big Windows".

Future support in Windows 8 for ARM and SoC based systems is, frankly, exciting. But it's challenging work. Whether we're working with ARM and SoC vendors, or device or solution providers further down the line that need to support ARM or SoC on Windows, we've found the experience enlightening.

You have, on one hand, the challenges of breaking existing mindsets. After all, Windows is, um, "different", from WinCE/Mobile/Phone7.

In addition, we are finding that (go figure), the devs from an SoC or ARM background that come to us to learn about Windows architecture are keenly focused on implementation details when it comes to support for *their specific device* on Windows.

On the other hand, this work is keeping us on our toes. The folks we've had the pleasure to teach and consult with are very much "non null", and bring a great deal of experience and expertise from "their world" to our work with them. This makes for some pretty cool discussions.

So hey, if you're in the mobile space and are interested in working with OSR to help move your team of architects/developers over to "big Windows", we look forward to hearing from you!

Peer Help?

Writing and debugging Windows system software isn't easy. Sometimes, connecting with the right people at the right time can make the difference. You'll find them on the NTDEV/NTFSD/WINDBG lists hosted at OSR Online (www.osronline.com)

The NT Insider—Digital Edition

If you are new to The NT Insider (as in, the link to this issue was forwarded to you), you can subscribe at:

http://www.osronline.com/custom.cfm?name=login__joinok.cfm.

Peter Pontificates:

Do Christmas Dreams Come True?

Back in November 2009, I wrote a Christmas list describing my hopes and dreams for the future of Windows driver development (see *What I Want for Christmas (2011)*). Since the release of the Windows Developer Preview, a number of people have asked me how the Win8 WDK stacks up against that list. So, let's see!

The first and most important item on my Christmas wish list was a driver development environment that was integrated with Visual Studio. I wrote (in part):

> *While the VS editor might suck ass (might?) there are just too [many] useful things in VS to ignore it any longer. Too many add-ins. Too many features. So, I want a driver development environment that's both supported by Microsoft and fully integrated with the VS IDE. This means I want VS to be able to successfully parse ntifs.h (and his include files), reasonably handle the conditionals therein, and provide me IntelliSense, F1 for help, and all the other VS features and wonderments. I want the various VS add-ins that provide cool features such as smart refactoring (including the ability to rename variables in a project… man, do you have any idea how great that is) to work properly.*
>
> *I need the VS IDE to front-end the kernel debugger, preserving all its features, so I don't have to suffer the slings and arrows of WinDbg's arbitrary behavior.*

It seems that this wish, my first and most important wish, has come true. We finally have a WDK that is fully integrated with Visual Studio. And while there's certainly room for a bit of smoothing out of that integration, what we've seen in the Win8 WDK and VS11 is pretty much the core of what I asked for.

Note, too, that the new VS-integrated WDK actually does more than what I asked for on my list. Not only are the driver coding and debugging processes integrated, but so are the signing, deployment, and testing processes built into this same environment. This is a level of integration that I, quite frankly, didn't even think to ask for. So, kudos to the WDK team on reaching beyond what I had hoped.

But I didn't only ask for a new, VS integrated, WDK build environment, I also wished for backward compatibility with BUILD. I wrote:

> *I need to be able to build driver projects seamlessly in either VS or using "build" without any inconvenience. I need to be able to seamlessly "round trip" between VS and traditional build, so that any files I add to a "sources" file are easily (or, better yet, automatically) included in my VS project, and any files that I add to my VS project easily (or, better yet, automatically) show up in my sources file.*

This is a wish that has not come true. And that's a darn shame. While we understand that we all face resource constraints, I think this is a feature that's far too important to be "below the line" and not implemented.

Why is it important to be able to "round trip" between the old build environment and the new VS integrated environment? One reason is that this would allow us to support "down level" build environments much more easily. In case you didn't notice, the Win8 WDK does not support building drivers that are targeted for Windows XP. If you're like us here at OSR, you still need to support Windows XP and probably Windows 2000 as well. This means that you'll have to manage and support two entirely separate build environments that use

(Continued on page 5)

Peter Pontificates...

(Continued from page 4)

entirely separate sets of project metadata. Which totally and completely sucks. When you add a file to your project, change the compiler flags, or define some new preprocessor value, you'll need to do this in two separate places: You'll need to do it in VS for your new Vista and later build environment and you'll need to do it in your sources file for your XP build environment.

However, having to maintain two entirely separate build environments is not the only reason that not having the ability to "round trip" between VS and sources is important. The most important reason is that the ability to convert from sources to VS and back to sources again allows you to *check that your driver project was properly converted*. Unless you happen to be a Visual Studio and/or MSBUILD expert, when you import or convert your driver project into the Win8 WDK environment, you'll have no clue if the import was done correctly. Sure, you'll be able to tell if the resultant project builds and runs. But that doesn't necessarily mean all the little settings you painstakingly established in your sources file have been properly brought forward. Given that you do understand the build/sources system, if you had the ability to convert your project to VS and then back again, you'd at least be able to check to see if the project is the same as the one you converted.

So, in summary… for my first and most important Christmas wish for Visual Studio integration, I got about 85% of what I asked for. Plus, I got a bunch of stuff (like automated signing, deployment, and testing) that I didn't even think to ask for. Darn good work, by anybody's standards.

The next thing I asked for was DMA virtualization. I wrote:

> *I want to see DMA scribbles eliminated for all time. IOMMUs are here, and I want Windows to use them. For all DMA operations. This would also mean that I can stop writing articles about why people need to use Map Registers and not just call MmGetPhysicalAddress; and you, dear reader, would be saved from having to read those articles.*

I still think this is a laudable goal. Unfortunately, I'm not aware of any details that have been made public about DMA virtualization in Win8.

So we'll have to mark this particular wish, DMA Virtualization, as "wait and see."

The third item of which I dreamed was a more advanced, cleaner, architecturally more complete and rigorous KMDF. I wrote (again, in part):

> *What we have from KMDF today is not enough. I want the next major revision of KMDF [with] aggressive, continued, forward architecture and development.*
>
> *I want the rough edges smoothed out. I want somebody with a solid knowledge of driver development, and a broad architectural view, to spend serious time reviewing KMDF with the goal of making the level of abstraction consistent, and on average, just a bit higher than it is today. [And] I want KMDF to include a new set of features and extensions that ease common programming patterns.*

My wish list actually described several specific things that I hoped would be changed, enhanced, and smoothed out in KMDF.

When I wrote this, I quite frankly didn't expect that I'd get it. And my expectations were met. There are a few very nice tweaks to KMDF in this new release of Windows, but in general, KMDF remains what it was. I'm sure I know why this is: The KMDF/UMDF group was plenty busy adding other features. But, as much as I like KMDF – and as I said back in the original article, I *am* a major KMDF fan – KMDF remains a good, but sometimes disappointingly ragged and incomplete, framework. The most coherent and compelling part of KMDF is its elegantly simple, integrated implementation of PnP and Power Management. And while this alone makes KMDF worth using, there is so much more that KMDF could be. The abstractions could be so much more coherent and clear. The ease-of-use could be so much greater. And I don't think it would take that much work to elevate KMDF from "good" to "really really great" – I just wish somebody with adequate time and vision would step-up and get this done. Maybe what Microsoft needs to do is hire OSR to sketch out a vision for this. Give me a call anytime, guys. You have my number :-)

A dynamically advanced and cleaned-up KMDF? Not in Win8. So this wish remains unfulfilled.

Another item that was on my Christmas list that I didn't get, and that I'm not surprised at, was a new implementation of UMDF. My original request was:

> *I would like the ability to take my KMDF driver unchanged and compile it for use in either user mode or kernel mode. I very strongly believe that having entirely different programming models for user-mode and kernel-mode unnecessarily complicates WDF and actively hinders the movement of drivers out of the OS.*

But, as far as I know, there's been no work done in this area at all for Win8.

So, in terms of our getting an integrated KMDF and UMDF, the answer for Win8 is no.

(Continued on page 29)

WDK Preview:

Installation Through Debugging

If you haven't yet had the opportunity to install the WDK from the Windows Developer Preview, we thought we'd provide you some guidance on installing the WDK and configuring a target machine for network debugging. Hopefully, when you decide to give it a spin, this guide will get you up and running a bit more quickly.

Install the Win8 Dev Preview on Target

The first step is to get Win8 installed on your target machine. For our testing, we've used a mix of real systems and virtual machines. For the writing of this article, VMWare Workstation 8 was used for both the host machine and the target machine.

To properly prepare your target machine, make sure that you create a local administrator account with a non-blank password. More importantly, once the installation is complete be sure to disable UAC! Failure to do so will lead to access denied errors on the host when trying to remotely configure the target.

Configure Host Machine

You can choose any host system that you like, though for this article we're using a 32-bit install of Win8. Once you've chosen your host, you need to install VS2011 and the WDK, both of which are available through the MSDN Subscriber Downloads. Unlike on the target machine, it is very important that you do *not* disable UAC on the host. Currently, network debug requires that the VS debugger application run without being elevated. If UAC is disabled on the host, there will be no easy way to guarantee this and will therefore result in strange errors when it comes time to debug.

We strongly urge you to install the *32-bit build* of VS2011. We were unsuccessful in our attempts to get the WDK properly integrated with the x64 build of VS2011, so don't waste your time. As long as you're using the 32-bit build of VS2011, installation of the WDK should be painless. Simply run \wdksetup.exe from the downloaded package and you're on your way.

As part of your host system configuration, it is strongly recommended that you assign the host a static IP. This is solely because we'll be using the network as our debug protocol and, in doing so, we'll be providing the host machine's IP as part of the debug settings. Failure to use a static IP may result in unexpected loss of the debug connection if the host's IP changes at any point.

Configure Target through VS2011 on Host

For this article, we'll forego using any project and just worry about getting VS to configure and connect to the target machine.

Even though we've explicitly told you in this article to leave UAC enabled on the host, in order to remotely configure the target machine you will need to run VS elevated on the host. This is a known restriction in the kit that may be addressed in the future and will not interfere with your ability to do network kernel debugging.

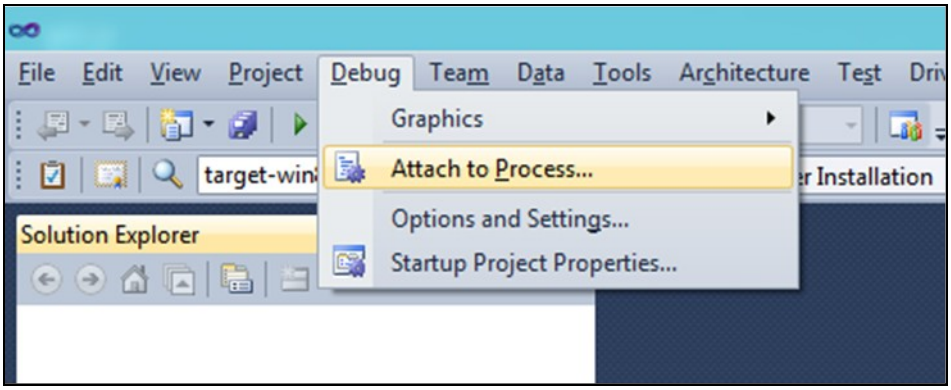


Figure 1

Once you've launched an elevated instance of VS, instruct it to attach to a process via the Debug->Attach to Process menu item, as seen in *Figure 1*.

This will bring up the process attachment dialog which, if you've done everything correctly up to this point, should have a couple of new debuggers to choose from. We'll be using the Windows Kernel Mode Debugger, as seen in *Figure 2*.

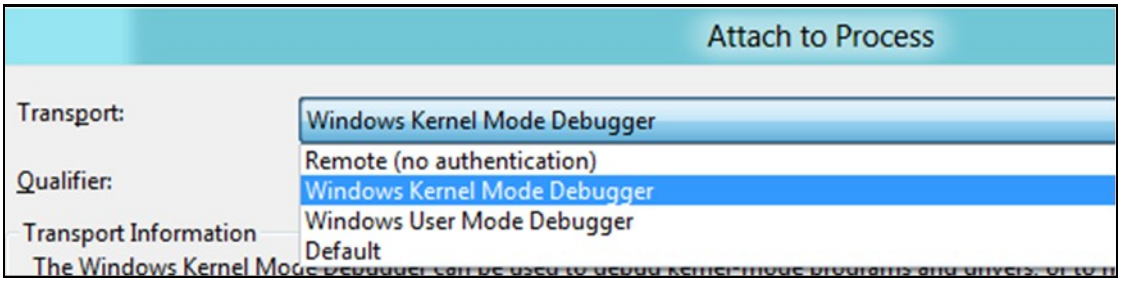


Figure 2

(Continued on page 7)

WDK Preview...

