The NT Insider

The only publication dedicated entirely to Windows® system software development

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

May — June 2010 Volume 17 Issue 1

Writing a Virtual Storport Miniport Driver (Part III)

Storport is a welcome relief to storage driver writers wishing to develop a driver that exports a virtual device. This third and final article in a series, completes the development of the Virtual Storport Miniport Driver that we started in the earlier two issues of *The NT Insider*.

A Short Recap

In our previous articles we discussed that our example driver design was divided into 2 parts, the upper-edge which handled the Storport interaction, and the lower-edge that implemented the virtual SCSI Adapter and devices (see *Figure 1*). This article continues to discuss the specifics of our example driver's implementation.

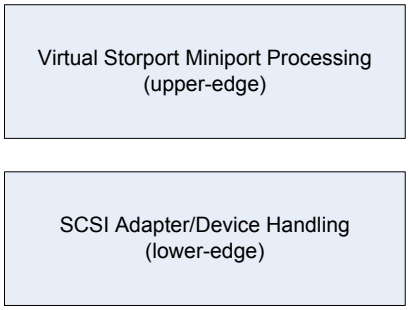


Figure 1 - OSR Virtual Storport Miniport Architecture

In our last article we ended our discussion mentioning 2 points:

- The Virtual Storport Miniport does not use **IOCTL_SCSI_MINIPORT** requests for configuration since this handler would be called at IRQL DISPATCH_LEVEL, and
- To define a SCSI device our driver must respond to Storport with an **INQUIRYDATA**, when requested.

Given those 2 points we'll continue by first describing how the driver handles user configuration requests and then describe how to respond to a Storport request for **INQUIRYDATA**.

Handling User Mode Requests

Because the example Virtual Storport Miniport driver that we're developing needs to respond to user mode requests to configure devices, it needs some way to receive those user requests and also be at IRQL PASSIVE_LEVEL. Being at IRQL PASSIVE_LEVEL is required because our design needs to be able to interact with the Windows file systems. Since **IOCTL_SCSI_MINPORT** doesn't meet the IRQL criteria, we use a new IOCTL that was added for Storport called **IOCTL_MINIPORT_PROCESS_SERVICE_IRP**. The documentation for this IOCTL indicates that it is used by a user-mode application or kernel-mode driver that requires notification when something of interest happens in the virtual miniport. Our use of this might fall a bit outside of its intended use, but it works. Processing of these requests is handled by the **HwProcessServiceRequest** function that the driver set in the **VIRTUAL_HW_INITIALIZATION_DATA** before it called **StorPort**

(Continued on page 10)

Summertime, Summertime, Sum, Sum, Summertime...

Yes, the summer season is just around the corner, folks. What that means for you in your part of the world, I'm not sure. What it means here at OSR is a maddening schedule of driver work, customer support, forward development on toolkits, varied consulting engagements, oh, and several public and private seminars interspersed about it all. Pretty soon, we'll look up from our keyboards, and it will be September. But hey, we'd rather be busy than...well, you know.

So, how can we entice you to join the fun? How about this: **All OSR public seminar registration fees will be discounted by 15% for seminars held in July and August.**

Hope you can take advantage of it and come join us!

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

Inside This Issue:

Peter Pontificates:

Pods, Pads, Ques and Their Kin

According to some of my colleagues, on 3 April 2010 all innovation in information technology ceased. At least, according to them, there was no further progress from that day forward unless Uncle Steve said there could be. And this is because, on that fateful springtime day, the iPad was released and we were all immediately in thrall to it and the apps that Microsoft, oops sorry, I mean Apple, would allow us to run on that device. And this, dear readers, has ushered in the end of the internet, connectivity, and free choice, and would forever place us behind the Great Firewall of Apple.

The logic seems to go that the iPad is such a rip-roaring fabulous device, that it will change computing and how we access information forever. It will bring about, excuse the phrase, a major global paradigm shift the likes of which we've rarely seen. Remember watches before quartz movements? Remember the internet before the web? Remember life before the iPad?? Well??? Do you? After the introduction of each of those disrupters, nothing was the same as it was before, *was it*?

And -- again according certain colleagues -- sitting at the controls of this major change in the way we communicate and consume information is the great Napoleon, Steve Jobs. As gatekeeper, vicar general, General Secretary, or whatever you want to call him, he decides what is and what is not allowed in the new iPadded world. Because, if it ain't on the iPad, it ain't shit.

And it's apparently not just some of my truly insane colleagues who believe this. The press (what's left of it) and -- mostly -- the blogosphere was just awash with this dire prediction.

I don't know about you, comrade, but I don't remember joining this particular party.

The iPad seems to me to be an iPod Touch with a case of Elephantiasis. I don't want one. I don't know why I *would* want one. In fact, I can *barely* conceive of why *anyone* would want one.

But, for the sake of laughs, let's assume that I'm the only person on the planet with this opinion. Let's assume that *everybody else on Earth* (from farmers in Kenya to members of the always-connected generation-upload in Seoul) thinks the iPad is great and can't wait to get their fingers doing those wondrous gestures. After all, somebody must want the damn thing, right? According to published reports, Apple sold something like 750K iPads in 10 days (most of them in California, incidentally... this should tell you something).

Even if everyone thinks the iPad is a great idea, and given that Apple will need to review and bless every application that will appear in the iPad App Store, this still doesn't lead us anywhere remotely near technological slavery. If the "pad" genre catches the imagination of the market, there'll be numerous clones developed. Not everybody will want, or be able to afford, the genuine fruity article. We can be sure that scads of very similar devices will be brought to us by the clever folks in Taiwan, at prices that will be hard to beat. Not to mention, as I write this, Microsoft is reportedly readying "Courier" (a dual-screen device that folds), and Google (according to press accounts of Eric Schmidt after a couple of pops at a recent party) is preparing an Android-based slate. Both of these are said to be direct competitors to the iPad.

(Continued on page 19)

Get Low

Collecting Detailed Performance Data with Xperf

Over the last several years Microsoft has added more and more tracing points to the operating system for use in performance and problem analysis. While Vista collected an impressive amount of information that was available for analysis to the external community, the lukewarm reception of Vista from our customers prevented us from ever really getting a chance to play it. Fortunately, we've seen a bigger interest in Windows 7 that has caused us to take a look at the data that is available and how we might be able to use it. To that end, this article will focus on gaining access to these tracing points via the Xperf utility and interpreting the resulting data with the XperfView utility.

Getting the Tools

All of the tools discussed in this article are distributed as part of the Windows Performance Toolkit (WPT). While the WPT was previously a separately downloadable kit, it is now distributed only as part of the Platform SDK. Unfortunately, that means that you need to install the Platform SDK on some system someplace so that you can extract the WPT installer from the \Bin directory of the PSDK.

There are three installers that are dropped into the PSDK's bin directory: wpt_ia64.msi, wpt_x64.msi, and wpt_x86.msi. All three packages will install both the performance capturing utilities and the data analysis utility, so you'll want to install the package on both your target system and whatever system you expect to be doing your analysis on.

Basics of Capturing Data

Capturing data with Xperf is actually quite simple, though the staggering number of options can at first make the task daunting. We'll cover some basic examples and that will hopefully provide you with enough confidence to experiment and really unlock the hidden power of the tracing present in Windows.

If you don't have any experience with Event Tracing for Windows (ETW), just know that at its heart there are two players involved: trace providers and trace consumers. Trace providers provide a set of *events*, which are really just tracing points that they support. For example, a trace provider might provide a process creation event, which fires when a process is created and provides interesting information about the process. By default a trace provider's events are all disabled and it is the job of the *trace consumer* to both enable these events and capture the data that they generate.

This is pertinent because there are trace providers existing in Windows that contain interesting performance and event data that we want to collect. Xperf is the trace consumer utility that we're going to use to both enable those events and capture the data.

For the purposes of this article, providers come in two flavors: user mode and kernel mode. As the name implies, user mode providers generate events from user mode as a result of user mode actions and kernel mode providers generate events from kernel mode as a result of kernel mode actions.

