Kernel Debugging with WinDbg

The debuggers in Debugging Tools for Windows are powerful, but they have a steep learning curve. This is particularly true for WinDbg and KD, the kernel debuggers used by driver developers (CDB and NTSD are user-space debuggers). The aim of this tutorial is to give a developer experienced with other debuggers enough information to get launched into kernel debugging and to use the Debugging Tools for Windows help file as a reference. The developer is assumed to be familiar with the general concepts of the Windows operating system and the build process.

The focus will be mainly on WinDbg, a kernel-mode and user-mode debugger with a graphical interface. KD is more useful for scripts and automated debugging and enjoys the reputation of being the tool of choice of the most serious programmers, but this tutorial will focus on WinDbg and will merely allude to KD from time to time.

The target operating system is Windows 2000 or later. Much of what is described here works for Windows NT 4.0, too. Furthermore, the target computer's processor uses x86 architecture. While much here will work for 64-bit targets, no specific attention will be given them.

In overview, the tutorial begins with a brief description of setting up for debugging. The bulk of the tutorial is two sections, fundamentals and selected techniques. Fundamentals are the basic and most-often used debugger commands. Selected techniques are those other commands and investigative approaches that will be useful in many situations. This latter section is not, however, an exploration of substantive techniques that might be used to investigate things like deadlocks, memory corruption or resource leaks. On first reading of the tutorial, you may want to skip the selected techniques. The tutorial concludes by pointing to the Microsoft debugger discussion group and the debugger feedback e-mail address for further questions.

Getting set up

Get the latest!

Obtain the latest debugger, and update it regularly. The value of having the latest release can hardly be overstated, because the debugger enjoys frequent improvements and fixes. The debugger can be downloaded at http://www.microsoft.com/whdc/devtools/debugging/default.mspx.

Host and target connection

The debugging setup will be a two computer arrangement, connected by a null-modem cable or by a 1394 cable. This tutorial does not examine "local" debugging on a single system (where the debugger examines the computer it is running on). Three computer debugging (target, debugging server and debugging client) will be briefly discussed.

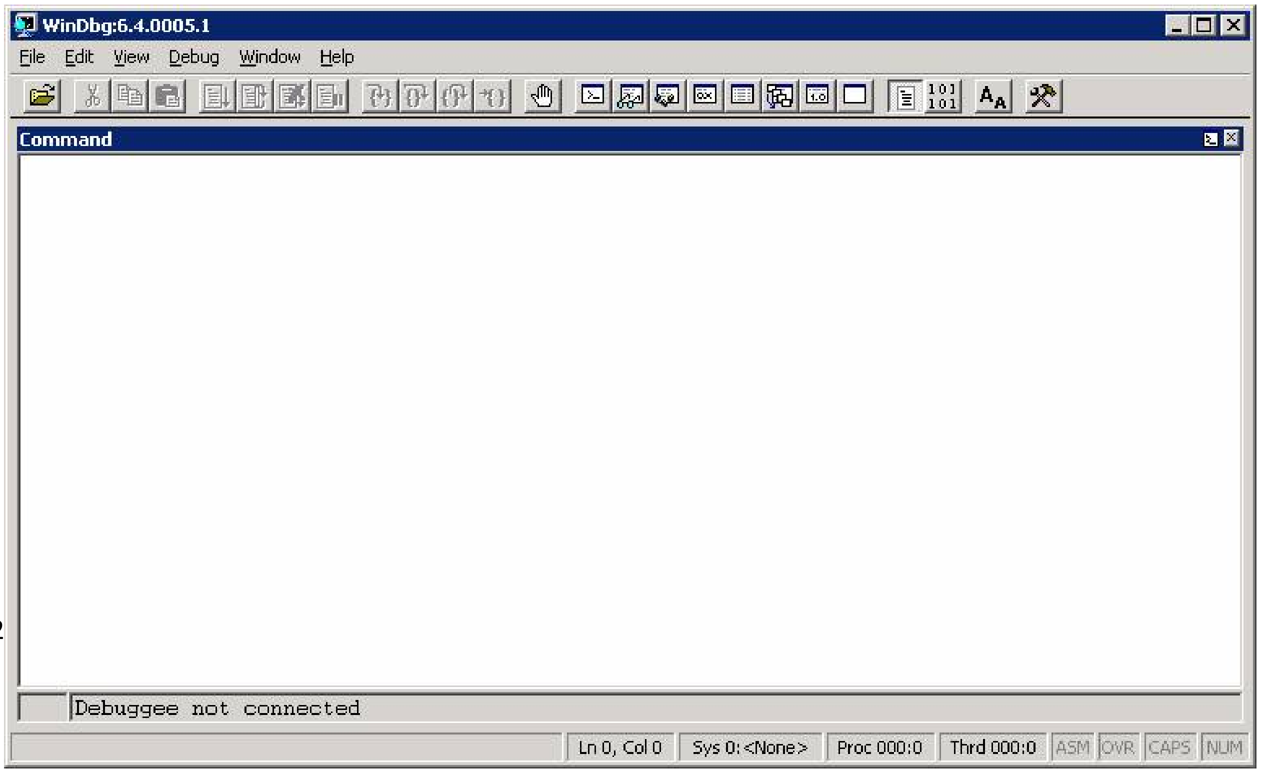
A debugging session is a cooperative process between the host-side debugging application (WinDbg or KD) and the target operating system; each party must do something. More specifically, WinDbg is not a "hypervisor operating system" that runs the target as a guest and is a real operating system in its own right. WinDbg is a debugging application in partnership with a target operating system that is aware of its role in the debugging process. In that partnership, the target sends information to and receives information from WinDbg. The communication mechanism must be good, and it must be efficient.