(Continued from page 6)

Next, we need to choose which machine we want to kernel debug. This is done via the Qualifier field of the Attach to Process dialog. This should be blank at this point, so we'll click the Find button to locate a computer to configure (as highlighted in *Figure 3*).

Doing so will bring up the Configure Computers dialog, as shown in *Figure 4*. This dialog will be empty due to the fact that you have yet to configure any computers. Therefore, the next step is to add a computer, which can be done by clicking the Add button in the Configure Computers dialog.

Computers are configured for debugging by their network name, thus you must be able to properly resolve your target computer by name. In our case, we've cleverly named the target machine *win8-target*. Therefore, that's the name that we will provide for the Computer Name field in the dialog. The result of this can be seen in *Figure 5*.

In *Figure 5*, you can see that VS has defaulted to using the network as the debug protocol. It's also chosen a port number for us and even generated an encryption key to be used to

(Continued on page 24)

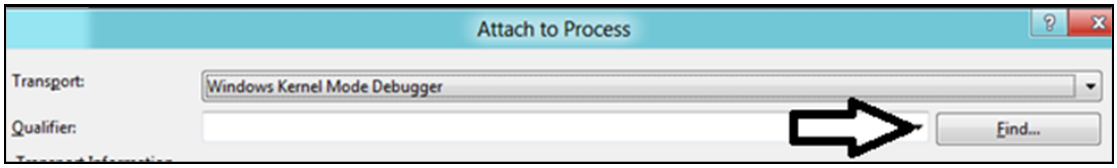


Figure 3

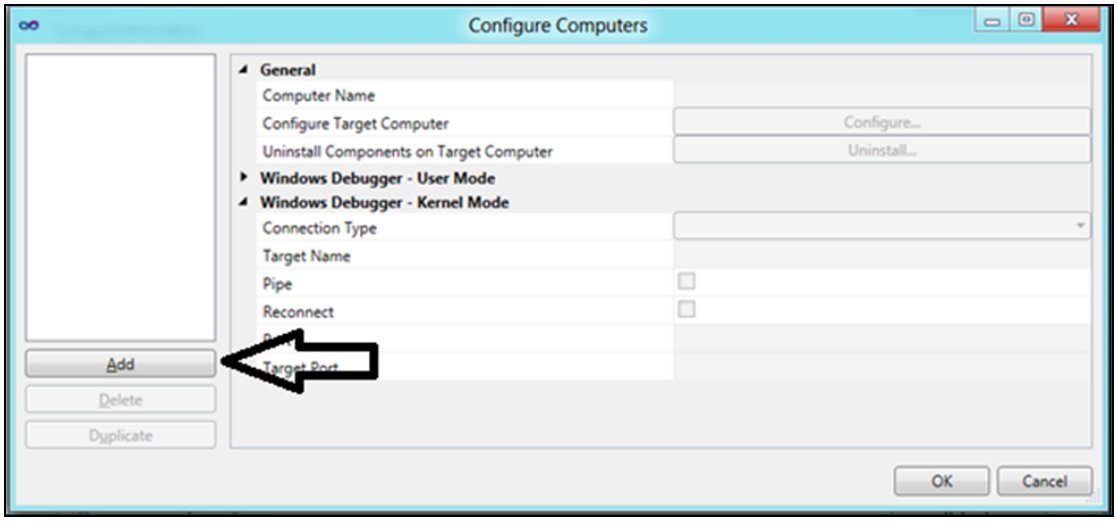


Figure 4

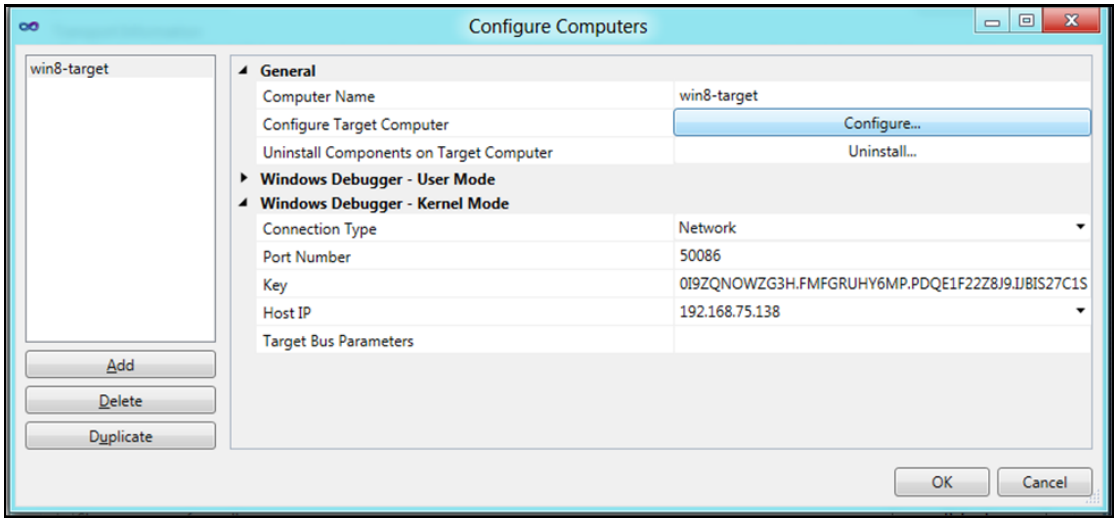


Figure 5

But Wait...There's More!

Win8 and WDK Changes You'll Care About

While the big news might be that the Win8 WDK brings Visual Studio integration to the driver world, there are *many* more interesting things that were described at Build about both Windows 8 and the Win8 WDK. In this article, we'll try to summarize a few of them that we found most interesting. See http://dev.windows.com for a link to videos of all the Build talks, as well as to the current WDK documentation.

New Debugging Connections

Win8 brings two new, interesting, debugging options. The first is network-based debugging. You can connect your host (development) machine running WinDbg (or the Visual Studio debugger) to your target (test) machine via the network. According to what we've been told, this will work across most network cards "built within the last several years." We've tested it here at OSR. It does work, and it's fast, too.

The second new option for debugging connections is USB 3. The only "gotchas" here are that you'll need a slightly unusual cable to connect between two hosts and you'll need a USB implementation that supports debugging. The good news is that all the ports on the USB 3 XHCI controller will support debugging, so you won't have to search for *the* single, specific, mystical (or mythical) port that supports debugging the way you do for USB 2 today.

Drivers in C++

If you're a C++ fan, you be pleased to know that C++ will be supported "as a first class citizen" in the new WDK. This is due to compiler and linker changes that enable things like placement of class instances in appropriate (non-paged) memory areas. Don't lose your mind, though: You can't use C++ native exception handling or run-time type information. But, for those who love it, this will be a big step forward. For the rest of us, I feel a headache coming on.

Hardware Access in UMDF

The latest version of UMDF includes support for UMDF access to ports and registers, as well as the ability to handle (edge-triggered and MSI) interrupts. You can choose (via an INF file option) to access your device registers directly from your UMDF driver, or do that device access via a system service. And, yes… this means that your little UMDF driver can now have EvtDevicePrepareHardware and EvtDevice ReleaseHardware event processing callbacks, just like its big brother KMDF.

In fact, UMDF is receiving a great deal of emphasis in Win8. Several small changes to UMDF have made it more generally useful, easier to use when working with a kernel-mode driver, and have in general expanded the reasonable usage scenarios.

Low Power Bus Support

As Windows 8 expands to support smaller, lower powered, devices (think tablets and such), the need to provide system-level support for the low-power buses used by these devices becomes extremely important. Windows' classic PnP/Power scheme in which a parent bus provides power, interrupt and data connections to the world (the way PCI does) isn't sufficient to describe the way that many devices are connected in, for example, System On Chip (SoC) based systems. Win8 introduces system-wide support of low-powered buses included I2C, SPI, GPIO, and even high speed embedded UARTs.

(Continued on page 9)

But Wait...There's More...

(Continued from page 8)

In Win8, the topology of these buses and the devices that are connected to these buses are described using ACPI 5.0 ASL. This description can also include connections to multiple buses, such as in the case where a SPI device generates an interrupt using a GPIO pin. The platform vendor provides the driver for the (I2C, SPI, and/or GPIO) buses. Then, OEMs/IHVs write client drivers that talk (in a controller and even platform independent manner) to devices on those buses. A new Windows component called the Resource Hub connects client drivers to controller resources. You should expect to see an expanded use of these low power buses in Windows, in both SoC and even in non-SoC bases systems starting in Windows 8.

Co-Installers and Support for Older WDF Versions

The new model for WDF will be to move away from co-installers. YAY! This change makes lots of things (like, er, the installation of certain filter drivers) much easier. The new plan is apparently to push new versions of the Frameworks via Windows Update. This means client machines may or may not be updated, and if they are updated they can be updated on the client's timeframe, in the same way that updates of the .Net Framework are handled today.

Another new feature that goes along with the above: Starting with the Win8 WDK, it's now possible to use the latest WDK to build against an older version of the Framework. So, for example, you could choose to build your driver against KMDF 1.9 (which was released with Windows 7) instead of against KMDF V1.11 (which is the version that will be released with Windows 8).

Power and Power Framework (PoFx)

There have been several changes to power management that focus on mobile devices, such as tablets and phones, where power management is critical.

One of the biggest changes to power in Win8 is the support for D3-Cold state. Prior to Win8, devices that were in D3 were always in D3-Hot. This meant that while the devices were in a very low power state, these devices were never 100% powered off and that they remained enumerable on their bus. In Win8, support for D3-Cold has been introduced to enable maximal power savings.

Win8 also introduces support for a new concept called "functional power states" or F-States, via the Power Framework (PoFx). F-States, which are defined by a driver, facilitate rapid transitions between higher and lower powered states. PoFx also facilitates support of power management on multi-function devices, where power to individual functions of the device can be separately changed.

System DMA

You thought it was dead, didn't you. Well, it's baaaack! Both WDM and KMDF have been expanded to add support for new, SoC-style, system DMA. In system DMA, multiple components share a single DMA controller, as opposed to bus master DMA where the device itself contains sufficient logic to master the bus and autonomously perform transfers between itself and memory.

Improved Documentation

With this release of the WDK, even though it's early days, the WDK documentation has continued its steady evolution. To my eyes, there's been a lot of work done and the docs are better than ever. The new layout is crisp, clean, and clear. Most, if not all, functions now include the "Requirements" section which quickly shows the earliest OS version in which a given DDI is supported, the header in which the function is declared, and even the IRQL at which the function can be called. You'll find many of the newer functions documented already, which is a surprise and a treat. And, what may be the biggest surprise of all, I actually found several of the example code segments provided helpful. Gasp! What next!?!

And The Rest

There are numerous other changes in Win8 that might be of interest to you, depending on the area in which you work. The reader/writer spinlocks that were first introduced in Windows 7 are finally documented (see ExAcquireSpinLockExclusive and friends). Directory oplocks have been introduced. A couple of dozen new KMDF and more than 2 dozen or more new UMDF functions have been provided. New support for certain WiFi functions has been added to NDIS. Storport has had considerable enhancements, including a whole new meaning for the term SRB.

These are just a few of the changes you'll find in Win8. We can't enumerate them all. Of course, you can count on us to write more about these – and more – in future issues of *The NT Insider*.

Five Things to Like:

Visual Studio Integration

Change is hard. When I first heard that the new WDK would be integrated with Visual Studio, I, just like a lot of people, pitched a fit. In my mind, Visual Studio is too bloated and complicated. I already have a lightweight editor (SlickEdit), a lightweight debugger (WinDBG), and a build system that makes sense to me. Why would I want to throw this all away just to have some hideous IDE crammed down my throat?

To prove a point (to no one but myself, really), I ran a test to see how many of my development environments I could bring up before VS even launched. Between the time I clicked the VS icon and when I was presented with the Start Page, I was able to launch:

- 12 Build Environment Windows
- 17 WinDBG Instances
- An entire project consisting of four drivers

Clearly my current environment is more streamlined and VS is a pig, right?

Well, turns out, yes and no. When I finally decided to put my hubris away for a little bit and, *gasp!*, actually *try* to find some good in the change, I discovered that there's quite a bit to like here. There's definitely going to be some growing pains and not absolutely everything is great, but here's a list of five things that I had to tip my hat to.

1. Configuring Symbols via the GUI

Remember how annoying it was to have to remember the full path to the Microsoft Symbol Server? Probably not, because *.symfix* has alleviated us of that need for quite a while now. However, keeping your symbol search path correct across multiple WinDBG workspaces is always a pain. Also, trying to parse that long, wrapping, semicolon delimited symbol search string when something isn't working is never fun.

Given that, it was with great joy that I found the Symbols editor in the VS Debug->Options and Settings dialog, shown in *Figure 1*.

The GUI makes it exceptionally clear where the debugger is looking for symbols and you can even quickly edit the priority of the locations by moving them with the up and down arrows.