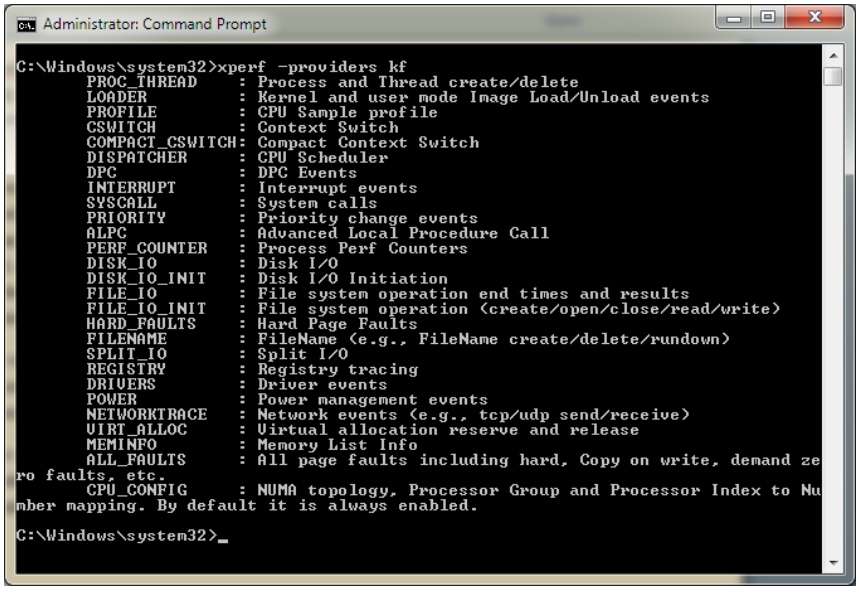


Figure 1—Event Data Available for Collection

Because Xperf is geared towards collecting performance related ETW data, the command line interface is really geared towards collecting kernel mode provider data. Specifically, Xperf is geared towards collecting information from the, "NT Kernel Logger" provider. This provider is supplied by the operating system itself and generates events for all sorts of interesting things, such as DPCs, ISRs, disk I/O, registry access, file access, etc. We can actually see all of the events available to us from this provider by supplying the *–providers kf* switch to Xperf (See *Figure 1*).

From here on we'll focus on gathering kernel trace data and at the end of the article we'll provide a brief overview of how to incorporate user trace provider data with your kernel traces.

(Continued on page 5)

Get Low...

(Continued from page 4)

Starting a Kernel Trace

Now we can begin to capture our first Xperf trace. The basic syntax that we're going to use to initiate a trace operation is:

Xperf.exe –start "Provider Name" –on EventOne+EventTwo+EventThree –f LogFileName.etl

Since we're interested in the O/S provided kernel trace points, our provider name will *always* be, "NT Kernel Logger". In addition, the *EventXxx* in the above will be one or more of the event names that we saw in the *–providers kf* output (PROC_THREAD, LOADER, etc). Thus, if we wanted to begin a trace that would enable the DPC trace events in the system, we would use the following command:

Xperf.exe –start "NT Kernel Logger" –on DPC –f dpc.etl

If we were interested in adding interrupt service routine data to this log file, we could simply use the same command but provide another event in the *–on* options:

Xperf.exe –start "NT Kernel Logger" –on DPC+INTERRUPT –f dpcisr.etl

Unusual syntax? That's a bit of an understatement. But, whatever. It seems to work once you know how to manipulate the options.

Because we typically want to capture multiple points of data at once, Xperf also provides a set of kernel event *groups* that are convenient shortcut names to use when enabling multiple kernel events at once. We can see the list of shortcut names with the *–providers kg* option (See *Figure 2*).

Using one of these when enabling a trace looks like the following:

Xperf.exe –start "NT Kernel Logger" –on IOTrace –f iotrace.etl

You'll notice that in the above groups all options include both the PROC_THREAD and LOADER events. The documentation alludes to this in a few locations, but it is important to *always* include these two events in any kernel trace that you perform. Without both of these events selected the loaded module list will not be captured from the target machine, which will make it impossible to later perform symbolic analysis using PDB files. Thus, even if we are only interested in DPC and ISR event data our start command needs to be:

Xperf.exe –start "NT Kernel Logger" –on PROC_THREAD+LOADER+DPC+INTERRUPT –f dpcisr.etl

Stopping a Trace

Once you've started your trace, stopping the trace is a simple matter of running Xperf again and supplying the *–stop* switch and supplying the name of the provider:

Xperf.exe –stop "NT Kernel Logger"

This will stop the tracing session from generating events and properly close the ETL file specified on the command line. However, before we perform our analysis we will need to complete a *merge* step to add some critical system information to the generated ETL file. This step can be done with the *–merge* option, supplying the ETL file from our trace and an output file name that we will perform our analysis on:

Xperf.exe –merge dpcisr.etl dpcisr_final.etl

In this example, we direct xperf to perform a merge operation on the trace data that was collected in the file "dpcisr.etl" and place the merged output data into the file named "dpcisr_final.etl" – Simple, right?

We now have an ETL file with all of the data that was generated during the capture period plus some extra data that the tool needs to perform some auxiliary functions. You'll find that the contents of the file itself aren't all that helpful, which is why the next step will be to analyze the resulting file using XperfView:

XperfView.exe dpcisr_final.etl

(Continued on page 6)

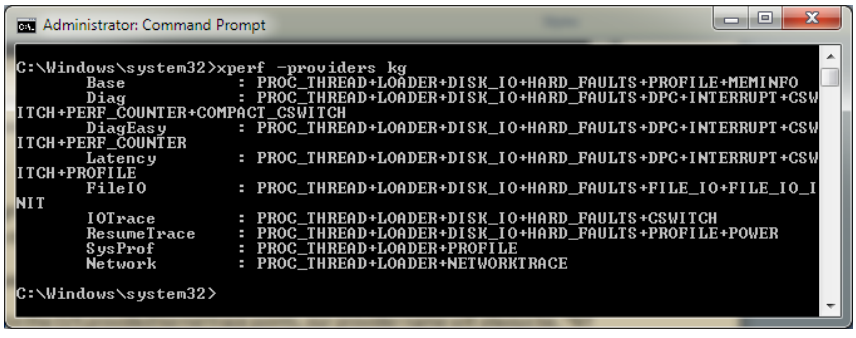


Figure 2 — "Groups" For Enabling Multiple Events

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)

Get Low...

(Continued from page 5)

Analyzing a Trace

Analyzing a trace is where the real work begins. The XperfView application will provide us an incredible amount of insight into exactly what our systems were doing during the test period. Depending on the type of data collected, the application will choose which type of graph best suits the data displayed. For example, for our DPC and ISR collection the application chooses a typical CPU usage graph to display our data, shown in *Figure 3*.

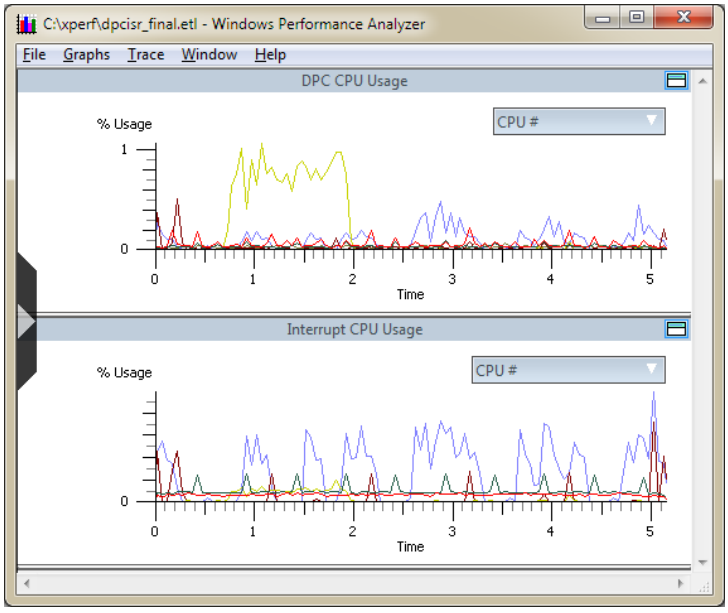


Figure 3 — Graph of the Data

If we would like to zoom in to a particular range of time, we can simply highlight an area on the graph by clicking and dragging. Right clicking at this point gives us access to the Zoom To Selection option (shown in *Figure 4*).

Clicking on this option will cause all graphs in the display to zoom in to that time frame, which will allow us to focus our analysis on interesting spikes such as the one that we see here. Another interesting option available via the right click menu in XperfView is the Overlay Graph option, which allows us to merge any two graphs available on to each other. For example, *Figure 5* shows how we overlay the ISR graph on top of the DPC graph in order to see how the two relate.

From the same right click menu we can also choose the Summary Table option, which will take us to a new window containing detailed information about the events during that time period. However, before doing that it will benefit us to configure XperfView to download symbols for the modules loaded on the target and translate virtual addresses into actual function names.

The way we will do this is to first go to the Trace menu option and choose Configure Symbol Paths. There, we'll want to set the symbol path to point to the Microsoft Symbol Server:

*srv*c:\websymbols*http://msdl.microsoft.com/download/symbols*

Note that this step can be skipped by setting your _NT_SYMBOL_PATH environment variable to the above string.