A serial protocol is the tried-and-true mechanism for communication between the debugger application and the target system. You connect the host and target machines with a null-modem cable at a COM port at each end. An alternative communication mechanism is 1394. The Debugging Tools for Windows help file describes each of these and how to configure the target system in the topic "Configuring Software on the Target Computer."

Your first session

Your host computer is assumed to be running Windows 2000 or later. The host operating system can be a different version of Windows than the target operating system. The host computer may be where you do your usual development, maintenance or troubleshooting. It should be connected to the network if you want access to symbol and source servers (see symbols and source).

From a Command Prompt window, change the current directory to the installation directory of Debugging Tools for Windows. This is the directory in which windbg.exe and kd.exe are



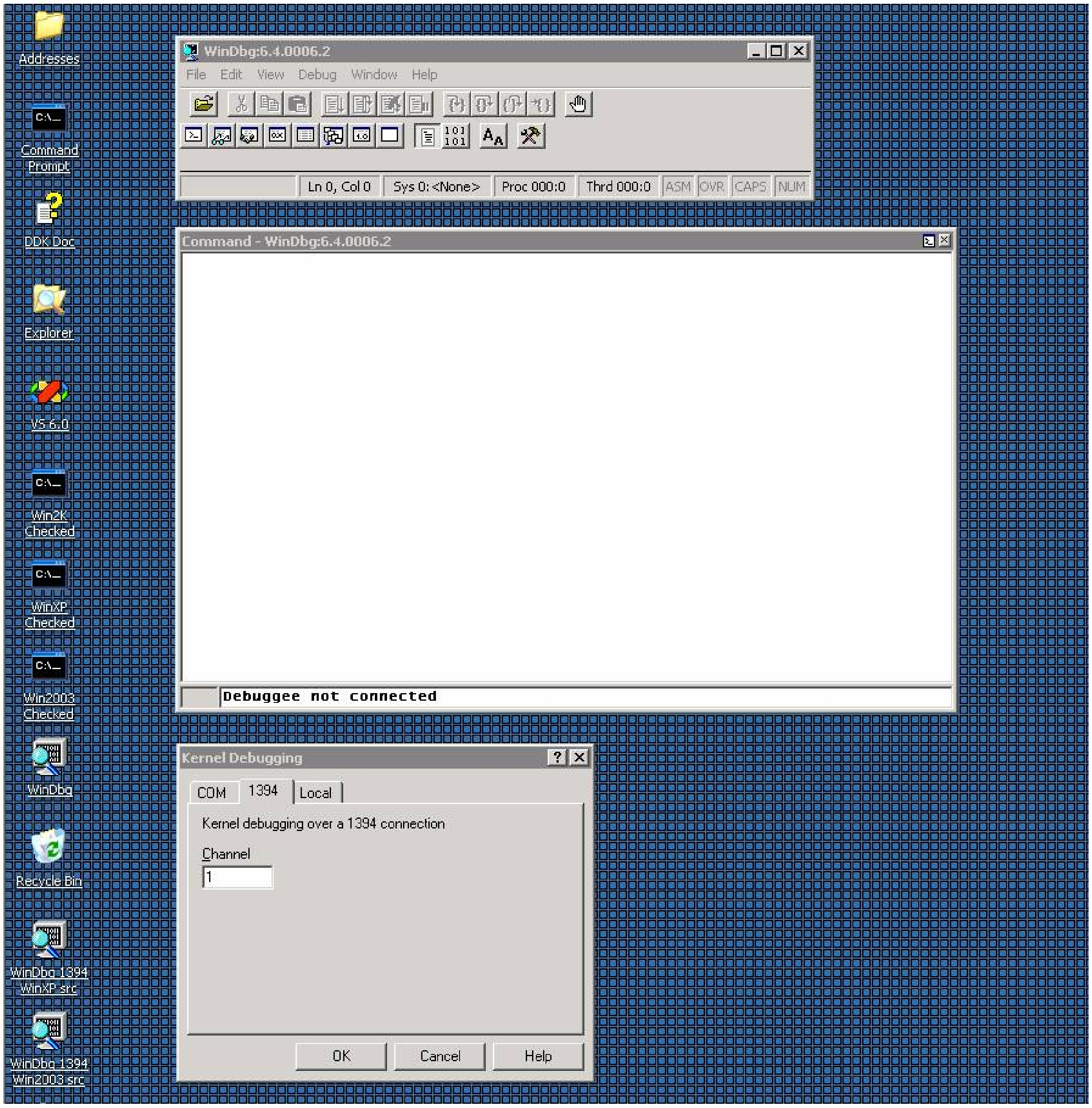
```
Debuggee not connected
```

located. Type **windbg** and press Enter. You should see this:

Windowing

At this point you can arrange your windows. The following example involves floating windows. Start with the combined window that is the first above. Click on the bar labeled "Command" and drag that bar and its window away from the main frame. Then shrink the main frame, since you can use keystrokes instead of directly using the menu or the buttons.

Then use **File→Kernel Debug** to get the protocol popup, and choose 1394 with channel 1. At this point, your desktop looks like this:



```
Debuggee not connected
```