As an added bonus, you can even exclude or include the loading of symbols for specific modules. This is great for avoiding the time spent going over the network for symbols that will never be on the Microsoft Symbol Server, such as

(Continued on page 26)

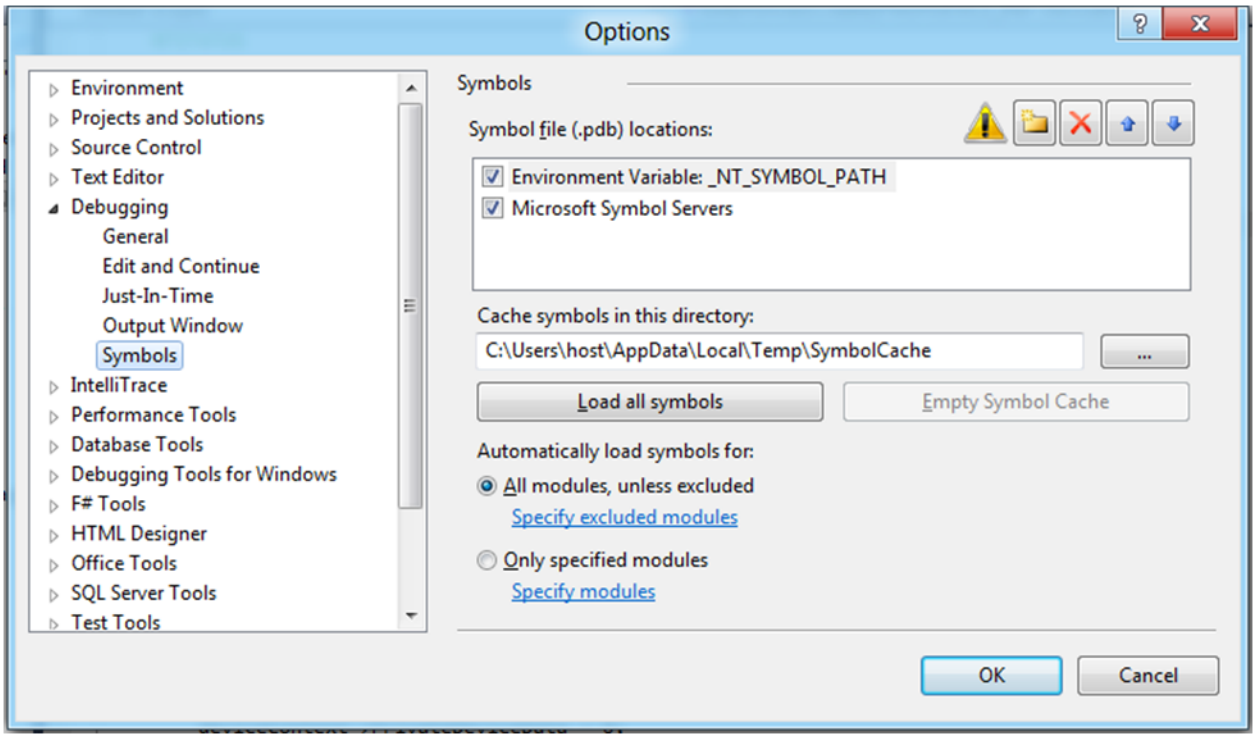


Figure 1

Five Things to Not Like:

Visual Studio Integration

To complement the other article in this issue, *Five Things to Like About Visual Studio Integration*, we decided it would also be good to take a more critical view of the new environment. In doing so, we've put together a list of five things that we're not so excited about with the new WDK.

While putting together this list, we've tried to avoid things that are general to VS and not specific to the WDK integration. So, please don't take this article to mean that we actually like the VS editor or that we couldn't come up with a zillion things that we didn't like about it. I mean, is there seriously no way to add a column marker without a crazy registry hack that no longer appears to work?

1. What Works for Drivers and What Doesn't?

While the WDK integration adds a few new driver-specific menu items to VS, it currently doesn't *remove* any menu items that aren't applicable to drivers. This quickly leads to frustration as you'll find yourself getting excited for some new tool that's going to improve your driver but, alas, isn't supported.

Code Metrics? Not for drivers. Code Coverage? Nope. Error Lookup?? Doesn't support NTSTATUS values. Right now it doesn't appear as though we'll be getting any of VS' modern development features supported for drivers, though I'm really hoping that changes. If not, why bother having all of this stuff cluttering up my menus?

2. No Support for Building XP Drivers

It's been argued to death on NTDEV, but it's worth mentioning again: no support for XP as a target build platform. With XP not at End of Life until 2014, most of us are going to be stuck supporting XP targets for years to come. Without official support for XP, this could possibly mean maintaining two entirely different build environments or holding off on adopting the new build environment.

Here at OSR we're planning on supporting two different build environments for our kits and drivers. While changes to the build files are typically rare in an existing project, this does increase our test matrix as we're not going to ship what we don't test.

3. It's WinDBG in Visual Studio! Oh, wait…

After running a few of your favorite WinDBG commands, you may be lulled into thinking that you are, in fact, still in WinDBG. You have to remember though that this is VS with a KD plugin, so things don't exactly map one-to-one.

For example, VS currently doesn't appear to support the Debugger Markup Language (DML). Hopefully this will be fixed in the future because I've gotten quite used to clicking *! analyze –v* when the system crashes, and we have internal tools here that rely heavily on DML. Another thing that I'm missing already is the ability to right click text to send it to the clipboard and then right click again to paste it to the command line. I never quite realized how often I used this until I started debugging an issue inside VS.

4. Visual Studio is SO a User Mode IDE

As driver writers, we live in a different world than application developers. Take, for example, a case where you're working on some new code and want to step through it to make sure you have your logic right (please don't send us flame mail about whether or not "good" developers need to step through code!). If you're developing a user application, you add some

(Continued on page 28)

Windows 8 Preview: File System Changes

With the recent release of the Windows Developer Preview (i.e., Win8), I took the opportunity to review the changes that they've made public that will affect file system and file system filter driver developers for the Windows 8 Platform.

While one can certainly look at the documentation, my approach in preparing this article was to look at the changes in the critical header files: ntifs.h and fltKernel.h, as these suggest much of the current scope of changes.

With that said, my experience with previous pre-release versions of Windows is that *everything is subject to change.* Thus, it is unwise to count on any feature or change that we describe in this article until the final release of Windows 8.

New File Systems

The header files suggest the presence of new file systems as part of the Windows 8 release. This includes the Protogon file system (although there is no new Protogon file system that I could see in the Developer Preview). The headers indicate:

```
#define FILESYSTEM_STATISTICS_TYPE_PROTOGON 4
typedef struct _PROTOGON_STATISTICS {
    ULONG Unused;
} PROTOGON_STATISTICS, *PPROTOGON_STATISTICS;
```

Speculation in the press is that this will be the database-as-filesystem model previously promoted as WinFS. If so, I suspect it will be a substantially different implementation (likely with a strong emphasis on performance). Then again, if it wasn't far enough along to include in the Developer Preview, it might be one of those features that doesn't make the initial release.

In addition, there is quite a bit of new information about something called CSVFS (perhaps the "cluster shared volume file system" to work with "cluster shared volumes"? If so, this may just be exposing new information to an existing technology). Again, it seems a bit premature to speculate, especially given that if it is related to clustering, it is not likely that this feature would be present in the client release.

Emphasis on Data Verification

Many of the changes we observed relate to data verification support; whether this is an NTFS only feature, or if it will be present in other file systems (e.g., Protogon). There are two new FSCTLs (**FSCTL_{GET,SET}_INTEGRITY_INFORMATION**) that appears to be related to these changes, and a structure used to control this information (as shown in *Figure 1*).

Frankly, this makes quite a bit of sense: modern disk drive technologies are vastly more complicated than they have been in the past and can (and do) suffer from real issues with respect to data integrity. Thus, it is quite possible to read data back from the drive and receive something other than what was originally written. While it doesn't happen frequently, it *does* happen and most applications are not written to withstand that sort of data corruption. Some applications, particularly database applications, are highly susceptible to this type of data corruption – the relationship information between the components can be lost when critical database information is lost. While transactional database models will protect against some forms of failure, they do not protect against incorrect data being returned from the underlying storage media.

```
#define FSCTL_INTEGRITY_FLAG_CHECKSUM_ENFORCEMENT_OFF        (1)

typedef struct _FSCTL_INTEGRITY_INFORMATION_BUFFER {
    USHORT ChecksumAlgorithm;   // Checksum algorithm. e.g. CHECKSUM_TYPE_UNCHANGED,
CHECKSUM_TYPE_NONE, CHECKSUM_TYPE_CRC32
    USHORT Reserved;            // Must be 0
    ULONG Flags;                // FSCTL_INTEGRITY_FLAG_xxx
} FSCTL_INTEGRITY_INFORMATION_BUFFER, *PFSCTL_INTEGRITY_INFORMATION_BUFFER;

And equally interesting, there is the presentation of "integrity streams":

#define FILE_SUPPORTS_INTEGRITY_STREAMS     0x04000000  // winnt
```

Figure 1

Data Deduplication

Another intriguing hint from the header files are suggestions for supporting data deduplication techniques. For example the following new FSCTL operations for data deduplication are present (see *Figure 2*).

(Continued on page 13)

```
// Dedup FSCTLs
// Values 162 - 170 are reserved for Dedup.
//
#if (_WIN32_WINNT >= 0x0602)
#define FSCTL_DEDUP_FILE                    CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 165, METHOD_BUFFERED,
FILE_ANY_ACCESS)
#define FSCTL_DEDUP_QUERY_FILE_HASHES       CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 166, METHOD_NEITHER,
FILE_READ_DATA)
#endif /*_WIN32_WINNT >= 0x0602 */
```

Figure 2

File System Changes...

(Continued from page 12)

Thus, this suggests using a scheme by which file hashes are being queried and data deduplication is being done for those blocks with identical hash values (for good deduplication hash algorithms, the likelihood of a data hash collision is very low).

Offload Support

The header files strongly hint at new support for "offloading" I/O operations. While we are speculating a bit, if this is similar to other forms of offloading, it would suggest the use of hardware to perform some operations (e.g., I/O operations as well as computing hashing). This might be used, for example, for intelligent disk drives to allow them to perform additional high level processing, such as is done in Object Storage Devices. When combined into file systems, such devices can actually provide specialized support and can even split data and meta-data across multiple drives (local and remote). Whether or not that is what is envisioned here is still uncertain (after all, this is just based upon header file information).

There are two new FSCTL operations for offload (*Figure 3)*.

What is particularly important about this new support is that it is *disabled* if file system filter drivers do not support it, according to the comments within the header file itself (*Figure 4*).

Thus, it is important for those building file system filter drivers to be both aware of this new functionality and to ensure that they support it. Otherwise, it runs the potential risk of breaking some other product's functionality (or at least degrading performance) without supporting it.

Advanced FCB Header Changes

File system filter driver writers won't notice this change as much as those of us building file systems, but the Windows 8 version of the advanced FCB header has changed. The Advanced FCB header has gained a new field (**Oplock**) and the definition of **FsRtlSetupAdvancedHeader** (implemented in the header file) has changed to initialize the header properly.

The comment from the header file is fairly clear on this one (see *Figure 5*).

Note that this declares there to be a new FCB header version (2), set in the **Version** field of the common FCB header.

Oplocks

This new oplock field appears to tie into a new round of oplock support changes in Windows 8. For example, we have several new support functions: **FsRtlCheckLockForOplockRequest** and **FsRtlAreThereWaitingFileLocks**. Interestingly, these new support routines have been optimized so that byte range locks on regions of the file outside the allocated range will not conflict with oplocks (recall that byte range locks and oplocks are traditionally inconsistent).

(Continued on page 14)

```
#define FSCTL_OFFLOAD_READ     CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 153, METHOD_BUFFERED, FILE_READ_ACCESS)
#define FSCTL_OFFLOAD_WRITE    CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 154, METHOD_BUFFERED, FILE_WRITE_ACCESS)
```

Figure 3

```
//
//  To better guarentee backwards compatibility for
//  selective new file system functionality, this new
//  functionality will be disabled until all mini
//  file system filters as well as legacy file system
//  filters explicitly opt-in to this new functional-
//  ity.  This is controlled by a new registry key in
//  the filters service defintion called
//  "SupportedFeatures".
//
//  File System filters need to update their .INF
//  files to set state that the given functionality is
//  now supported.  Even if a filter can't actually
//  support the given operations they should mark in
//  the .INF that it is supported and modify their
//  filter to fail the operations they don't support.
//
```

Figure 4

```
//
//  The following fields are valid only if the Version
//  field in the FSRTL_COMMON_FCB_HEADER is greater
//  than or equal to FSRTL_FCB_HEADER_V2.  These
//  fields are present in Windows 8 and beyond.
//
//  For local file system this is the oplock field
//  used by the oplock package to maintain current
//  information about opportunistic locks on this
//  file/directory.
//
//  For remote file systems this field is reserved.


        union {

            OPLOCK Oplock;
            PVOID ReservedForRemote;

        };
    #endif
    #define FSRTL_FCB_HEADER_V2             (0x02)
```

Figure 5

File System Changes...