(Continued on page 7)

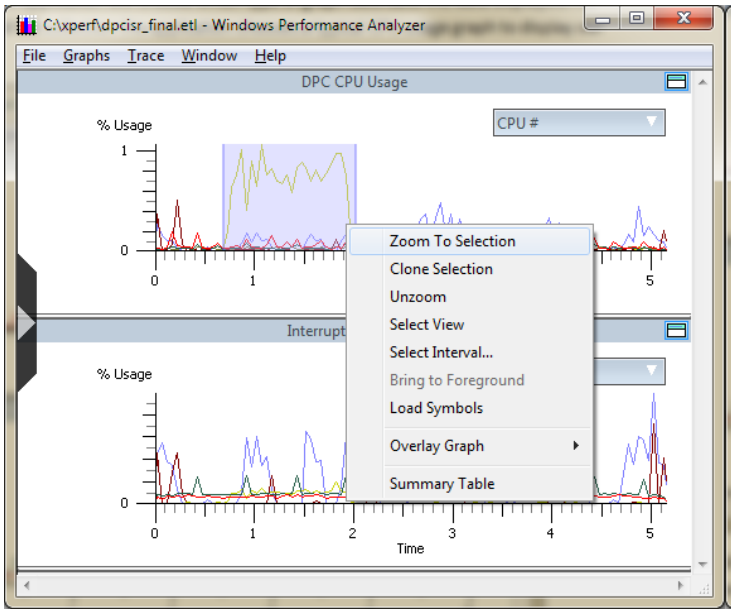


Figure 4 — Zoom To Selection

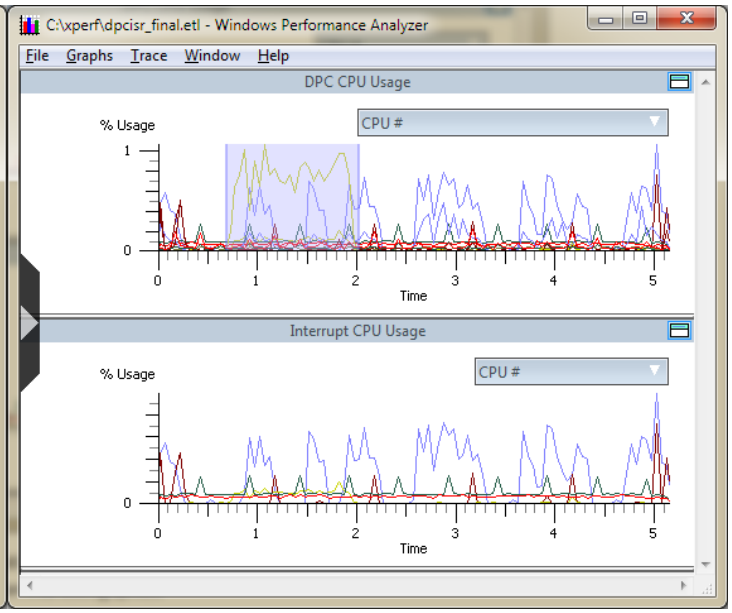


Figure 5 — Overlaying Graphs

Get Low...

(Continued from page 6)

Once our symbol path is set appropriately, we return to the Trace menu and check the Load Symbols option to enact the change. We can then return to the graph view (*Figure* 6) and bring up the Summary Table via the right click menu.

You'll see here that the top number of DPCs seen during the period were from the USB host controller, which makes sense considering the fact that I jiggled my USB mouse during the sample period in the hopes of generating some sort of spike in activity. We can also see the maximum DPC time, the average DPC time, and along with some other columns with more data off screen to the right. The columns shown here are configurable via the Columns menu item.

In all of the previous screen captures you will notice an arrow on the left hand side of the window. This arrow provides access to a window that allows us to show or hide the various graphs that are available from the trace. If you remember, we selected four events when starting our trace: PROC_THREAD, DPC, ISR, and LOADER. If we expand out the window by clicking the arrow we'll notice that we have four graphs to choose from (shown in *Figure 7*).

Choosing Process Lifetimes will bring up an interesting graph that shows which processes were loaded during the course of the trace. We can even see the Xperf process coming and going at the beginning and end of the trace (*Figure 8*).

Enabling Stack Walking

A really cool feature that Xperf provides is the ability to capture stack traces of various events that occur during the trace. Stack walking is enabled by specifying the *–stackwalk* command and, much like the events in the previous examples,

(Continued on page 8)

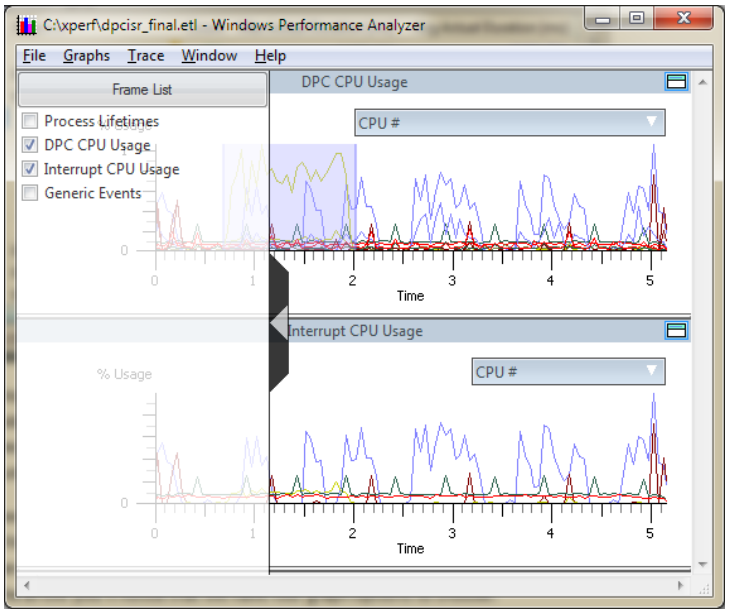


Figure 7 — Show/Hide Graphed Data

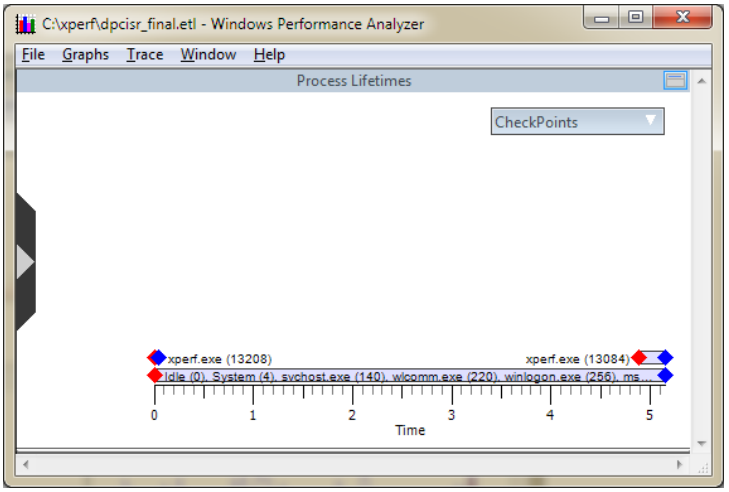


Figure 8 — Process Lifetimes

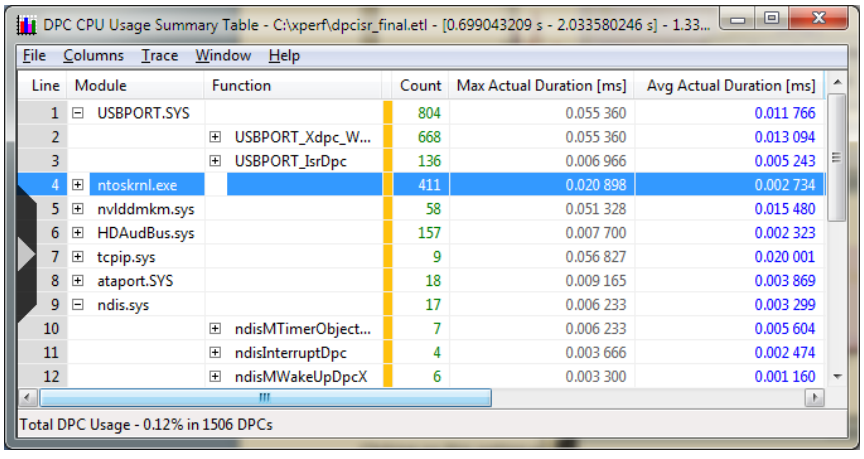


Figure 6 — Summary Table

Get Low...

(Continued from page 7)

we must specify each individual event for which we want this option enabled. The list of stack walking options is too long to list here. The full list of stack walking options is available in the stack walk help:

Xperf.exe –help stackwalk

As an example, we can enable the profiling kernel event, and pair that with the profiling stack walk option:

Xperf.exe –start "NT Kernel Logger" –on PROC_THREAD+LOADER+PROFILE –f profile.etl –stackwalk Profile

Unfortunately, there doesn't appear to be a master list of which stack walking options make sense with which kernel events. This means that the best thing to do is trial and error, just bring up the provider list with *–providers kf* and the stack walk list with *–help stackwalk* and try to come up with combinations that make sense.

Once we've collected our trace with the stack walking option enabled, we'll want to make sure that we have the Load Symbols option enabled so that we can actually make sense out of the stack walks. From there, we can bring up a Summary Table of an interesting period in the trace and add the Stacks column. This gives an impressive view into the call trees of each module during the period, shown in *Figure 9*.