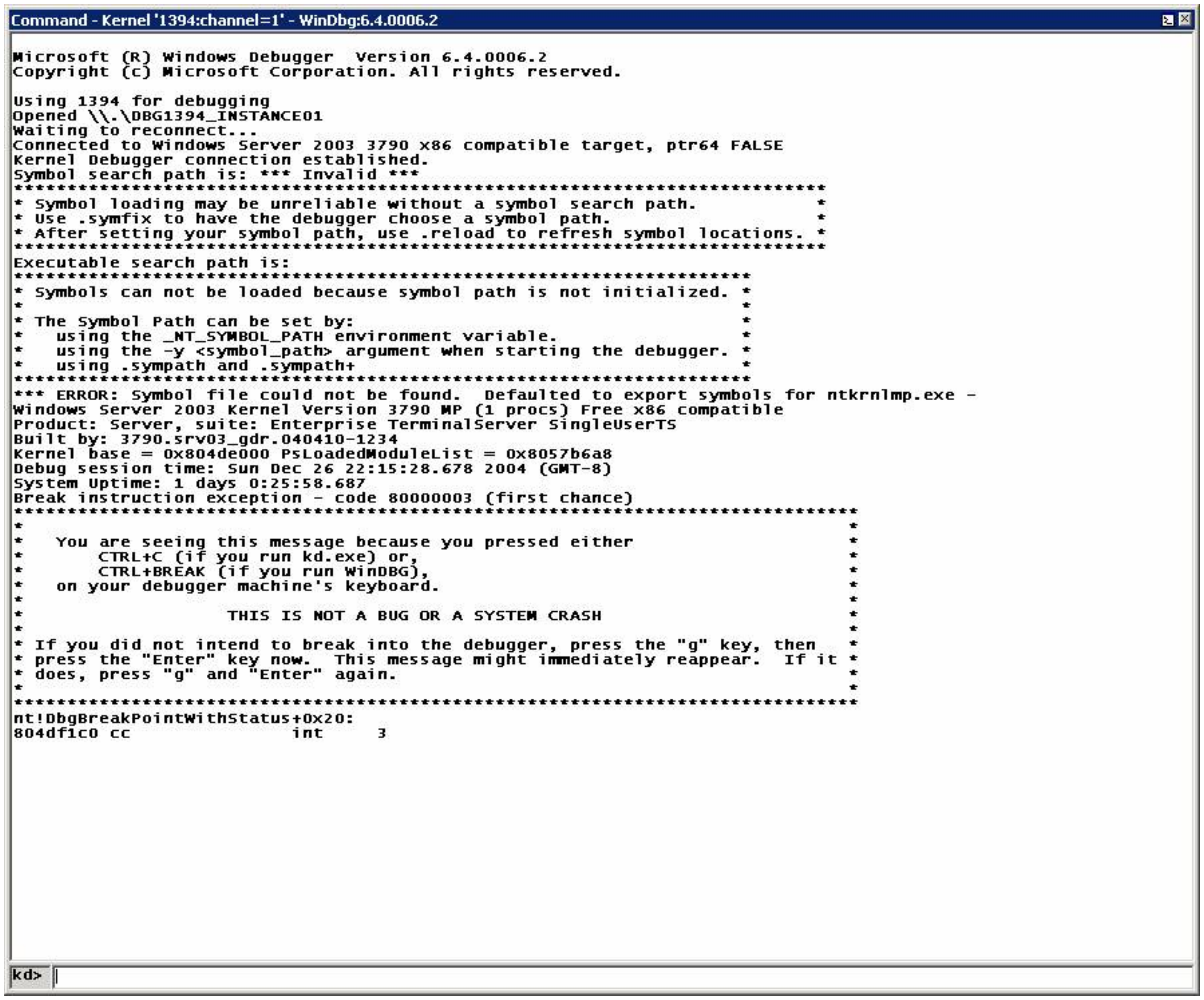
Kernel debugging over a 1394 connection

Channel: 1

Then click OK in the Kernel Debugging window.

Making the connection active

Now you are ready to make a connection between host and target. Go to the target machine and boot Windows from one of the debugging entries. Immediately go back to the host system, touch the WinDbg command window with the cursor to make it active, and press CTRL+BREAK. In a few seconds you should see this:



Command - Kernel '1394:channel=1' - WinDbg:6.4.0006.2

```
Microsoft (R) Windows Debugger  Version 6.4.0006.2
Copyright (c) Microsoft Corporation. All rights reserved.

Using 1394 for debugging
Opened \\.\DBG1394_INSTANCE01
Waiting to reconnect...
Connected to Windows Server 2003 3790 x86 compatible target, ptr64 FALSE
Kernel Debugger connection established.
Symbol search path is: *** Invalid ***
****************************************************************************
* Symbol loading may be unreliable without a symbol search path.           *
* Use .symfix to have the debugger choose a symbol path.                   *
* After setting your symbol path, use .reload to refresh symbol locations. *
****************************************************************************
Executable search path is:
*********************************************************************
* Symbols can not be loaded because symbol path is not initialized. *
*                                                                   *
* The Symbol Path can be set by:                                    *
*   using the _NT_SYMBOL_PATH environment variable.                 *
*   using the -y <symbol_path> argument when starting the debugger. *
*   using .sympath and .sympath+                                    *
*********************************************************************
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for ntkrnlmp.exe -
Windows Server 2003 Kernel Version 3790 MP (1 procs) Free x86 compatible
Product: Server, suite: Enterprise TerminalServer SingleUserTS
Built by: 3790.srv03_gdr.040410-1234
Kernel base = 0x804de000 PsLoadedModuleList = 0x8057b6a8
Debug session time: Sun Dec 26 22:15:28.678 2004 (GMT-8)
System Uptime: 1 days 0:25:58.687
Break instruction exception - code 80000003 (first chance)
*******************************************************************************
*                                                                             *
*   You are seeing this message because you pressed either                    *
*       CTRL+C (if you run kd.exe) or,                                        *
*       CTRL+BREAK (if you run WinDBG),                                       *
*   on your debugger machine's keyboard.                                      *
*                                                                             *
*                   THIS IS NOT A BUG OR A SYSTEM CRASH                       *
*                                                                             *
* If you did not intend to break into the debugger, press the "g" key, then   *
* press the "Enter" key now.  This message might immediately reappear.  If it *
* does, press "g" and "Enter" again.                                          *
*                                                                             *
*******************************************************************************
nt!DbgBreakPointWithStatus+0x20:
804df1c0 cc               int     3
```

```
kd>
```

Don't worry for now about the messages concerning symbols. You have a WinDbg connection to a Windows Server 2003 system. You're in business!

A small but crucial thing to understand: The area at the very bottom of the command window shows the "kd>" prompt. That means that WinDbg is ready to accept commands there. If no prompt is displayed, WinDbg cannot process commands at this moment, although any commands you type will be stored in a buffer and executed as soon as possible. You must wait for "kd>" to appear to be sure WinDbg is ready to respond. It is a fact of WinDbg's life that sometimes it is busy doing something you cannot see (such as getting information from the target, and the information can be voluminous). The absence of "kd>" is your only clue that WinDbg is busy. One possibility is that WinDbg is working to resolve a symbol and taking longer than you might expect. Unfortunately, there is also the occasional situation where WinDbg is waiting for something such as connection with a target that will never respond (maybe boot.ini was configured badly, or the wrong option was chosen). You will have to decide when enough time has passed to resort to drastic measures like pressing CTRL+BREAK or, conceivably, stopping WinDbg and starting a new instance.

Finding symbols and source

By now you are probably eager to start debugging, but there are a few more things you should do, since they will improve your debugging experience enormously.

The first thing is to ensure that WinDbg can find the symbols for a module of interest. Symbols indicate to what statement a binary instruction corresponds and what are the variables in force. Symbols map, in other words. You have available the symbols and the source files for your own modules, if they are in the same place as they were at build time. But what if you need to step through some other code that may have been built long before now? Or, for that matter, what if your code isn't in the same location where it was built?

To explicitly set a location for symbols, use the **.sympath** command. Break (CTRL-BREAK) in the command window and type

```
.sympath
SRV*<DownstreamStore>*http://msdl.microsoft.com/download/symbols
```

to tell WinDbg to look for symbols on the Microsoft public symbols server. To get WinDbg to use that server and to keep a copy of downloaded symbols in a local store, for example, in D:\DebugSymbols, you would do:

```
.sympath SRV*d:\DebugSymbols*http://msdl.microsoft.com/download/symbols
```