(Continued from page 13)

Logically, this makes sense. The rationale for not allowing both is that applications using byte range locks want to obtain coherent copies of the data, which is incompatible with the basic premise of caching. However, byte range locks on regions of the file where there is no data are used by applications as a means of inter-process communications (potentially on different computers) and not related to data coherency. Thus, these routines should actually permit the use of oplocks in a broader range of situations.

New ECP Types

Another interesting change related to oplocks is the new support for extra create parameters (ECPs) for tracking oplock "ownership" state for not only the target of an open operation, but also the parent. Note the definition of the new dual oplock key:

```
DEFINE_GUID( GUID_ECP_DUAL_OPLOCK_KEY, 0x41621a14,
0xb08b, 0x4df1, 0xb6, 0x76, 0xa0, 0x5f, 0xfd, 0xf0,
0x1b, 0xea );
```

Presumably, this would be useful for rename operations (for example) and would allow operations to proceed without forcing oplock breaks (much like the GUID_ECP_OPLOCK_KEY was used previously to keep from breaking oplocks with associated opens to the original oplock holder).

Beyond this, we also have a solution to the frustration of receiving back an error whenever a reparse occurs inside a call to **IoCreateFileSpecifyDeviceObjectHint** (or any function that it calls it, including the various Filter Manager APIs for opening files). This error (STATUS_MOUNT_POINT_NOT_RESOLVED) is difficult to resolve cleanly in a file system filter driver. Having done so (by iterating through the path until finding the reparse point, then opening it to query the contents of the reparse information) it is encouraging to find that there will be a solution to this problem for future releases of Windows.

Basically, this ECP information will allow a filter to obtain detailed information about the reparse point:

```
typedef struct _IO_DEVICE_HINT_ECP_CONTEXT {
    PDEVICE_OBJECT TargetDevice;
    UNICODE_STRING RemainingName;
} IO_DEVICE_HINT_ECP_CONTEXT,
*PIO_DEVICE_HINT_ECP_CONTEXT;
```

Indeed, the comment from the header file fairly clearly describes the model for this new ECP:

```
// This GUID and structure are for passing back
// information from the I/O manager to the filter
// manager about a reparse when the reparse target
// goes to a new device.
```

Cache Manager

There are a number of Cache Manager changes present in the Windows 8 header file as well. One that has me a bit mystified is the exposure of a new structure:

```
typedef struct _READ_AHEAD_PARAMETERS {

    CSHORT NodeByteSize;

    //
    //  Granularity of read aheads, which must be an
    //  even power of 2 and >= PAGE_SIZE
    //  See Also: CcSetReadAheadGranularity.

    ULONG Granularity;

    //
    //  The request size in number of bytes, to be
    //  used when performing pipelined read-aheads.
    //  Each read ahead request that is pipelined is
    //  broken into smaller PipelinedRequestSize
    //  sized requests. This is typically used to
    //  increase the throughput by parallelizing
    //  multiple requets instead of one single big
    //  one.
    //
    //  Special behavior:
    //  If this value is zero, then Cc will break
```

(Continued on page 15)

File System Changes...

(Continued from page 14)

```
    //  every read-ahead request into two. This is
    //  used for backward compatibility where we
    //  used to break every read-ahead request for
    //  remote FS into two.
    //
    ULONG PipelinedRequestSize;

    //
    //  Growth of read ahead in percentage of the
    //  data that has already been read by the
    //  application so far
    //
    ULONG ReadAheadGrowthPercentage;

} READ_AHEAD_PARAMETERS, *PREAD_AHEAD_PARAMETERS;
```

There is some interesting information in here, but the existing interface does not seem to provide any mechanism for us to obtain or directly modify these values. Perhaps we will see something in a future release, or some clarification as to the ultimate use of this structure.

There are a number of additional new functions exported from the Cache Manager as well: **CcCopyReadEx**, **CcScheduleReadAheadEx**, and **CcSetAdditionalCache AttributesEx**. These new functions introduce the concept of an "I/O issuer" – these new functions take an extra parameter, a PEPROCESS pointer.

Filter Manager Changes

There are a number of Filter Manager changes visible in the new header files as well. Perhaps the most interesting is that there is now a *section* context – so that file system filter drivers can associate specific state with a section (in addition to the other context types already supported by the Filter Manager). From the material available (including preliminary documentation) it appears that the purpose of this is to allow mini-filters to synchronize changes to section objects. A number of new APIs have been introduced to support this, including **FltGetSectionContext**, **FltSetSectionContext**, **FltRegisterForDataScan, FltCreateSectionForDataScan,** and **FltCloseSectionForDataScan.**

The registration structures have changed to accommodate the new section contexts. Note that section contexts are only present in Windows 8 and more recent.

Beginning with Windows 8, Filter Manager now supports filters for the named pipe file system (NPFS) as well as the mail slot file system (MSFS). To filter these, mini-filters must indicate their interest as part of their registration information. To further aid in supporting these, the Filter Manager now provides new functions for opening named pipes (**FltCreateNamedPipeFile**) and mail slots (**FltCreate MailslotFile**). Presumably these are wrappers around the existing OS calls.

The Filter Manager APIs have been extended to support new features, such as the reparse point support (see GUID_ECP_FLT_CREATEFILE_TARGET).

The MDL interfaces are now also available as part of **FltReadFileEx** and **FltWriteFileEx**. While not fundamental changes to the model, they should simplify development for those mini-filters that wish to use the capabilities of existing OS interfaces that have not previously been available via the Filter Manager interface directly.

There are also new Filter Manager APIs for invoking the fast I/O MDL operations directly, as well as Filter Manager wrappers around the get/set interface for quota information.

Finally, there is now support for retrieving multiple contexts simultaneously using the new **FltGetContextsEx**. There is a corresponding "bulk release" operation in **FltRelease ContextsEx**.

Conclusions

There are other changes (notably in the security area) that we haven't been able to cover, but it is clear that while there are a number of interesting new changes for file systems and filter drivers in Windows 8, they are more along the lines of evolutionary, rather than revolutionary.

Stay tuned as we watch Windows 8 evolve – after all, as we said before, none of this is known until the final version ships.

Windows 8 WDK

Converting "Sources." Based Projects to ".vcxproj"

In case you haven't heard, Microsoft has converted the WDK to integrate with Visual Studio 11. What this means for us developers is that the driver projects that we previously built with Win7 WDK build environment will have to be converted to use the new build environment.

When we say that a project needs to be converted, what exactly do we mean? Well, remember that in the previous WDK a driver project had a sources file (SOURCES.) and optionally a makefile.inc. In order to build a driver project in the new build environment you need a Visual Studio project file (.vcxproj). So how do you get of those?

The new version of the WDK introduces a tool called "NMake2MsBuild." What this tool does is the topic of this article. So without further ado let's talk about NMake2MsBuild.

NMake2MsBuild

As we mentioned, NMake2MsBuild is a Microsoft-provided utility that is used to convert an existing WDK "Sources." based project into a ".vcxproj" based project that can be built by Visual Studio or from the command line directly using MsBuild. While we are sure everyone knows about building projects with Visual Studio, there may be a few people who don't know about MsBuild.

MsBuild is a utility that is included with Visual Studio that allows a developer to build Visual Studio based projects (.vcxproj files) from the command line, like the "Build" utility did in the Win7 WDK. So whether you want to build your driver project from within Visual Studio or from the Command Line with MsBuild, NMake2MsBuild is the tool you use to convert your Win7 WDK project.

Now that we've distinguished between MsBuild and Visual Studio, let's continue with discussing NMake2MsBuild.

NMake2MsBuild Command Line

NMake2MsBuild is shipped with the new WDK and is typically found in your %SYSTEMROOT%\Program Files\Windows Kits\8.0\bin\[x86,amd64] directory. Run this command from the directory that contains the "Sources." or "Dirs" file that you want to convert. The syntax of the command is shown in *Figure 1*, with descriptions as follows:

- Sources or Dirs indicates what is to be converted (i.e. a project that has a "Sources." file or a project that contains a "DIRS." file.
- Verbosity is one of the "System.Diagnostics. SourceLevels" which are:
  - *Off* – does not allow any events through
  - *Critical* – Allows only Critical events through
  - *Error* – Allows Critical and Error events through
  - *Warning* – Allows Critical, Error, and Warning events through
  - *Information* – Allows Critical, Error, Warning, and Information events through
  - *Verbose* – Allows Critical, Error, Warning, Informational, and Warning events through
  - *ActivityTracing* – Allows Stop, Start, Suspend, Transfer, and Resume events through
  - *All* – Allows all events through
- SafeMode – does not provide IDE/UI support for Nmake targets but may provide a more accurate conversion for Nmake targets. (Only specify "–SafeMode" if you experience issues during build steps that were previously performed in your project's Nmake targets)
- Arm – adds ARM as a valid target CPU architecture to the project's build configurations. The generated project will still require that the installed build environment and WDK support targeting ARM.

The default logging level for "–Log" is "Verbose" while the default logging level for "–ConsoleLog" is "Information".

As you can see the NMake2MsBuild command has very few options, but don't let that fool you. This is a very complex utility that was written to convert very complex driver projects into ".vcxproj" based projects.

You might be wondering if you have to use the command line to do the conversion to a ".vcxproj" based project. The answer is NO! In Visual Studio 11 you can skip doing this conversion from the command line and do the conversion

(Continued on page 17)

NMake2MsBuild <sources|dirs.> [-Log:[<LogFile>]:[<Verbosity>] [-ConsoleLog:<Verbosity>] –SafeMode -Arm

Figure 1—Syntax for NMake2MsBuild

Converting to ".vcxproj"...

(Continued from page 16)

inside Visual Studio via the "Convert Sources/Dirs" option found via the File Open menu item as shown in *Figure 2*.

Since that is the way most developers are going to do their conversions, we will perform all the conversions of projects described in this article via Visual Studio.

Converting projects with NMake2MSBuild

When you convert a project, NMake2MsBuild will *attempt* to create a ".vcxproj" project file that you can use to build your driver in the new build environment. When we say "attempt" we mean the conversion utility will do its best to perform the conversion. Most of the time, and certainly for any simple projects we've tried, the conversion is performed without a hitch. Unfortunately there will also be times when NMake2MSBuild will not be able to perform the conversion. For example, here at OSR we have projects that been around since NT 4 and we have been hacking, whacking, and patching our sources files to keep the projects building with each subsequent release of newer WDKs. As a result of this "evolution", these old sources files have a lot of crufty and unused things that NMake2MsBuild doesn't like. That shouldn't be very surprising. In fact, it's more surprising that some of these old files worked with Build at all. So unfortunately for us, we're probably going to have to create new projects from scratch. Oh well, it was probably time to clean up all the old junk anyway.

Let's take a look at a couple of example project conversions. For our first example, the project containing the sources file is shown in *Figure 3* below. You will notice that this is for a KMDF driver and there are no pre-build or post-build steps (.i.e. there is no "makefile.inc" file with this project).