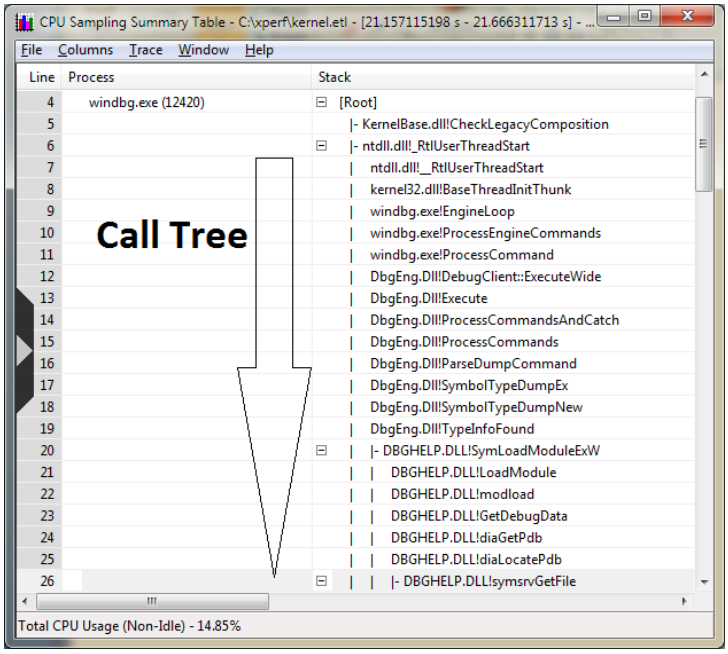


Figure 9 — Finally, Making Sense of the Data

Right clicking any entry in the call tree brings up two options: Callers and Callees. These options allow us to open new summary window with that entry presented as the root. This allows us to narrow the focus down instead of trying to sift through lots of other uninteresting call frames.

Shortcuts

Now that we've discussed the basics, you might want to know that there are shortcuts that can be taken to save some typing. You'll find that many of the existing samples assume that the shortcuts are common knowledge, which can make deciphering those commands tricky if you're not aware of them.

As mentioned previously, Xperf is mostly focused on capturing kernel mode traces. In keeping with this, if a provider name is not specified then a default of "NT Kernel Logger" is assumed. This means that our previous start and stop commands can be abbreviated to:

Xperf.exe –on PROC_THREAD+LOADER+DPC+INTERRUPT –f dpcisr.etl
Xperf.exe –stop

As another shortcut, the merge step can be incorporated with the stop command via the *–d* switch:

Xperf.exe –stop –d dpcisr_final.etl

As a final round of shortcuts, the *–f* parameter and the *–stop* parameter to the above commands are also optional. Thus, you might see someone starting and stopping a kernel trace with the following:

Xperf.exe –on PROC_THREAD+LOADER+DPC+INTERRUPT
Xperf.exe –d dpcisr_final.etl

User Mode Tracing

The very last thing that we'll discuss is how to generate a user mode trace. The commands are much the same as those for a kernel mode trace, but instead of using the "NT Kernel Logger" provider we're simply going to make up our own provider name. In addition, instead of providing a list of events that we want to view from the provider, we simply choose all events of the provider by giving the user mode provider name as part of our *–on* switch. Strange, but true.

The list of user mode providers can be retrieved by executing the following command:

Xperf.exe –providers i

When I ran this command on my workstation, it showed 633 providers to choose from, so obviously I won't list them all here. One caught my eye as amusing, so we'll use the Microsoft-Windows-Sidebar provider and see what results we get:

(Continued on page 9)

Get Low...

(Continued from page 8)

Xperf.exe -start "My User Trace" -on Microsoft-Windows-Sidebar -f sidebar.etl

After then doing a sidebar related activity (adding and removing the Clock gadget in my case), stop and merge the trace:

Xperf.exe -stop "My User Trace" -d sidebar_final.etl

Firing up the resulting ETL file in XperfView results in a series of events displayed and the Summary Table view indicates the events that the sidebar provider fired during the test (See *Figure 10*).

Exciting stuff, huh? No, not really. Perhaps there are other providers with events that are interesting to specific debugging scenarios. Being able to merge a user trace with a kernel trace and plot them against each other would allow us to see the kernel activity during the user scenario, which is a powerful technique. In order to do that we can have both the kernel provider and a user provider collecting data at the same:

Xperf.exe –start "NT Kernel Logger" –on PROC_THREAD+LOADER+DPC+INTERRUPT –f dpcisr.etl
Xperf.exe –start "My User Trace" –on Microsoft-Windows-Sidebar –f sidebar.etl

Once the appropriate testing has been performed, we can stop both providers:

Xperf.exe –stop "NT Kernel Logger"
Xperf.exe –stop "My User Trace"

And then merge the two ETL files together into our final ETL:

Xperf.exe –merge dpcisr.etl sidebar.etl dpcisrsb_final.etl

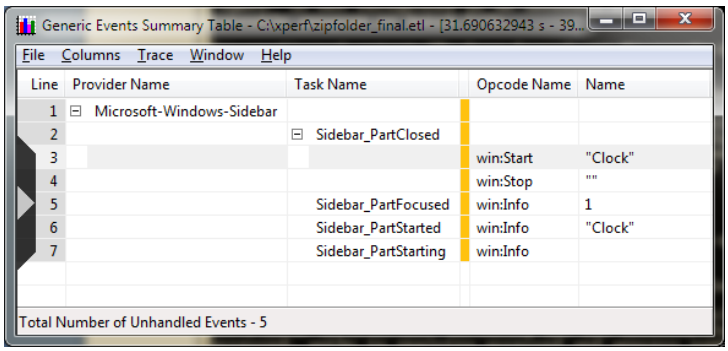


Figure 10 — User-Mode Tracing Example

Really Just the Beginning…

In this article, we've focused solely on runtime logging of performance and other events using Xperf and viewing the resulting data with XperfView. While we've tried to give you the basics, there are a vast number of options and other neat things that can be done with these utilities. In addition, there is an entire other utility XbootMgr for profiling boot related activities. Hopefully we'll have more time to cover Xperf, XperfView, and XbootMgr in the future.

Virtual Storport...

(Continued from page 1)

Initialize in its **DriverEntry** routine. The code for this function is shown in *Figure 2*.

You will notice that the driver ensures that it has received an **IRP_MJ_DEVICE_CONTROL** request before passing it off to its lower-edge **OsrUserProcessIoCtl** routine. This routine processes user mode requests that have data in the buffer supplied in the **AssociatedIrp.SystemBuffer** field of the IRP. This input buffer contains a command code as well as the parameters needed for the command. The commands the example driver processes are:

- **IOCTL_OSRVMPORT_SCSIPORT** – used to confirm that this is indeed the OSRVMPORT driver.
- **IOCTL_OSRVMPORT_CONNECT** – used to connect a new SCSI device.
- **IOCTL_OSRVMPORT_DISCONNECT** – used to disconnect an existing SCSI device.
- **IOCTL_OSRVMPORT_GETACTIVELIST** – used to enumerate the list of active SCSI devices.

We'll discuss **IOCTL_OSRVMPORT_CONNECT** and **IOCTL_OSRVMPORT_DISCONNECT** in this article. You can learn more about the implementation of the other IOCTLs by reading the code.

IOCTL_OSRVMPORT_CONNECT

This IOCTL is used by a caller to request the Virtual Storport Miniport Driver create a new SCSI device based upon provided input information. The caller initializes a **CONNECT_IN** structure and passes it to our driver as the input buffer of the IOCTL_OSRVMPORT_CONNECT IOCTL sent via a Win32 **DeviceIoControl** request. The **CONNECT_IN** structure is defined in *Figure 3*. The **CONNECT_IN** structure allows the user to specify the full path name of the file to be used to back the virtual SCSI device that's being created. It also allows the user to indicate whether or not the device is to be CDROM, and if not, the driver treats the device as a disk. Finally, the user can specify whether or not the media is read-only (the example treats all CDROM devices as read-only by default).

From this input information the driver, creates the **CONNECTION_LIST_ENTRY** structure, shown in *Figure 4* (P. 11), which is used to represent the connection to the media it is using to back the device. The code then calls **OsrSPCreateScsiDevice**, a routine in the "Virtual Storport Miniport Processing" part of our driver (what we refer to as the upper-edge), to create an **OSR_VM_DEVICE** structure, shown in Figure 6 (P. 12), which is used to internally represent the device. Notice that the **OSR_VM_DEVICE** structure contains a pointer to an **INQUIRYDATA** block, described in Part II of this series, which is used to describe our device to Storport. Once these are done, all the driver has to do is announce to Storport that our virtual storage bus has changed by calling the upper-edge routine **OsrSPAnnounceArrival**, which calls **StorportNotification**

(Continued on page 11)

```
#define MAX_NAME_LENGTH 256

typedef struct _CONNECT_IN {
    COMMAND_IN      Command;
    WCHAR           PathName[MAX_NAME_LENGTH];
    BOOLEAN         ReadOnly;
    BOOLEAN         Cdrom;
} CONNECT_IN, *PCONNECT_IN;
```

Figure 3 - CONNECT_IN Structure

```
VOID OsrHwProcessServiceRequest(IN PVOID PDevExt,IN PVOID PIrp)
{
        PIRP                                pIrp = (PIRP) PIrp;
        PIO_STACK_LOCATION        irpSp = IoGetCurrentIrpStackLocation(pIrp);
        NTSTATUS                        status = STATUS_INVALID_DEVICE_REQUEST;
        POSR_DEVICE_EXTENSION        pDevExt = (POSR_DEVICE_EXTENSION) PDevExt;

        OsrTracePrint(TRACE_LEVEL_VERBOSE,OSRVMINIPT_DEBUG_FUNCTRACE,
                                ("OsrHwProcessServiceRequest Enter\n"));

        if(irpSp->MajorFunction == IRP_MJ_DEVICE_CONTROL) {
                __try {
                        status = OsrUserProcessIoCtl(pDevExt->PUserGlobalInformation,pIrp);
                } __except(EXCEPTION_EXECUTE_HANDLER) {
            status = GetExceptionCode();
        }
    }

    if(status != STATUS_PENDING) {
        pIrp->IoStatus.Status = status;
        IoCompleteRequest(pIrp,IO_NO_INCREMENT);
    }

    OsrTracePrint(TRACE_LEVEL_VERBOSE,OSRVMINIPT_DEBUG_FUNCTRACE,
                ("OsrHwProcessServiceRequest Exit\n"));
}
```

Figure 2 - OsrHwProcessServiceRequest Routine

Virtual Storport...

