The NT Insider ™
The only publication dedicated entirely to
Windows system software development

What I Want For Christmas (2011)

I know it's "the holidays" (that's what we call it here in the States anyhow), and around this time of year I usually try to write a more light-hearted, perhaps even funny, pontification. But given the state of the global economy, the state of healthcare in the US, and that fact that I have a paper cut on my finger that's been driving me absolutely freakin' insane for the past day or so, I'm not feeling so very jolly. So, instead of offering good tidings for the season (when do we offer "tidings" except during this time of year, by the way… and are they ever anything but "good tidings"? I often wonder these things, and don't have anyone to ask but you. Well, never mind), and taking into account that plans for the next version of Windows are currently being formulated, I figured I'd describe my Christmas list. For Christmas 2011. This is, according to published reports, approximately when we might expect to see Windows 8 hit the streets.

Let me start by summarizing my wants: What I want for Christmas 2011 is the ability to write Windows drivers more easily and produce drivers with much higher quality. That's all. I mean, how much *is that* to ask for? So, without further ado, here's what I want for Christmas 2011, in order of what I judge to be increasing complexity and likelihood.

- **Driver Development in Visual Studio** – OK, OK, OK. I give up. 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. Visual Slick Edit might be nice, but it's just not enough for me anymore. 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.

  It goes without saying that the VS implementation must use the compiler, linker, headers and libs from a version of the WDK that I define.

  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.

  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.

In short, I need a VS IDE that's a *super set* (not a subset) of what I can do today using build, sources, and dirs.

Readers who are still awake might note that the VisualDDK tool, described in an article elsewhere in this issue, provides some of this functionality. This is great, especially given it's a free, community created, tool. But I want more. I want a complete solution. I want a Microsoft Approved solution.

- **DMA Virtualization** – Given that one of my first loves is DMA, I figured I throw this one in here. 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.

- **More and Better KMDF –** Long time readers will know that I'm a monster fan of KMDF. If you're writing a new driver for which KMDF is suitable, and you're not writing it using KMDF, you're making a bad decision. Pure and simple.

  But what we have from KMDF today is not enough. It's just a start. What I want from the next major revision of KMDF is aggressive, continued, forward architecture and development.

  First, I want the rough edges smoothed out. What do I mean? Well, I can give you a couple of examples off the top of my head: Raise your hand if you think the rules around **WdfRequestSend** and **WdfRequestFormatWhateverForPookieGiggleBlah** are easy to understand. Or, let me know if you think the concept of sync scope, execution-level constraint, and the ability to extend sync scope to various callbacks by setting an attribute during creation of various objects is clear and simple. Are these major problems? No, *totally* not. But there are many things that I consider "rough edges" in KMDF that could use some sandpaper. A major revision of KMDF would be a great opportunity to smooth out some of these bumps.

  Next, 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. Look at the way KMDF abstracts native Windows PnP and Power Management. It doesn't just mirror the way Windows does things, it creates a different, easier to use, coherent, consistent, paradigm. It's a thing of architectural beauty. *Seriously*. Now, look at I/O Targets. The I/O Target model effectively wraps the underlying Windows architectural model with a set of helper functions. It is difficult to understand I/O Targets without understanding I/O Stack Locations. Trust me, I know this. We try to explain I/O Targets without ever mentioning I/O Stack Locations in our KMDF seminars, with some limited success, but it takes a lot of tap dancing and hand waving. I'm not picking on I/O Targets here… just trying to use them as an example.

Reprinted from The NT Insider, Vol. 16, Number 4, November-December 2009

Finally, I want KMDF to include a new set of features and extensions that ease common programming patterns. Passing parameters between ISR and DPC is one such common problem. Canceling in-progress requests is another (one thing that drives me absolutely insane is the fact that I can't effectively use a WDFQUEUE to hold in-progress Requests, because as soon as I forward a Request to a Queue to hold that Request while the I/O completes, that triggers the release of another Request from my "top-edge" Queue). I'm sure I can think of more.

- **KMDF in User Mode** – Why are the WDF models for kernel-mode and user-mode different? Actually, I know the answer, but that answer doesn't make any sense to me. 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.

  So, because this is *my* Christmas list, I would like, pretty please and thank you, the ability to take my KMDF driver (for USB or a filter or whatever) unchanged and compile it for use in either user mode or kernel mode. Yes, I've asked for this before. I asked for this, in fact, before there even *was* a UMDF. There are many advantages to doing this including the ability to debug/test your driver in user mode. Not to mention the fact that this would eliminate the front-end choice folks have to make today: Am I really *sure* that the performance of my driver will be good enough in user mode? Today, if I'm not convinced, I'll opt for using KMDF… thus consigning my driver to kernel mode forever.

  So, a user-mode KMDF wrapper for the same sorts of devices for which you can write a UMDF driver today would make me very happy.

  Hey, here's another idea: While we're doing user-mode KMDF, we might want to look at changing the I/O Manager to intercept I/O requests in user mode *before* sending them to kernel mode. That'd reduce a pair of otherwise unnecessary ring transitions for user mode drivers. Yeah, I know it's not as easy as I'm writing here. Nothing good and useful is ever easy.

- **Real Time PFD in the VS IDE** – Once we have driver development support in VS, we're not done. 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. Everything from bad source formatting to potentially unhandled exceptions… these tools provide suggestions and assistance. I find when I don't have these sorts of tools available, I actually *miss* them.

  What I want is to take this one more level, and do real-time PFD from within the IDE. It shouldn't be impossible. You call a function, and you get a squiggly underline and a message saying you're trying to call it at the wrong IRQL. Stop laughing! It'd be a hell of a lot better than OACR.

Reprinted from The NT Insider, Vol. 16, Number 4, November-December 2009

- **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.

  So, I want the ability for *any type* of KMDF driver – even those that do DMA, handle interrupts, and directly interact with device registers, to be run in user mode. It's not rocket science. It just needs to be done.

- **Windows Drivers in C#** -- Again, this is *my* Christmas list and I get to ask for whatever *I* want. So, I'm asking for something that I think is wildly unlikely but would certainly rock the house. I've written drivers in C# (though, obviously, not for Windows). I like it. I don't like having to use C. For anything. Ever. Of course, 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#.

So, friends, that's what I want for Christmas: Driver writing on Windows that's easier, that's more fun, and that produces more reliable drivers. Is that really so much to ask? And, I figure by making my Christmas list out 2 years in advance, those who are in a position to fulfill at least some of my Christmas dreams might see fit to do so.

At least I can dream, right?

Reprinted from The NT Insider, Vol. 16, Number 4, November-December 2009