(Continued on page 18)

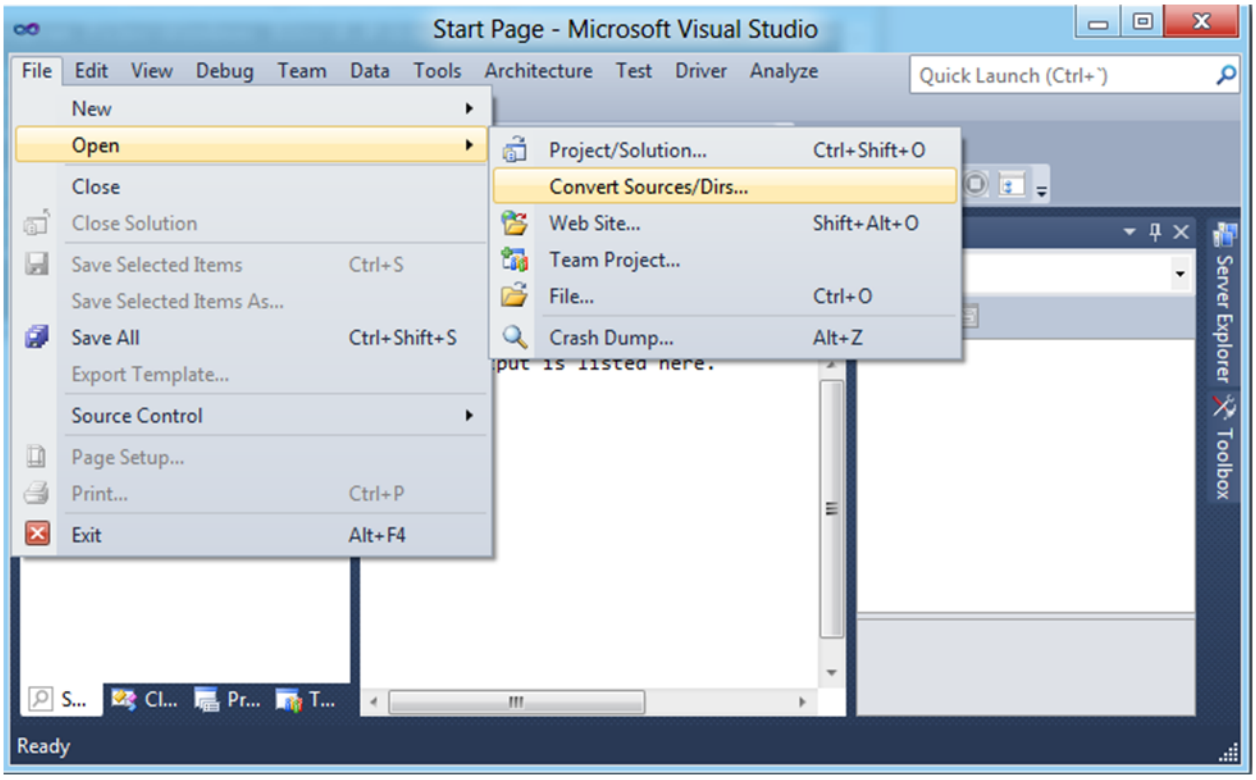


Figure 2—Converting Inside of Visual Studio

```
TARGETNAME=Nothing
TARGETTYPE=DRIVER
TARGETPATH=obj

KMDF_VERSION_MAJOR=1
KMDF_VERSION_MINOR=9

SOURCES=nothing.cpp
```

Figure 3—Nothing Sources. File

Converting to ".vcxproj"...

(Continued from page 17)

From within Visual Studio, we use the "Convert Sources/Dirs" option and get the results shown in *Figure 4* below. Notice that we have a small log written to the Output pane describing how the conversion went (truncated to save space), a more detailed log (NMake2MsBuild_Sources.Log) was written out during this conversion and is stored in the directory being converted. In addition you will see that the files of our example "Nothing" project are now visible in the Solutions Explorer window.

As you can see, the conversion operation here was pretty simple. We had a simple sources file that was easily converted into a ".vcxproj" project. Now, let's look at a little more complicated project.

For our next conversion we will convert a UMDF driver from the Win7 WDK to a ".vcxproj" based project. The sources file for this project is shown in *Figure 5* (next page). In looking at this sources file you should notice a couple of things:

- The project uses WPP tracing
- The project has a custom build step (NTTARGETFILE1) which requires a makefile.inc file shown in *Figure 6 (next page)*.

(Continued on page 19)

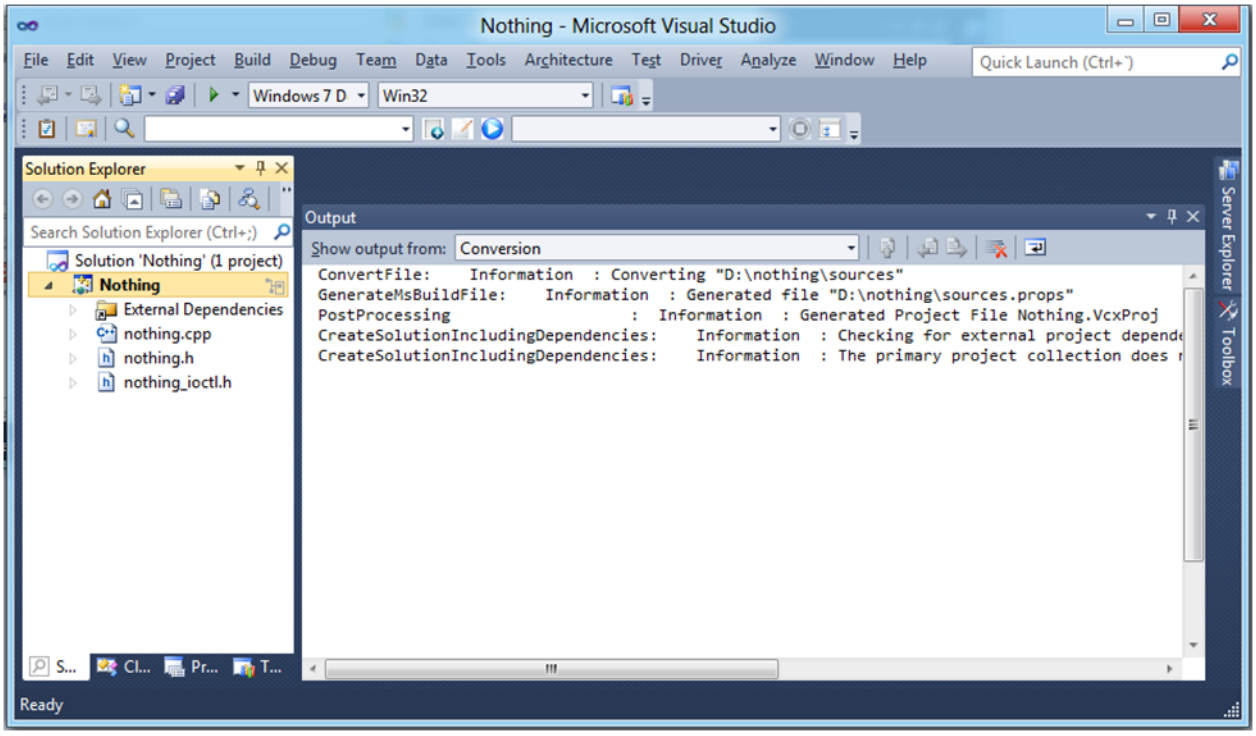


Figure 4—Converting Nothing Project

Converting to ".vcxproj"...

(Continued from page 18)

(Continued on page 20)

```
# UMDF_VERSION_MAJOR controls the headers that the driver uses.
# UMDF_VERSION_MAJOR + UMDF_VERSION_MINOR control which version
# of UMDF the driver is bound to in the INF and which
# update coinstaller it requires (through stampinf).
#

UMDF_VERSION_MAJOR=1
UMDF_VERSION_MINOR=9

KMDF_VERSION_MAJOR=1
KMDF_VERSION_MINOR=9


TARGETNAME=WUDFOsrNothing
TARGETTYPE=DYNLINK

USE_MSVCRT=1

WIN32_WINNT_VERSION=$(LATEST_WIN32_WINNT_VERSION)
_NT_TARGET_VERSION=$(_NT_TARGET_VERSION_WINXP)
NTDDI_VERSION=$(LATEST_NTDDI_VERSION)

#
# Set the warning level high
#
MSC_WARNING_LEVEL=/W4 /WX

#pragma warning( disable: 4201 ) // nonstandard extension used : nameless struct/union
MSC_WARNING_LEVEL=$(MSC_WARNING_LEVEL) /wd4201

#
# If you want to use WPP tracing then uncomment the following line.
#
#C_DEFINES = $(C_DEFINES)  /D_UNICODE /DUNICODE /DUSE_WPP=1
C_DEFINES = $(C_DEFINES)  /D_UNICODE /DUNICODE

DLLENTRY=_DllMainCRTStartup
DLLDEF=exports.def

INCLUDES=$(INCLUDES);..\inc

SOURCES=\
    OsrNothing.rc               \
    dllsup.cpp                  \
    comsup.cpp                  \
    driver.cpp                  \
    device.cpp                  \
    queue.cpp                   \
    ioQueue.cpp

TARGETLIBS=\
        $(SDK_LIB_PATH)\strsafe.lib     \
        $(SDK_LIB_PATH)\kernel32.lib    \
        $(SDK_LIB_PATH)\advapi32.lib

NTTARGETFILE1=$(OBJ_PATH)\$(O)\WUDFOsrNothing.inf

#
# This sets up the WPP preprocessor and tells it to scan internal.h to find
# the trace function definition that's in there.
#

RUN_WPP= $(SOURCES) -dll -scan:nothing.h

TARGET_DESTINATION=wudf
```

Figure 5—UMDF Driver Sources File

```
.SUFFIXES: .inx

STAMP=stampinf

# $(OBJ_PATH)\$(O)\$(INF_NAME).inf: $(_INX)\$(INF_NAME).inx
.inx{$(OBJ_PATH)\$(O)}.inf:
    copy $(@B).inx $@
    $(STAMP) -f $@ -a $(_BUILDARCH) -k $(KMDF_VERSION_MAJOR).$(KMDF_VERSION_MINOR) -u
$(UMDF_VERSION_MAJOR).$(UMDF_VERSION_MINOR).0
```

Figure 6—Makefile.inc for UMDF Project

Converting to ".vcxproj"...

(Continued from page 19)

For this conversion, Nmake2MsBuild has to take both the WPP tracing and the custom build steps into account when performing the conversion. We'll skip showing the conversion output since it worked without a problem, but what we will show you is the output of a build of our converted project.

See *Figure 7* for output of the converted project in Visual Studio.

See *Figure 8* (next page) for the output of the converted project in MsBuild.

Notice that the NTTARGETFILE1 step that was created to do a StampInf call was successfully imported into the ".vcxproj" project and performed when we built the project.

(Continued on page 21)

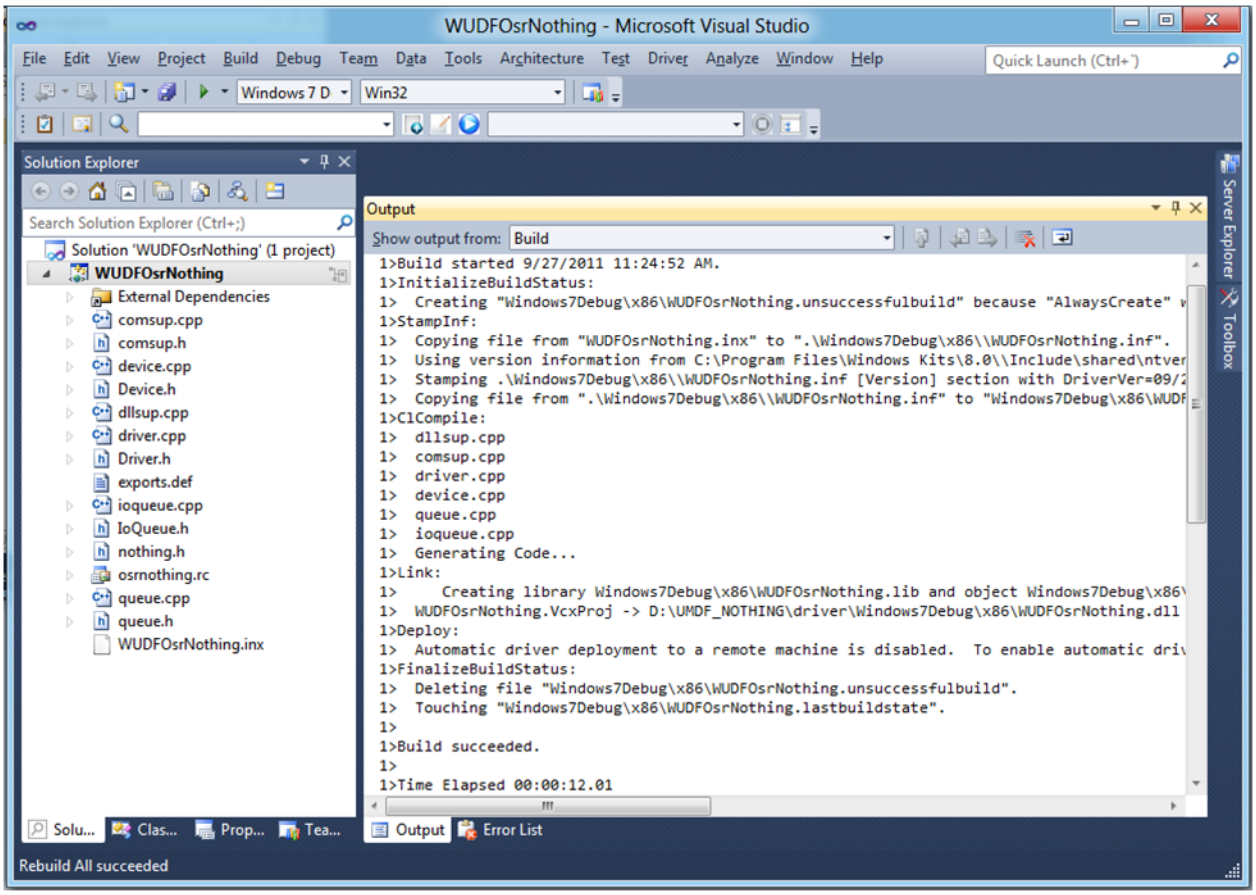


Figure 7—Building a Converted Project Using Visual Studio

Converting to ".vcxproj"...