(Continued from page 10)

indicating **BusChangeDetected**. Sometime after doing this Storport will call back into our driver at the **OsrHwStartIo** handler with a **SRB_FUNCTION_EXECUTE_SCSI** request, which will in turn call our **OsrVmExecuteScsi** routine shown in *Figure 5* (P. 12).

When **OsrVmExecuteScsi** is called it will call **FindOsrVmDevice**, shown in *Figure 6*. This routine will find the device, i.e. an **OSR_VM_DEVICE** structure corresponding to the input PathId, TargetId, and Lun (Which was created when we called OsrSpCreateDevice). Finding this structure will result in a call to **OsrUserHandleSrb** to handle the SRB targeted at the device. We will talk about **OsrUserHandleSrb** later in this article.

IOCTL_OSRVMPORT_DISCONNECT

This IOCTL is used by a caller to request that the Virtual Storport Miniport Driver disconnects an existing SCSI device based upon supplied input information. To do this the caller initializes a **CONNECT_IN** structure and passes it to the driver as the input buffer of the IOCTL_OSRVMPORT_DISCONNECT IOCTL sent via a Win32 **DeviceIoControl** request. The **CONNECT_IN** structure, previously discussed and shown in *Figure 3*, is also used for DISCONNECT processing. It allows the user to specify the full path name of the file used to back the SCSI device that is to be disconnected.

If the input information is valid, the IOCTL handler will search thru the list of connected devices and return the **CONNECT_LIST_ENTRY** of the device to be disconnected. Upon getting this entry, the code calls the **OsrSPAnnounceDeparture** in the upper-edge routines which will set the **Missing** field in the OSR_VM_DEVICE structure. The driver then calls **StorportNotification** indicating **BusChangeDetected.** As described in the previous section, calling the **StorportNotification** routine will result in Storport sending a request back into the driver at its **OsrHwStartIo** handler in attempt to identify all devices attached to the drivers' virtual bus. As the driver responds to each call, it will call **FindOsrVmDevice,** shown in *Figure 7* (P. 13), to determine if it has a valid device corresponding to the PathId, TargetId, and Lun being queried. As you can see below, if the driver finds a match in its list, it looks at the **Missing** field in the structure. If this is set to TRUE, then the driver knows that the device is being deleted and will indicate that to Storport. In addition, it will set the **ReportedMissing** field which indicates to **DeleteDevicesThread** that the structure can be removed from the list and deleted because Storport was notified that the device is no longer present.

So now that we know how a SCSI device is added and removed, the only thing we really need to talk about is how the driver will handle a **SRB_FUNCTION_EXECUTE_SCSI** request for a device that is connected to the driver's virtual bus. As mentioned previously, these requests are processed by the driver's lower-edge in **OsrUserHandleSrb**, which we'll talk about in the next section.

OsrUserHandleSrb

The **OsrUserHandleSrb** routine is the code in the lower-edge of the driver that performs the operation described by the **CDB** that's contained in the received **SRB**. One issue to be aware of is that commands contained within the **CDB** come in different sizes. The size of the **CDB** is contained within the **SRB**. No matter what the size of the **CDB** is, the first byte of the **CDB** contains the operation code, and given the operation code, the code can determine how to interpret the rest of the **CDB**. Another issue to be aware of is that the operations received depend on the type of device being exported, and the completion statuses that the driver returns must be a SRB_STATUS_XXXXX and be returned in the **SrbStatus** field of the **SRB**.

(Continued on page 12)

```
typedef struct _CONNECTION_LIST_ENTRY {

LIST_ENTRY                  ListEntry;
WCHAR                       FileName[MAX_NAME_LENGTH*2];
HANDLE                      FileHandle;
UCHAR                       FileAttributes[sizeof(FILE_ALL_INFORMATION)+MAX_NAME_LENGTH*2];
ULONG                       FileType;
PFILE_OBJECT                FileObject;
struct _USER_INSTANCE_INFO* PIInfo;
ULONG                       BusIndex;
ULONG                       TargetIndex;
ULONG                       LunIndex;
BOOLEAN                     HandleClosed;
BOOLEAN                     Connected;
BOOLEAN                     ContainingMediaRemoved;
BOOLEAN                     UNCConnection;
PVOID                       PnPNotificationEntry;
ULONG                       IdentifierIndex;
BOOLEAN                     Closing;  /* Indicates connection is closing */
CONNECT_IN                  ConnectionInfo;

} CONNECTION_LIST_ENTRY, *PCONNECTION_LIST_ENTRY;
```

Figure 4 - CONNECTION_LIST_ENTRY

Virtual Storport...

(Continued from page 11)

Before continuing, we should note an important limitation of the example code presented. The example does not handle the necessary SCSI Operation codes for disks with capacities greater than 2.2 TB. Without this support, the example will handle virtual volumes less than or equal to 2.2TB in size, which we expect will be large enough for just about all purposes. Support for disks greater than 2.2TB first appeared in Windows (for secondary volumes only) starting in Windows Vista. For the example to handle a disk larger than 2.2TB it would have to handle at least the 16-byte variants of the Read, Write and Read Capacity commands. There may be others. In order to determine which 16-byte CDB commands need to be supported, the reader can examine the Disk and ClassPnP code contained with the Win 7 WDK "SRC\STORAGE\CLASS" directory.

Let's take a look of some of the functions that we support in the driver implementation. For the functions not discussed here, please read the source code.

(Continued on page 13)

```
UCHAR OsrVmExecuteScsi(IN POSR_DEVICE_EXTENSION PDevExt,
                                                              IN PSCSI_REQUEST_BLOCK PSrb,
                                                              IN PBOOLEAN PComplete)
{
    POSR_LU_EXTENSION    luExt;
    UCHAR               srbStatus = SRB_STATUS_INVALID_REQUEST;
    NTSTATUS                                    status;
    PCDB                                                pCdb = (PCDB) &PSrb->Cdb;
     POSR_VM_DEVICE                         pOsrDevice;

   *PComplete = TRUE;

    luExt = (POSR_LU_EXTENSION) StorPortGetLogicalUnit(PDevExt,
                                  PSrb->PathId,
                                  PSrb->TargetId,
                                  PSrb->Lun );

    if(!luExt) {
          return SRB_STATUS_NO_DEVICE;
    }

    pOsrDevice = FindOsrVmDevice(luExt,PDevExt,PSrb->PathId, PSrb->TargetId, PSrb->Lun,FALSE);

    if(pOsrDevice && pOsrDevice->PUserLocalInformation) {
           InterlockedIncrement(&pOsrDevice->OutstandingIoCount);
           status = OsrUserHandleSrb(pOsrDevice->PUserLocalInformation,PSrb);
           if(status == STATUS_PENDING) {
               *PComplete = FALSE;
               srbStatus = SRB_STATUS_PENDING;
           } else {
               InterlockedDecrement(&pOsrDevice->OutstandingIoCount);
               srbStatus = PSrb->SrbStatus;
           }
    } else {
           srbStatus = SRB_STATUS_NO_DEVICE;
   }
    return srbStatus;
}
```

Figure 5 - OsrVmExecuteScsi Routine

```
//
// This represents a device that has been detected on a specific bus,target,
// and lun.      PUserLocalInformation represents the handle given to us
// by the lower layer of our code that implements the adapter and scsi devices.
//

typedef struct _OSR_VM_DEVICE {
ULONG          MagicNumber;
LIST_ENTRY     ListEntry;
PVOID          PUserLocalInformation;
ULONG          PathId;
ULONG          TargetId;
ULONG          Lun;
PINQUIRYDATA   PInquiryData;
BOOLEAN        BReadOnlyDevice;
BOOLEAN        Missing;
PVOID          PDevExt;
LONG           OutstandingIoCount;
BOOLEAN        ReportedMissing;
} OSR_VM_DEVICE, *POSR_VM_DEVICE;
```

Figure 6 - OSR_VM_DEVICE

Virtual Storport...

(Continued from page 12)

SCSIOP_INQUIRY (0x12)
As mentioned previously, a SCSI device is described by an **INQUIRYDATA** structure, and this structure is retrieved by Storport via a **SCSIOP_INQUIRY** request. The example driver code to handle this request is shown in *Figure 8* (P. 14). As you can see, the data buffer which is used to return the **INQUIRYDATA** is obtained from the SRB by calling the upper-edge function **OsrSpGetSrbDataAddress.** The data that the driver returns in that buffer depends upon the device (s) the driver is exporting, which in the example driver is either a disk or cdrom. You may notice that the only real difference in the return data is the DeviceType, VendorId, and ProductId: The drive returns **READ_ONLY_DIRECT_ACCESS_DEVICE** for cdrom versus **DIRECT_ACCESS_DEVICE** for disk.

SCSIOP_MODE_SENSE (0x1A)
The **SCSIOP_MODE_SENSE** command is used by the Windows class drivers to retrieve more detailed information about a detected device. Since the device is either a generic disk or generic cdrom, we opted to support and return minimal information. We determined what that information was by looking at the Disk and Cdrom class driver implementations contained within the WDK's "SRC\STORAGE\CLASS" directories. The driver's processing for this command is shown in *Figure 9* (P. 15). The important points to notice in this function are where the driver returns **MODE_DSP_WRITE_ PROTECT**, which tells the requestor that this media is read-only and the **MODE_PAGE_CAPABILITIES** handler where the driver returns capabilities information if the device is a cdrom.

SCSIOP_READ_CAPACITY (0x25)
This function is used by a caller to determine the capacity of the connected device. The caller must return the number of

(Continued on page 14)

```
POSR_VM_DEVICE FindOsrVmDevice(IN POSR_LU_EXTENSION LuExt,
                               IN POSR_DEVICE_EXTENSION PDevExt,
                               IN UCHAR PathId,
                               IN UCHAR TargetId,
                               IN UCHAR Lun,
                               IN BOOLEAN ReturnMissing)
{
    KIRQL    lockHandle;
    POSR_VM_DEVICE pDevice = NULL;

    OsrTracePrint(TRACE_LEVEL_VERBOSE,OSRVMINIPT_DEBUG_FUNCTRACE,(__FUNCTION__": Entered\n"));

    OsrAcquireSpinLock(&PDevExt->DeviceListLock,&lockHandle);

    for(PLIST_ENTRY pEntry = PDevExt->DeviceList.Flink;
        pEntry != &PDevExt->DeviceList; pEntry = pEntry->Flink) {

        pDevice = (POSR_VM_DEVICE) CONTAINING_RECORD(pEntry,OSR_VM_DEVICE,ListEntry);

        OSR_VM_DEVICE_VALID(pDevice);

        if(pDevice->PathId == PathId && pDevice->TargetId == TargetId &&
            pDevice->Lun == Lun) {
            if(!pDevice->Missing) {
                if(LuExt && !LuExt->OsrVmDevice) {
                    LuExt->OsrVmDevice = pDevice;
                }
            } else if(!ReturnMissing) {
                if(!pDevice->ReportedMissing) {
                    OsrTracePrint(TRACE_LEVEL_INFORMATION,OSRVMINIPT_DEBUG_PNP_INFO,
                        (__FUNCTION__": %p Reported Missing, signaling DeleteDevices Thread\n",
                        pDevice));
                    pDevice->ReportedMissing = TRUE;
                    KeSetEvent(&PDevExt->DeleteDevicesThreadWorkEvent,8,FALSE);
                }
                pDevice = NULL;
            }
            break;
        }

        pDevice = NULL;

    }
    OsrReleaseSpinLock(&PDevExt->DeviceListLock,lockHandle);

    OsrTracePrint(TRACE_LEVEL_VERBOSE,OSRVMINIPT_DEBUG_FUNCTRACE,(__FUNCTION__": Exit\n"));

    return pDevice;
}
```

Figure 7 - FindOsrVmDevice Routine

Virtual Storport...

(Continued from page 13)

blocks on the device and the block size. As with every other command the information returned depends on the device. The drivers' handling for this function is shown in *Figure 10* (P. 16).

SCSIOP_READ (0x28) and SCSIOP_WRITE (0x2A)
The **SCSIOP_READ** and **SCSIOP_WRITE** functions, as the names imply, are the functions issued to perform read and write functions on the device, respectively. The caller specifies the logical block number from which to start the operation in the input **CDB** along with the number of blocks to read or write. The caller also specifies an **MDL** that describes a buffer used for the read/write data. It will then be the driver's responsibility to translate the input information into something that makes sense for the device the driver is emulating. Whether the driver is emulating a disk or CD-ROM, this will entail translating the operation into a read from or a write to the file that the driver is using to back the device. We have shown only the **SCSIOP_READ** handler in *Figure 11* (P. 17) since the write handler is almost exactly the same. We'll discuss the actual implementation of the read and write code later in this article, but until then there is one point for you to notice: This function does not complete the SRB, but instead returns **STATUS_PENDING**, which will cause **OsrUserHandleSrb** to return **SRB_STATUS_PENDING** to Storport. This status indicates to Storport that the command has been accepted by the driver but is not yet complete. Any other return status would indicate that the command is complete.

(Continued on page 15)

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

            if(!pBuffer || PSrb->DataTransferLength < INQUIRYDATABUFFERSIZE) {
                status = STATUS_INSUFFICIENT_RESOURCES;
                PSrb->SrbStatus = SRB_STATUS_ERROR;
                goto completeRequest;
            }
            pInquiryData = (PINQUIRYDATA) pBuffer;
            // Fill in the correct Inquiry Data based on the type of device we are emulating.
            if(pIInfo->StorageType == OsrCdrom) {
                pInquiryData->DeviceType = READ_ONLY_DIRECT_ACCESS_DEVICE;
                pInquiryData->DeviceTypeQualifier = DEVICE_CONNECTED;
                pInquiryData->DeviceTypeModifier = 0;
                pInquiryData->RemovableMedia = TRUE;
                pInquiryData->Versions = 2;             // SCSI-2 support
                pInquiryData->ResponseDataFormat = 2;   // Same as Version?? according to SCSI book
                pInquiryData->Wide32Bit = TRUE;         // 32 bit wide transfers
                pInquiryData->Synchronous = TRUE;       // Synchronous commands
                pInquiryData->CommandQueue = FALSE;     // Does not support tagged commands
                pInquiryData->LinkedCommands = FALSE;   // No Linked Commands
                RtlCopyMemory((PUCHAR) &pInquiryData->VendorId[0],OSR_INQUIRY_VENDOR_ID_CDROM,
                              strlen(OSR_INQUIRY_VENDOR_ID_CDROM));
                RtlCopyMemory((PUCHAR) &pInquiryData->ProductId[0],OSR_INQUIRY_PRODUCT_ID_CDROM,
                              strlen(OSR_INQUIRY_PRODUCT_ID_CDROM));
                RtlCopyMemory((PUCHAR) &pInquiryData->ProductRevisionLevel[0],OSR_INQUIRY_PRODUCT_REVISION,
                             strlen(OSR_INQUIRY_PRODUCT_REVISION));
            } else {
                // The media is now either an OSR Disk or a regular disk,
               //  either waywe return the same information.
               ASSERT(pIInfo->StorageType == OsrDisk);
                pInquiryData->DeviceType = DIRECT_ACCESS_DEVICE;
                pInquiryData->DeviceTypeQualifier = DEVICE_CONNECTED;
                pInquiryData->DeviceTypeModifier = 0;
                pInquiryData->RemovableMedia = FALSE;
                pInquiryData->Versions = 2;             // SCSI-2 support
                pInquiryData->ResponseDataFormat = 2;   // Same as Version?? according to SCSI book
                pInquiryData->Wide32Bit = TRUE;         // 32 bit wide transfers
                pInquiryData->Synchronous = TRUE;       // Synchronous commands
                pInquiryData->CommandQueue = FALSE;     // Does not support tagged commands
                pInquiryData->LinkedCommands = FALSE;   // No Linked Commands
                RtlCopyMemory((PUCHAR) &pInquiryData->VendorId[0],OSR_INQUIRY_VENDOR_ID,
                              strlen(OSR_INQUIRY_VENDOR_ID));
                RtlCopyMemory((PUCHAR) &pInquiryData->ProductId[0],OSR_INQUIRY_PRODUCT_ID,
                              strlen(OSR_INQUIRY_PRODUCT_ID));
                RtlCopyMemory((PUCHAR) &pInquiryData->ProductRevisionLevel[0],OSR_INQUIRY_PRODUCT_REVISION,
                             strlen(OSR_INQUIRY_PRODUCT_REVISION));
            }
            status = STATUS_SUCCESS;
            PSrb->SrbStatus = SRB_STATUS_SUCCESS;
            goto completeRequest;
            }
```

Figure 8 - OsrUserHandleSrb SCSIOP_INQUIRY Handling

Virtual Storport...

(Continued from page 14)

Doing Real Work

The previous two articles discussed how to integrate the driver with Storport to become a Virtual Storport Miniport Driver and when integrated, the functions that the driver receives, and how the driver processes them. The current article has so far discussed how to programmatically request the driver to create a SCSI device (IOCTL_OSRVMPORT_CONNECT) and how the driver would respond to Storport requests to identify the device (SCSIOP_INQUIRY and SCSIOP_MODE_SENSE). In addition, we've talked about the preliminary handling of read and write requests via the SCSIOP_READ and

(Continued on page 16)