(Continued from page 20)

Once the conversion is complete, you can build the new project from within Visual Studio just like you build an application. Or, if you prefer, you can build your project outside of Visual Studio by invoking MsBuild directly via the command line. Either way, notice (and this is a *big deal* if you ask me) the new WDK automatically adds project steps to sign your driver (you can configure it to either use a test signing certificate or your release signing cert), and optionally *even automatically copies your driver to a test system and install it*. You really can't ask for much more than that. The key to getting these features is using the Win8 WDK. And the key to using the Win8 WDK is converting your project from sources/dirs. to ".vcxproj" format.

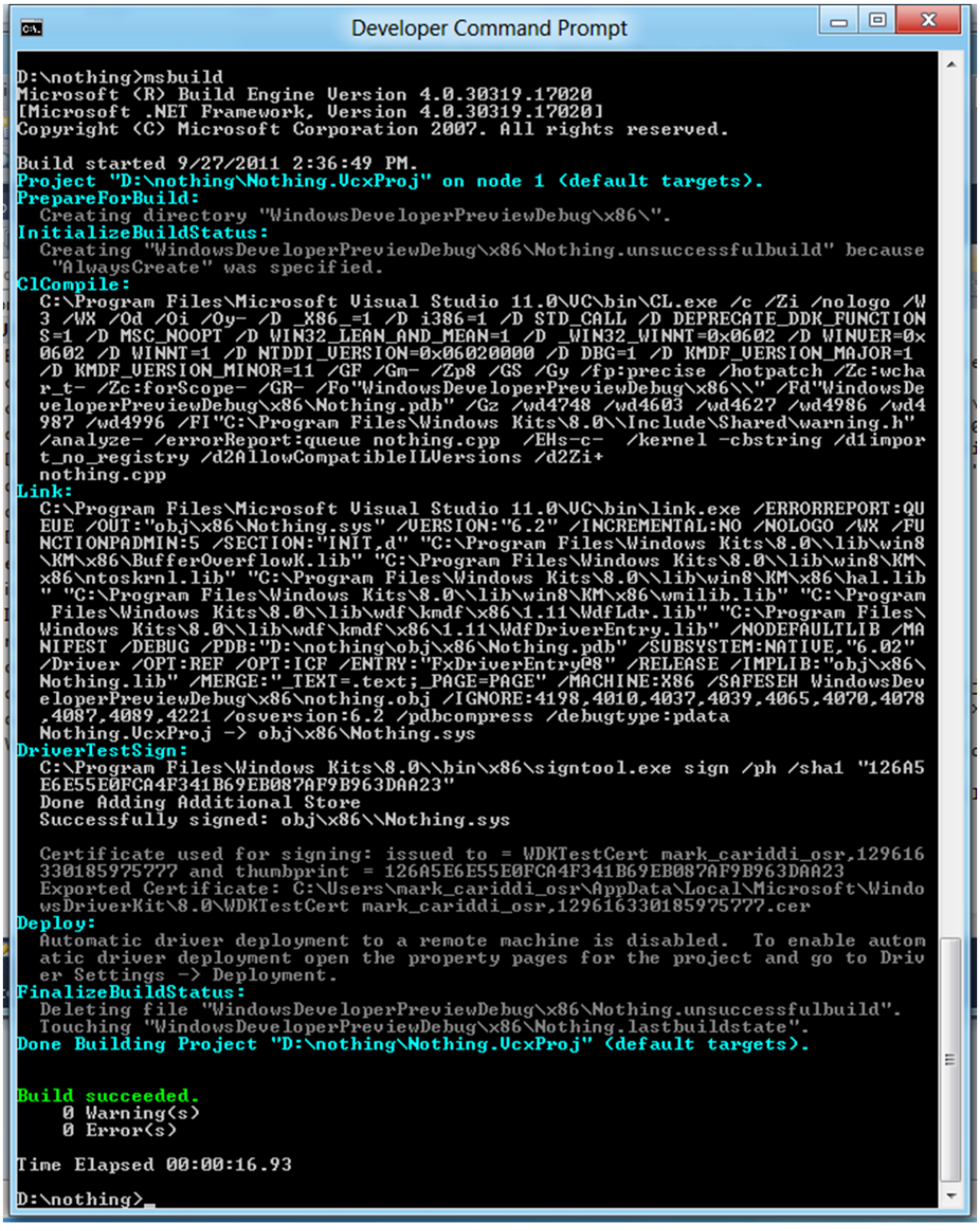


Figure 8—Building a Converted Project Using MsBuild

Summary

In this article on NMake2MsBuild, we have shown that, in general, converting simple driver projects from traditional dirs/sources format to the new ".vcxproj" format used in the Win8 WDK is pretty simple. Whether you choose to do your conversion from the command line with Nmake2Msbuild or from within Visual Studio, this Microsoft utility strives to do the right thing. And as we mentioned, there are going to be projects that may not be able to be converted, but hey, if it were easy, this conversion to Visual Studio would have been done a long time ago.

Epic Update...

(Continued from page 1)

complete with all the necessary cross-certificates so you don't have to go searching for them.

The Windows Driver Kit, which used to be called the Windows Driver Development Kit (DDK), has gone through only two significant changes in its almost 20 year history. The first versions of the DDK required developers to separately acquire and install the compiler and Platform SDK, and then install the driver kit. The DDK itself was assembled by hand. The few example drivers supplied rarely built without error, and when they did they rarely ran correctly. Installing the kit was a major PITA, a real trial in fact, as it required you to provide just exactly the correct versions of the VC++ and the PSDK and install them in precisely the correct order. This was true even when the "correct" versions of VC++ and the PSDK were no longer available from MSDN. So, woe unto the poor developer who didn't save the proper CDs! In short, these early versions of the DDK seriously sucked, but we lived with them for many years. As driver developers, many of us felt like Microsoft didn't much care about us or the tools they gave us to do our jobs.

The first major change to the DDK came with the introduction of Windows XP in 2001. Starting with this version of the DDK, the compiler, linker, headers, and libs were distributed as integral parts of the DDK. Many more sample drivers were provided, and these drivers all built without errors (though, not necessarily without warnings). While the samples were of varying quality, they mostly worked. It is hard to overstate how much having all the tools bundled into the DDK improved our lives as developers. Installing the DDK became simple; no longer an excruciatingly painful exercise in self-flagellation. It was an awesome step forward. Coincidentally or not, it was in this timeframe that many driver developers started to feel like Microsoft was listening to us, and trying to help meet our needs.

For the past several years, members of the driver development community have been asking for a more modern driver

(Continued on page 23)

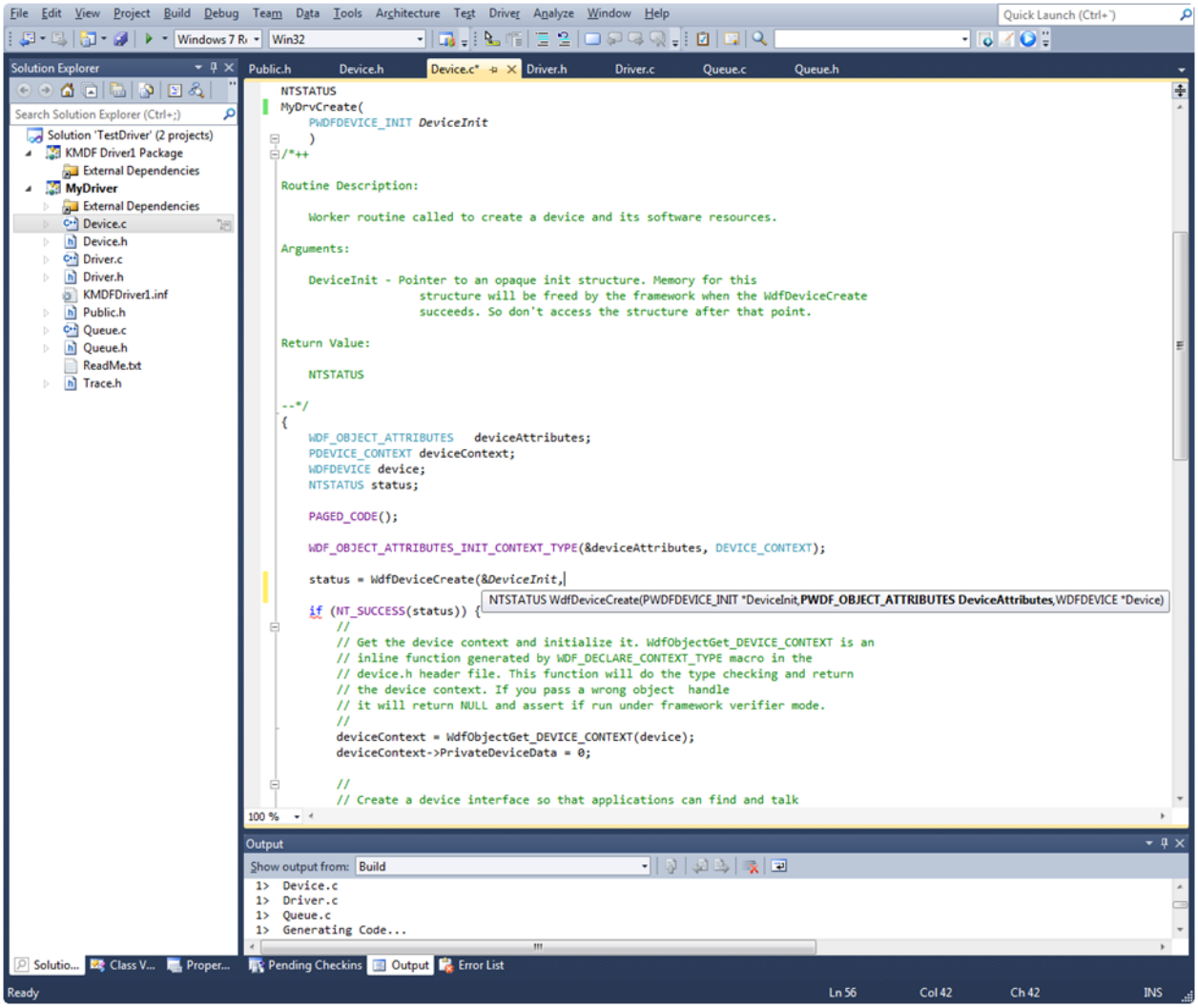


Figure 1 -- Driver Writing in Visual Studio with the Win8 WDK

Epic Update...

(Continued from page 22)

development environment. They didn't want to build from the command line anymore. Because many devs work on both user-mode and kernel-mode code, they wanted to be able to use one tool, Visual Studio, for both their user-mode and kernel-mode development tasks.

And thus, ten years after the introduction of Windows XP, the second major change to the Windows Driver Kit has now come to light. Gone are our old friends the various build environment command prompt windows, **build**, **sources**, and **dirs**. In their place, we get Visual Studio with IntelliSense, class browsing, and all the features I mentioned earlier. You can see what the interface looks like in *Figure 1*.

While this is an epic, most excellent, and much appreciated change, all is not sweetness and light with the Win8 WDK. We're not merely getting a new alternative for building drivers for Windows. The old tools are totally and completely gone. This means you will not find **build.exe**, **makefile.new** or **setenv.cmd** anywhere in the Win8 WDK kit. So, if you want to use the new WDK, you *will* migrate to the new environment whether you want to or not. Fortunately, the WDK provides a utility that'll convert most (many? some?) projects from the old sources/dirs format to the new vcxproj format. We describe the conversion process in this issue (See *Converting "Sources." Based Projects to ".vcxproj"*) . However, the point here is that Visual Studio integration isn't optional. It's the way things are, and the way things will be going forward.

There's another detail that's likely to cause many dev teams at least a little heartburn. This is the fact that, as shown in *Figure 2*, the new Win8 WDK does not support building drivers for Windows XP. Because most of us don't have the option of not supporting XP, this means we'll need to maintain parallel build environments with parallel project descriptions. Fortunately, once a project is developed, you typically don't have to play with the project's build description very much. But still, having to maintain two entirely separate project build environments, with two entirely different sets of project metadata, is just asking for trouble. Perhaps, if we ask nicely enough, we'll be able to prevail upon the WDK team to support building drivers for Windows XP in the Win8 WDK.

And, of course, there are rough edges. Some of these are undoubtedly due to the early nature of this release. But some of them look like they'll be with us into the eventual final release of the kit. I guess we can't have everything we might wish for given to us all at once.