```
case SCSIOP_MODE_SENSE          : // 0x1A
    {
        // We have received a MODE_SENSE command.  We need to
        // jury rig something up here....  We're returning
        // the bare MINIMUM (as I know it now) information
        // required.  If we need more we'll add it here.
        //
        PCDB                    pCdb = (PCDB) &PSrb->Cdb;
        PMODE_PARAMETER_HEADER  pModeHeader;
        PUCHAR                  pBuffer = (PUCHAR) OsrSpGetSrbDataAddress(pIInfo->OsrSpLocalHandle,PSrb);
         if(!pBuffer) {
            status = STATUS_INSUFFICIENT_RESOURCES;
            PSrb->SrbStatus = SRB_STATUS_ERROR;
            goto completeRequest;
        }
        pModeHeader = (PMODE_PARAMETER_HEADER) pBuffer;
         switch(pCdb->MODE_SENSE.PageCode) {
            case MODE_SENSE_CURRENT_VALUES:
                {
                pModeHeader->ModeDataLength = sizeof(MODE_PARAMETER_HEADER) + sizeof(MODE_PARAMETER_BLOCK);
                pModeHeader->MediumType = 0;
                __try {
                    if(OsrUserIsDeviceReadOnly(pIInfo)) {
                        if(pIInfo->ConnectionInformation->ConnectionInfo.Cdrom) {
                            pModeHeader->DeviceSpecificParameter = MODE_DSP_WRITE_PROTECT; // readonly device
                        } else {
                            pModeHeader->DeviceSpecificParameter = MODE_DSP_WRITE_PROTECT; // readonly device
                        }
                    } else {
                        pModeHeader->DeviceSpecificParameter = 0;  // Writeable Device
                    }
                } __except(EXCEPTION_EXECUTE_HANDLER) {
                    status = GetExceptionCode();
                    PSrb->SrbStatus = SRB_STATUS_ERROR;
                    goto completeRequest;
                }
                pModeHeader->BlockDescriptorLength = sizeof(MODE_PARAMETER_BLOCK);
                PMODE_PARAMETER_BLOCK pModeBlock = (PMODE_PARAMETER_BLOCK) (pBuffer + sizeof(MODE_PARAMETER_HEADER));
                RtlZeroMemory(pModeBlock,sizeof(MODE_PARAMETER_BLOCK));
                }
                break;
             case MODE_PAGE_CAPABILITIES:
                if(pIInfo->ConnectionInformation->ConnectionInfo.Cdrom) {
                    PCDVD_CAPABILITIES_PAGE pCapBlock = (PCDVD_CAPABILITIES_PAGE) (pBuffer +sizeof(MODE_PARAMETER_HEADER10));
                    pModeHeader->ModeDataLength = sizeof(MODE_PARAMETER_HEADER) + sizeof(CDVD_CAPABILITIES_PAGE);
                    RtlZeroMemory(pCapBlock,sizeof(CDVD_CAPABILITIES_PAGE));
                    pCapBlock->PageCode = MODE_PAGE_CAPABILITIES;
                    pCapBlock->PageLength = 0x18;
                    pCapBlock->CDRRead = TRUE;
                    pCapBlock->CDERead = TRUE;
                    pCapBlock->Method2 = TRUE;
                    break;
                }

            default:
                pModeHeader->ModeDataLength = sizeof(MODE_PARAMETER_HEADER);
                pModeHeader->MediumType = 0;
                __try {
                    if(OsrUserIsDeviceReadOnly(pIInfo)) {
                        if(pIInfo->ConnectionInformation->ConnectionInfo.Cdrom) {
                            pModeHeader->DeviceSpecificParameter =  MODE_DSP_WRITE_PROTECT; // readonly device
                        } else {
                            pModeHeader->DeviceSpecificParameter = MODE_DSP_WRITE_PROTECT; // readonly device
                        }
                    } else {
                        pModeHeader->DeviceSpecificParameter = 0;  // Writeable Device
                    }
                } __except(EXCEPTION_EXECUTE_HANDLER) {
                    NTSTATUS status = GetExceptionCode();
                    PSrb->SrbStatus = SRB_STATUS_ERROR;
                    goto completeRequest;
                }
                pModeHeader->BlockDescriptorLength = 0;
                break;
        }
        status = STATUS_SUCCESS;
        PSrb->SrbStatus = SRB_STATUS_SUCCESS;
        goto completeRequest;
    }
```

Figure 9 - OsrUserHandleSrb SCSIOP_MODE_SENSE Handling

Virtual Storport...

(Continued from page 15)

SCSIOP_WRITE operations. So what we really need to do now is discuss where and how the real work is being done in the driver.

As we have mentioned many times, one of the issues of working in the storage stack is that many of the functions are called by the Storport driver at IRQL DISPATCH_LEVEL. We all know that if we are going to interact with the underlying file system containing the files that back the virtual SCSI devices, the driver can't do it at elevated IRQL. This means that the real work in the driver will need to be done in worker threads, running at IRQL PASSIVE_LEVEL. When you look at the implementation of the driver, you'll notice that functions like **OsrUserWriteData** and **OsrUserReadData** queue work items (Notice that when the driver specified the size of the SRB extension in the initialization code, the driver added the size of an **OSR_WORK_ITEM**) to a set of worker thread that the driver created on initialization (**OsrUserAdapterStarted**).

A worker thread in the driver will perform the operation specified in the work item. Check out the **DoWorkThreadStart** routine (available in the downloadable code to our driver). It handles work item commands, **DO_CREATE**, **DO_CLOSE**, **DO_READ**, and **DO_WRITE**. Let's discuss each work item command.

DO_CREATE

This work item is used to open a file that is going to back a device that the driver is creating. You'll notice a few things about this handler. The driver:

- Uses **SeImpersonateClientEx** to ensure that it is using

(Continued on page 17)

```
case SCSIOP_READ_CAPACITY       : // 0x25
            {
                ULONG numBlocks;
                ULONG bytesPerBlock;
                //
                // Someone has asked us to read the disk capacity of the device,
                // so here we need to return to the caller the information about
                // the SPECIAL disk we represent.   Sooo here we go.
                //
                PREAD_CAPACITY_DATA pCapacityData = (PREAD_CAPACITY_DATA) PSrb->DataBuffer;
                __try {
                    OsrUserGetDiskCapacity(pIInfo,&numBlocks,&bytesPerBlock);
                } __except(EXCEPTION_EXECUTE_HANDLER) {
                    status = GetExceptionCode();
                    PSrb->SrbStatus = SRB_STATUS_ERROR;
                    goto completeRequest;
                }
                REVERSE_BYTES(&pCapacityData->LogicalBlockAddress,&numBlocks);
                REVERSE_BYTES(&pCapacityData->BytesPerBlock,&bytesPerBlock);
                //
                // Set status in Irp and in the SRB to indicate that the function was successful.
                //
                status = STATUS_SUCCESS;
                PSrb->SrbStatus = SRB_STATUS_SUCCESS;
                goto completeRequest;
            }
```

Figure 10 - OsrUserHandleSrb SCSIOP_READ_CAPACITY Handling

Virtual Storport...

(Continued from page 16)

the security credentials of the caller when opening the file, since the system thread may not have the same access rights as the requestor.

- Opens the file using **ZwCreateFile** for overlapped I/O, so that it can be doing multiple asynchronous operations against the file.
- Converts the file handle returned by **ZwCreateFile** to a pointer to the referenced File Object by calling **ObReferenceObjectByHandle**. This is done so that the driver can directly build and send Read and Write IRPS to the underlying file system.
- Calls **ZwQueryInformationFile**, so that it can get information on the file for commands like SCSIOP_READ_CAPACITY.

Once this command succeeds, the driver can mark the device as connected so that the software knows that the device is usable.

DO_CLOSE

This work item is used to close a file that backed a device that has been disconnected. Since the driver opened the file with **ZwCreateFile**, its closes the handle by calling **ZwClose** and dereferences the File Object that was returned by **ObReferenceObjectByHandle**.

DO_READ and DO_WRITE

These work items are used to read or write information to or from the file that backs the device. The code in **OsrUserHandleSrb** has converted the **SCSIOP_READ** or **SCSIOP_WRITE** logical block offset and block count into a file offset and byte count that the **DO_READ** or **DO_WRITE** code can use for the operation. Therefore all the **DO_READ** or **DO_WRITE** code has to do is issue a read/write to the underlying file system. Since the driver has the file object for the file and an MDL for the user data, the driver implements the read, via the **DoRead** function and the write via the **DoWrite** function. These functions build an IRP for the read or write operation, call the underlying file system directly, and wait for the response. Once the operation completes the driver code can call **OsrSpCompleteSrb** in the upper-edge of the driver to notify Storport that the SRB has completed.

Wrapping Up Processing

As you've seen, working with Storport to create a Virtual Storport Miniport is actually quite easy. We've covered how to create a device, notify Storport that a device has arrived, process I/Os to the device and make that device go away.

We've also covered, in detail, the process of actually performing I/O operations. This process is one of receiving SRB/CDB pairs, interpreting them, and then mapping them into some operation for the device the driver is emulating. Whether the driver is going to map operations to a file or some other target, with a virtual Storport Miniport, the concepts we have discussed will help you on your way.

(Continued on page 18)

```
case SCSIOP_READ               : // 0x28
    {
    // We have received a read request.   Process the read.
    ULARGE_INTEGER  startingLbn = {0,0};
    ULONG           readLength = 0;
    PCDB            pReadCdb = (PCDB) &PSrb->Cdb[0];
    ULONG           bytesRead;
    ULONG           numBlocks;
    ULONG           bytesPerBlock;

    __try {
        OsrUserGetDiskCapacity(pIInfo,&numBlocks,&bytesPerBlock);
        // Convert the starting LBN back to little endian.
        // Convert the LBN to a byte offset instead of a block offset.
        REVERSE_BYTES(&startingLbn.LowPart,&pReadCdb->CDB10.LogicalBlockByte0)
        startingLbn.QuadPart *= bytesPerBlock;
        // Convert the read length back to little endian
        REVERSE_2BYTES(&readLength,&pReadCdb->CDB10.TransferBlocksMsb);
        readLength *= bytesPerBlock;
        // Issue the read to ScsiPortUser.
        PMDL readMdl = OsrSpGetSrbMdl(pIInfo->OsrSpLocalHandle,PSrb);
        if(!readMdl) {
            status = STATUS_INSUFFICIENT_RESOURCES;
            PSrb->SrbStatus = SRB_STATUS_ABORTED;
        } else {
            status = OsrUserReadData(pIInfo,PSrb,readMdl,startingLbn,readLength,&bytesRead);
        }
    } __except(EXCEPTION_EXECUTE_HANDLER) {
        status = GetExceptionCode();
    }
    // Oh, the user did not want to complete the request, but pended it.  We'll
    // honor it and get out of here.  It is up to the user to complete the request
    // later on.
    if(status == STATUS_PENDING) {
        return status;
    }
    }
    goto completeRequest;
```

Figure 11 - OsrUserHandleSrb SCSIOP_READ Handling

Virtual Storport...

(Continued from page 17)

So what we have shown through the 3 articles on "Writing a Virtual Storport Miniport Driver" is that the Storport environment is pretty easy to work in once you know the constraints.

Now let's wrap up this series by talking about installing and building the included sample code.

Installation

Well, I suppose that since we now have discussed how the driver works we had better discuss how to install it. In the source code that we provide is the INF file used to install our Virtual Storport Miniport Driver. We're going to assume that you've seen INF files before, so we're not going to explain them. What we are going to do however is note some important points.

- The INF file defines its Class as **SCSIAdapter** which will indicate to the PnP Manager that this INF file is for a SCSI Adapter type of device.
- Since there is no hardware associated with this driver, the INF file indicates to the PnP Manager that this device is "ROOT" enumerated, in other words, the Pnp Manager must create a PDO for this device. This detail is defined in the INFs' "Models" section where it specifies %rootstr%, which equates to "ROOT\OsrSVm"
- The INF file adds this service to the SCSI Miniport LoadOrderGroup

How you actually get Windows to install the software depends on which platform you are on. For Pre-Windows 7 systems, you can use the Add Hardware Manager from the control panel to perform this installation. For Windows 7 however, the Add Hardware Manager is missing from the control panel, so in order to install the driver, you must go to the control panel, Select Administrative Tools, then select Computer Management. When the Computer Management window opens select Device Manager, right click on your computer in the left window pane and then select Add Legacy Hardware.

Building

The sample source code comes with 3 directories which build 2 components. The directories are:

- OsrVmSample - which contains the OSR Virtual Storport Miniport Driver software
- OsrVmSampleMgmt – which contains the Win32 MFC application that manages the Driver
- OsrVmSampleInc – which contains the include files shared by the Driver and the Management code.

The Driver and the Management Application can all be built with the Windows 7 WDK and have been tested on Vista and Win7. The included project contains a "dirs" file which will build both components of the software and it also contains the INF file which can be used to install the software.

Summary

In the three articles on "Writing a Virtual Storport Miniport Driver" we have tried to cover all the important aspects of architecture, design, and implementation of the software. Hopefully with these articles and the downloaded software example, you will be able to fully understand our discussion. Included in the sample is the driver source, INF, and the controlling user mode application.

Sample code to OSR's Virtual Storport Miniport Driver can be downloaded from:

www.osronline.com/article.cfm?article=551.

Pods, Pads...

(Continued from page 3)

But let's say the world has to have the real article. Clones will not do. Well, we're not even doomed at that point. The iPad isn't nearly as closed as most of the doomsayers are making out. I'm sure Apple wants it closed, but there's just no chance that they'll get their way. There were reports of iPad jailbreaks *within minutes* of the iPads release (with convincing displays showing root shell access). Well known jailbreaker *Geohot* (George Hotz) apparently had his iPad running Cydia (an app that lets users download and run applications that are not Apple approved) within just a couple of days.

So, even if the pad/slate form-factor catches on, it's far from certain that we'll all fall into line and bleat on our way to the iPad Apps Store to pay big bucks to download some second-rate, Apple approved, shite of an application.

I will grant that the iPad is an interesting idea. Not revolutionary, but interesting. In fact, I will be *very quick* to grant that there are *lots* of interesting devices out there. Given that print media is entering the infamous death spiral, content publishers are eagerly searching for "the next wave" that'll allow effective content distribution (for, ah, money of course). Technology companies are similarly searching for the next form factor, the next "hardware combined with online service" that'll be the new paper and ink. We all agree print is dying; we just haven't figured out what'll take its place.

Will that new content delivery mechanism the iPad? Or is it more like the long awaited (some would say, over-hyped) EInk-based Que Reader by Plastic Logic? Or, will it be something else?

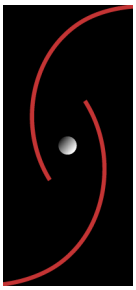
I've had an EInk-based ebook reader (a Cybook G3 by the French firm Bookeen, a very good a device) for over a year now, and I positively love reading from it. And, I've been lusting after the device from Plastic Logic for ages now. The Que Reader is basically an 8.5"x11" ebook reader, backed with a service offering (of course), that'll let you subscribe to newspapers, magazines, and *buy* ebooks to read on the plane.

I don't know about you, but I don't want to read the New York Times or a novel on an LCD display. And that means I don't want to read these things on an iPad. I spend all day long staring at LCD displays. I don't need to bask in its back-lit glow at home, at night, while I'm getting my Stieg Larsson on. So, for me, a big freakin' reflective EInk display trumps the transmissive iPad display any day.

Oh, and while we're talking about technology companies and their eager search for new devices, shall we talk about the Kin? The Kin is the Microsoft phone that is cloud connected. Another device that comes with a service. No, you can't read a book on it. But you can text, and talk, and update your Facebook page, and keep track of what's on your friend's Facebook pages. "The more you share, the more you get" it says on the Kin web site. Wow. OK, never mind. Let's not talk about the Kin.

Plastic Logic people, if you read this maybe you'll send me one so I can review it and say really nice things about it (you know, not that my opinion can be bought or anything, *but*...).

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



OSR OPEN SYSTEMS RESOURCES, INC.
105 State Route 101A, Suite 19
Amherst, New Hampshire 03031 USA
(603)595-6500 ♦ Fax (603)595-6503

RETURN SERVICE REQUESTED

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

First Class

The NT Insider™ is a subscription-based publication

New OSR Seminar Schedule!

Seminar	Dates	Location
Writing WDF Drivers (Lab)	17-21 May	Boston, MA
Writing WDM Drivers (Lab)	17-21 May	Reading, UK
Developing File Systems	24-27 May	Boston, MA
Internals and Software Drivers (Lab)	12-16 July	Washington, D.C.
Kernel Debugging & Crash Analysis (Lab)	19-23 July	Boston, MA
Writing WDM Drivers (Lab)	16-20 August	Seattle, WA

15% Off All OSR Public Seminars, July & August 2010

You read that right. ALL registrations for OSR public seminars held in July and August 2010 will receive 15% off. (may not be combined with other discounts) Contact an OSR seminar coordinator and mention the 15% discount. Email: seminars@osr.com or +1.603.595.6500

Course outlines, pricing, and how to register at: www.osr.com/schedule