But, for a while at least, let's not dwell on the negative. Let's celebrate this epic change that so many of us have been requesting. Let's praise the fact that they chose to do it right and not just toss in some command procedure that invokes the old build environment through a new front-end. And let's recognize and admire the effort (and, to put it quite frankly, the balls) it's taken to get the WDK from where it's been for the last ten years to where it is today.

I, for one, am stoked about the changes. I hope you are too.

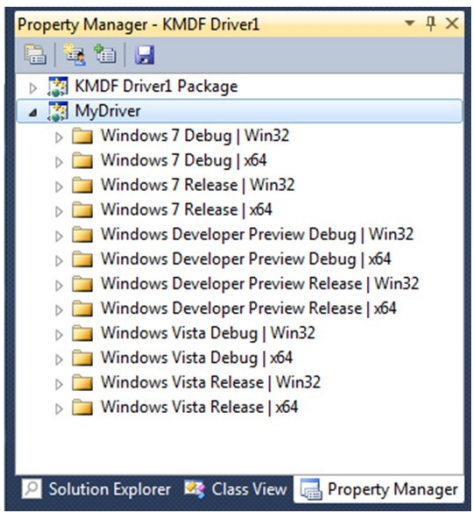


Figure 2 -- Hmmm... No Windows XP?

WDK Preview...

(Continued from page 7)

encrypt the KD data transferred. For our purposes we can just leave the defaults the way that they are, but now is when you could change the debug transport to be 1394, USB, serial, etc. or change any specific debug settings.

At this point, we're all set to click Configure and be presented with the dialog in *Figure 6*.

Expect this process to take several minutes, including a few reboots of the target machine. If you receive any access denied errors during this process, make sure that you have UAC *disabled* on the target and *enabled* on the host. Also be sure that you've run VS elevated on the host machine.

If everything works, you will eventually be told that the configuration succeeded and again be presented with the Attach to Process dialog. However, this time you'll be given the option to attach to the target machine, as can be seen in *Figure 7*.

(Continued on page 25)

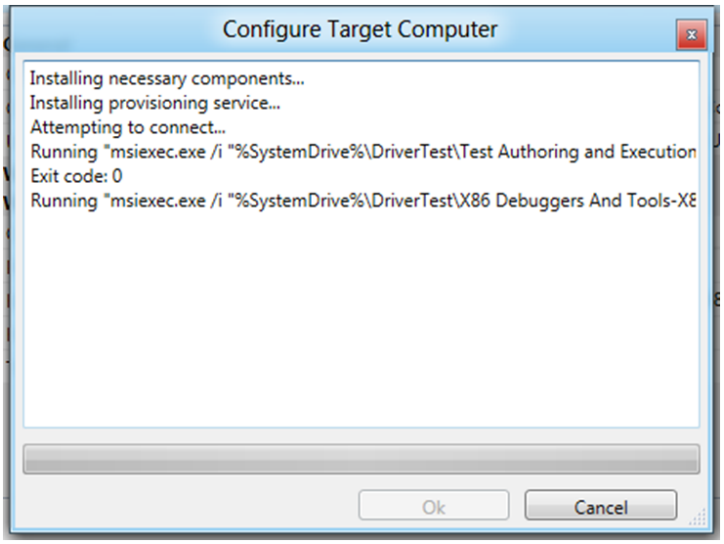
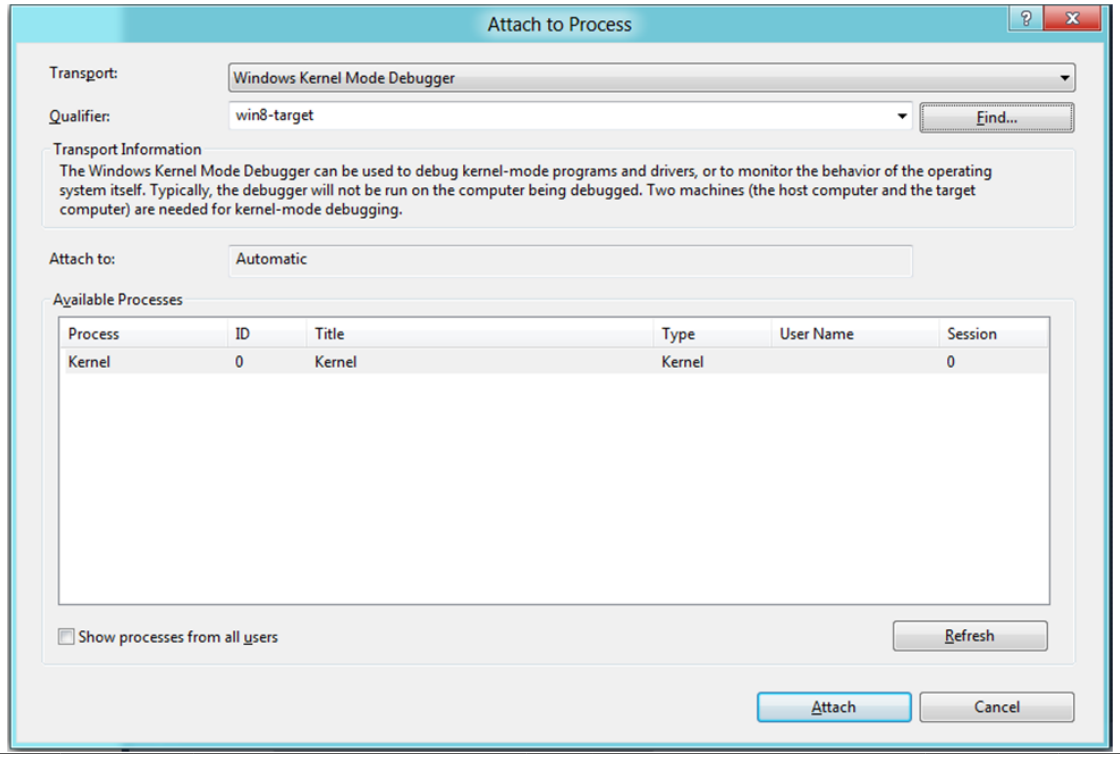


Figure 6 (above) and Figure 7 (below)



WDK Preview...

(Continued from page 24)

It's worth mentioning that we had several cases where the configuration process appeared to hang during the kernel debug options stage. Checking the target machine, however, showed that kernel debugging was properly configured. We reported this to the WDK team and it's being investigated. For now, it is safe to assume that if you make it to the, "Configuring kernel debugger" stage you can cancel out of the configuration but still successfully attach to the target.

The last thing to do is break into your target machine by clicking the Debug->Break All menu item (*Figure 8*). Then you're all set to start enjoying your new development environment (*Figure 9*)!

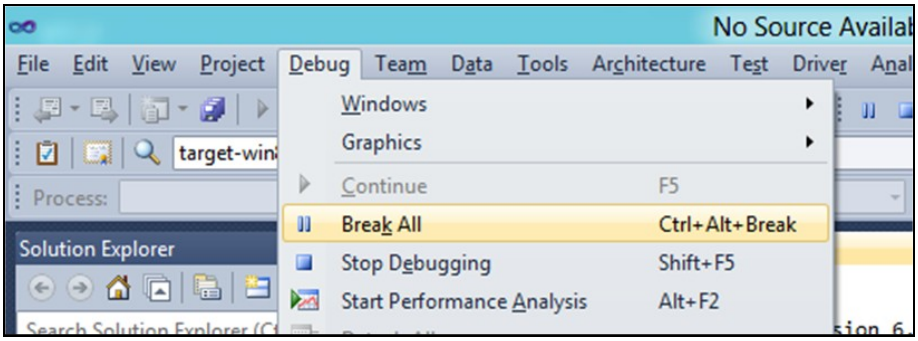
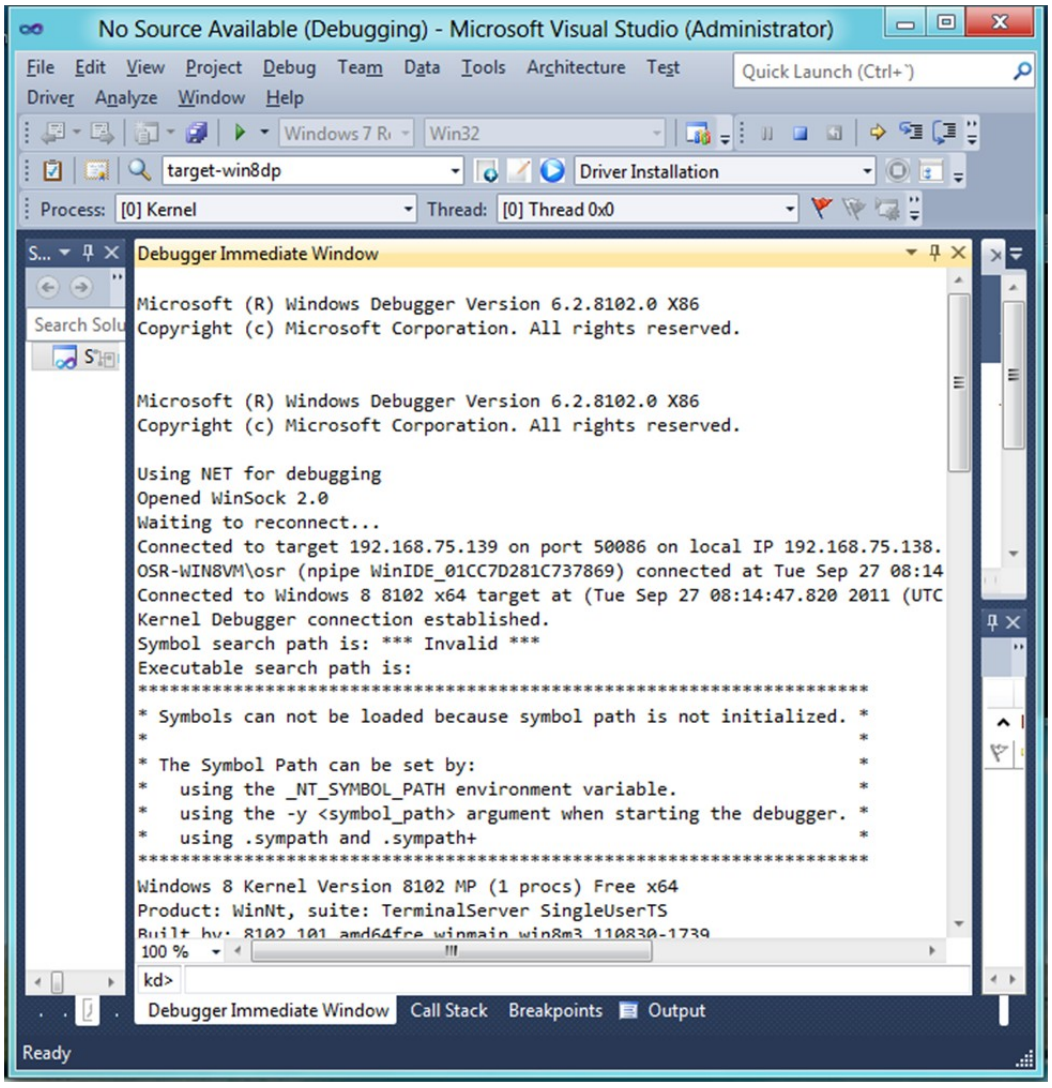


Figure 8 (above) and Figure 9 (below)



Five Things to Like...

(Continued from page 10)

third party driver symbols. This option has been available in WinDBG for a long while, but required magical incantations found only in a random MS Word document shipped with the debugger.

2. Better Experience Setting Breakpoints via the GUI

Using the WinDBG GUI to set breakpoints has always been a bit of a hassle. First off, you need to remember to break into the target before setting the breakpoint. Next, your driver had better already be loaded or you'll be punished with a long delay followed either by an error or no visual clue of whether or not your breakpoint was set. You're then forced to just, "try it" and see if the breakpoint was actually set or not.

Using VS, you're no longer responsible for stopping the target to set the breakpoint; All you have to do is open your source and hit F9. If the target is running, VS will stop the target, set the breakpoint, and then resume it seamlessly. In addition, it also doesn't matter if your driver is loaded or not. Just hit F9 and VS will set an unresolved breakpoint and, more importantly, will display the breakpoint in the GUI as it would any other breakpoint (see *Figure 2*).

3. Suppress PREfast Warnings via the GUI

I'm all for anything that helps us write better code. For the annoyance it caused in the beginning, it's hard to say that PREfast doesn't do just that. However, for all the times that it does right by me, the times that PREfast is wrong or spurious make me insane. All I want is for the warning to go away, but can never quite seem to remember the magic #pragma incantation to suppress the warning. This generally results in me searching my projects directory for, "#pragma" and looking for the last time I had to suppress something.

```
status = WdfDeviceCreate(&DeviceInit,
                         &deviceAttributes,
                         &device);

if (NT_SUCCESS(status)) {

    deviceContext = WdfObjectGet_DEVICE_CONTEXT(device);
    deviceContext->PrivateDeviceData = 0;
```

Figure 2

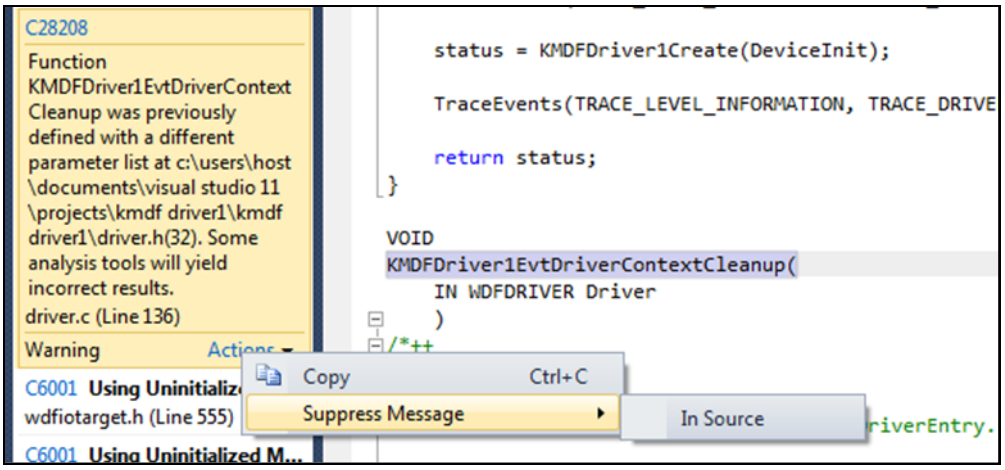


Figure 3

```
VOID
#pragma warning(suppress: 28208)
KMDFDriver1EvtDriverContextCleanup(
    IN WDFDRIVER Driver
    )
/*++
Routine Description:

    Free all the resources allocated in DriverEntry.
```

Figure 4

You can imagine then how excited I was when I discovered that not only can PREfast be run via the GUI, but the warnings can also be suppressed with a few clicks (see *Figure 3*).

Suppressing the message adds the appropriate #pragma to make the warning go away, as can be seen in *Figure 4*.

You'll note, however, two down sides to the automatic suppression. First, I'd like to see the suppression using a pre-defined constant with a meaningful name for the warning number. During a code review, I'd have absolutely *no* idea what warning 28208 was, which would make me wonder if it should really be suppressed or fixed. In addition, I'd like to see the suppression put some stub text either above or as part of the #pragma that leaves room to explain the warning. Again, when someone comes back to this in the future it would be nice if it was very clear about why this was suppressed instead of fixed.

Continued on page 27)

Five Things to Like...

(Continued from page 26)

4. Intellisense for Debugger Commands

This one doesn't need much explaining and it's just awesome once you've had a chance to use it. Start typing out a WinDBG command and you'll be greeted with a list of matching commands and even some documentation, as can be seen in *Figure 5*.

Sometimes you even get information about the parameters to the command, as can be seen with *.show_sym_failures* in *Figure 6*.

My only complaint at this point with this feature is that parameter information isn't shown for *all* commands. Hopefully as the support matures we'll get that in the future.

5. Automatically Configuring Target Machines for Debugging

I don't think that I need to convince anyone that BCDEdit is cumbersome and cryptic. If I'm not using the default debug configuration of COM1 at 115200, I'm lost when it comes to configuring a target machine for debugging. Luckily for all of us, those days are now over given the new support in VS for automatically configuring target machines. We're covering how to configure a target machine for debugging via the UI in another article in this issue (see *WDK Preview: Installation Through Debugging*), so I won't go through that again here. However, when it works, debugging is just a click away and that's pretty awesome.

One gripe I have about this is that when you configure the target machine for kernel debugging it doesn't add a separate BCD entry for debug. Thus, you'll be forced to always boot with the debugger enabled unless you change the settings during the boot process by hitting F10.

What's Do You Think?

What are your impressions after playing with the preview? Agree with the things we like here? Disagree? Have your own? Let us know! And be sure to check out our other article in this issue, *Five Things to NOT Like About Visual Studio Integration*.

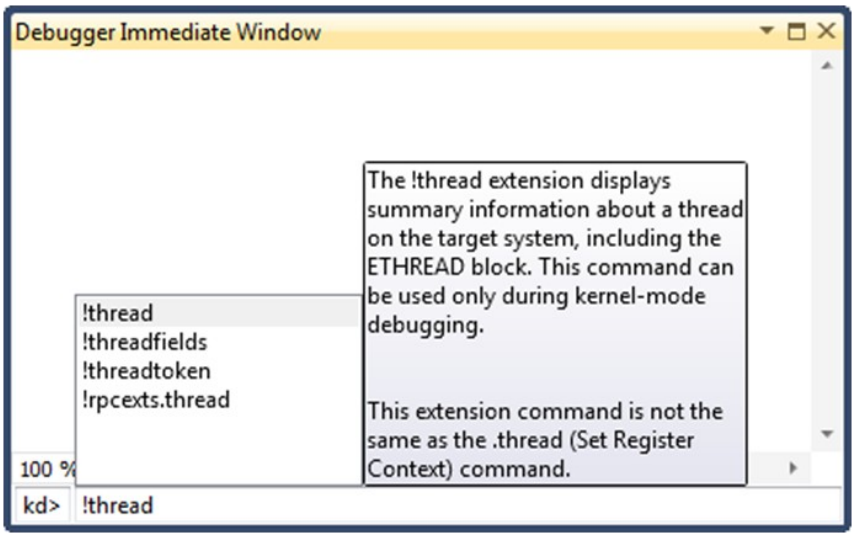


Figure 5

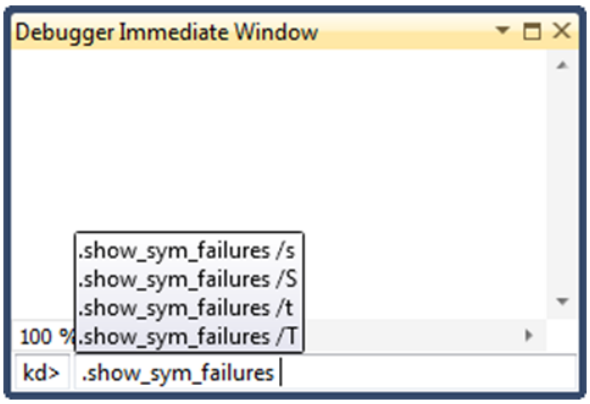


Figure 6

Five Things to Not Like...

(Continued from page 11)

code, hit F5 to run it under the VS debugger, and step through your code until you're satisfied. You then end the debug session and start working on your next piece of code.

Compare this to what happens when you're a driver developer. In this case, you already have an instance of WinDBG running and attached to the target machine. In your separate build environment, you build your driver and then copy it over to the target. You then perform some manual action on the target to load the new driver, possibly rebooting or just disabling/enabling a device. When your new driver loads, you set some breakpoints and step through the code. Once you're satisfied, you leave your debug session intact and move back to your editor so that you can repeat the process.

Given this, what's the issue? The trouble is that the VS IDE was created for the user application scenario, not the device driver scenario. Thus, while I'm attached to my target machine within the VS IDE, the build options are disabled. In order to build a new copy of the driver, I need to detach from the target machine, build, and then re-attach to the target machine. This is quite annoying, especially in the case where the re-attach doesn't work, which happened to me several times while network debugging.

Hopefully this is something that's addressed in the final release. Otherwise, we'll likely have to have two instances of VS running or revert to using WinDBG as the kernel debugger.

5. What it Means for the Future of WinDBG

While people like to bash it, WinDBG is truly an excellent, lightweight tool for crash dump analysis. I just don't ever see myself saying that about VS, so I'd really hate to see WinDBG fade into oblivion. Luckily it looks like we'll have WinDBG support with us at least through this release of Windows, but the future beyond that is not clear at the moment.

What Did We Miss?

We just *know* that we missed your least favorite thing about the new kit, so let us know what it is!

Peter Pontificates...

(Continued from page 5)

When I made my next request, I knew I was pushing the envelope. But I figured if I didn't aim high, I'd probably never get anything. So, I asked for:

> *I want real-time PFD [Prefast syntax checking] from within the [VS] IDE. You call a function, and you get a squiggly underline and a message saying you're trying to call it at the wrong IRQL. Ever use tools like ReSharper from Jet Brains? Visual Assist by Whole Tomato? I have, and they're pretty freakin' cool. They provide real-time feedback on the code you write, telling you all sorts of things about your usage.*

So, I asked, if ReSharper can do it, why can't we have the same kind of analysis done in real-time by Prefast?

Apparently, the answer is that we can't have it, at least not yet. In fairness, I have no idea what type of effort would be required to implement this feature. It's a user-mode thing, so I have no clue. And it *is* important to note that we *did* get a pretty reasonable level of PFD integration in the Win8 WDK. When you build your driver project, it can be set to automagically run PFD from within the VS IDE and display the warnings or errors. And this is pretty cool.

So, in terms of dynamic PFD integration within VS, let's say that I got 50% of what I asked for.

My penultimate request was for drivers that can do hardware access from user-mode. I wrote:

> *[I want] drivers using interrupts and register access in user mode. It is time to just bite the bullet and do this work, folks. If I get the things I've asked for above, it is beyond doubt that Windows drivers will be more reliable. But we don't just need reliability. We need to limit the consequences of driver failure. To do that, we need to be able to move all drivers (at least those that are not necessary to boot the machine) to user mode.*

I am very happy to say that, on this *big* and complicated request, I got my wish! Or, at the very least, most of it. UMDF does indeed support the ability to process interrupts and do direct register access in user-mode. There are a few restrictions, and some caveats (like, do you really want to handle PCI level-triggered interrupts in your user-mode driver?), but this very critical feature was implemented! Thanks!

User-mode hardware access? I'd say I got 95% of what I asked for. Yay!!

What's a Christmas list without dreams, right? So, not being content to limit my Christmas wish list to just things that were possible within a reasonable timeframe, I just had to add one final item. I asked for the ability to write Windows drivers in C#:

> *I never liked C (I think it's a terrible language). And you've doubtless read my columns over the last year or so talking all about how great it would be to write drivers in C#. You want to really, seriously, eliminate driver bugs? Do away with pool leaks, and memory scribbles, and reference counting forever? Bring on Windows drivers in C#.*

I didn't expect to get the ability to write Windows drivers in C#, and I didn't. But it's nice to dream.

When you add everything up, where does that leave us? Were my Christmas dreams fulfilled, or have I been left holding my Christmas stocking, sadly disappointed?

Well, dreams are just that: Fantasies that could all come true only in a perfect world. In our dreams, there are no

(Continued on page 30)

Peter Pontificates...

(Continued from page 29)

constraints, no limits, to what we can obtain. However, in the real world, one cannot wave a magic wand and make wishes come true. Progress is often only attainable in incremental steps.

Having our driver development environment fully integrated, from code, to build, to sign, to deploy, to test, to debug – is a level of integration that I didn't even dare hope for two years ago. The community has been asking for driver kit integration with Visual Studio for years, and somebody finally tackled the job, and it is here and it works! This is very obviously outstanding work. Plus, we have the integrated ability to run PFD within VS automatically.

And we got user-mode hardware access. This is epic, and it's critical to a more stable Windows in the future.

When I look at what we got, overall, in the Win8 WDK I'm thrilled with what's come to fruition. The Win8 WDK release is the most dramatic and important release since the compiler and linker were added to the DDK. It's courageous, it's visionary. And I'm sure it'll serve as the basis for all sorts of new features in the future. The WDK team deserves a hearty "well done" for their work on this project.

I just really wish they had done VS to sources "round trip" conversion support. That's the one thing that I'll really miss. As far as getting a dynamically revised KMDF, and a driver model that spans user-mode and kernel-mode without changes: I guess I'll simply have to wait and hope.

There's always Windows 9, right?

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

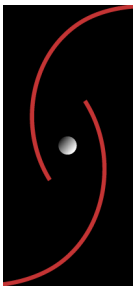
Peer Help?

Writing and debugging Windows system software isn't easy. Sometimes, connecting with the right people at the right time can make the difference. You'll find them on the NTDEV/NTFSD/WINDBG lists hosted at OSR Online (www.osronline.com)

OSR USB FX2 Learning Kit

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

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



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

The NT Insider™ is a subscription-based publication

Subscribe to The NT Insider—Digital Edition

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

New OSR Seminar Schedule!

Seminar	Dates	Location
Internals and Software Drivers (Lab)	17-21 October	Boston/Waltham, MA
Kernel Debugging & Crash Analysis	14-18 November	Columbia, MD
Writing WDM Drivers (Lab)	28 Nov—2 Dec	Santa Clara, CA
Windows Internals for Forensic Analysts	26-29 September	Columbia, MD

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