Occasionally you may have trouble in getting symbols from the symbols server. In such a case, begin with the **!sym noisy** command to get more information about what WinDbg is trying to do to obtain symbols. Next, use **!lmi** to see what WinDbg knows about the one essential Windows module, *ntoskrnl*. Then try to get symbols for *ntoskrnl*, using **.reload /f**. Thus:

```
kd> !sym noisy
```

```
noisy mode - symbol prompts on


kd> !lmi nt

Loaded Module Info: [nt]

         Module: ntoskrnl

   Base Address: 80a02000

     Image Name: ntoskrnl.exe

   Machine Type: 332 (I386)

     Time Stamp: 3e80048b Mon Mar 24 23:26:03 2003

           Size: 4d8000

       CheckSum: 3f6f03

Characteristics: 10e

Debug Data Dirs: Type  Size     VA  Pointer

             CODEVIEW    25,  ee00,    e600 RSDS - GUID: (0xec9b7590,
0xd1bb, 0x47a6, 0xa6, 0xd5, 0x38, 0x35, 0x38, 0xc2, 0xb3, 0x1a)

               Age: 1, Pdb: ntoskrnl.pdb

     Image Type: MEMORY   - Image read successfully from loaded memory.

    Symbol Type: EXPORT   - PDB not found

    Load Report: export symbols
```

The syntax of the commands used here is described in the Debugging Tools for Windows help file.

Exported symbols are usually pretty meager. Debugging Tools for Windows includes a symbol server that can connect to a public symbol store on Microsoft's internet site. Add this to your symbol path, and then load symbols:

```
kd> .sympath
SRV*d:\DebugSymbols*http://msdl.microsoft.com/download/symbols

Symbol search path is: SRV*d:\ DebugSymbols
*http://msdl.microsoft.com/download/symbols

kd> .reload /f nt
```

```
SYMSRV:
\\symbols\symbols\ntoskrnl.pdb\EC9B7590D1BB47A6A6D5383538C2B31A1\file.p
tr

SYMSRV:  ntoskrnl.pdb from \\symbols\symbols: 9620480 bytes copied

DBGHELP: nt - public symbols


d:\DebugSymbols\ntoskrnl.pdb\EC9B7590D1BB47A6A6D5383538C2B31A1\ntoskrnl
.pdb

kd> !lmi nt

Loaded Module Info: [nt]

         Module: ntoskrnl

   Base Address: 80a02000

     Image Name: ntoskrnl.exe

   Machine Type: 332 (I386)

     Time Stamp: 3e80048b Mon Mar 24 23:26:03 2003

           Size: 4d8000

       CheckSum: 3f6f03

Characteristics: 10e

Debug Data Dirs: Type  Size     VA  Pointer

             CODEVIEW    25,  ee00,    e600 RSDS - GUID: (0xec9b7590,
0xd1bb, 0x47a6, 0xa6, 0xd5, 0x38, 0x35, 0x38, 0xc2, 0xb3, 0x1a)

               Age: 1, Pdb: ntoskrnl.pdb

     Image Type: MEMORY   - Image read successfully from loaded memory.

    Symbol Type: PDB      - Symbols loaded successfully from symbol
server.


d:\DebugSymbols\ntoskrnl.pdb\EC9B7590D1BB47A6A6D5383538C2B31A1\ntoskrnl
.pdb

       Compiler: C - front end [13.10 bld 2179] - back end [13.10 bld
2190]

    Load Report: public symbols
```

```
d:\DebugSymbols\ntoskrnl.pdb\EC9B7590D1BB47A6A6D5383538C2B31A1\ntoskrnl
.pdb
```

While symbols will give you some information, they do not provide source code. In the simplest case, source files will be found in the same place they were at build time (the location will be in the binary and symbol files). But in many cases, they cannot be found there (they may have been moved), and you must specify where to look. For that you need a source path, for example,

```
.srcpath e:\Win2003SP1
```

That means: For source files, look in the e:\Win2003SP1 directory.

Another solution is to name a source server, if you have one:

```
.srcpath \\MySrcServer
```

If you experience trouble in getting source files, you should do **.srcnoisy 1** to get more information about what the debugger is doing to find them.

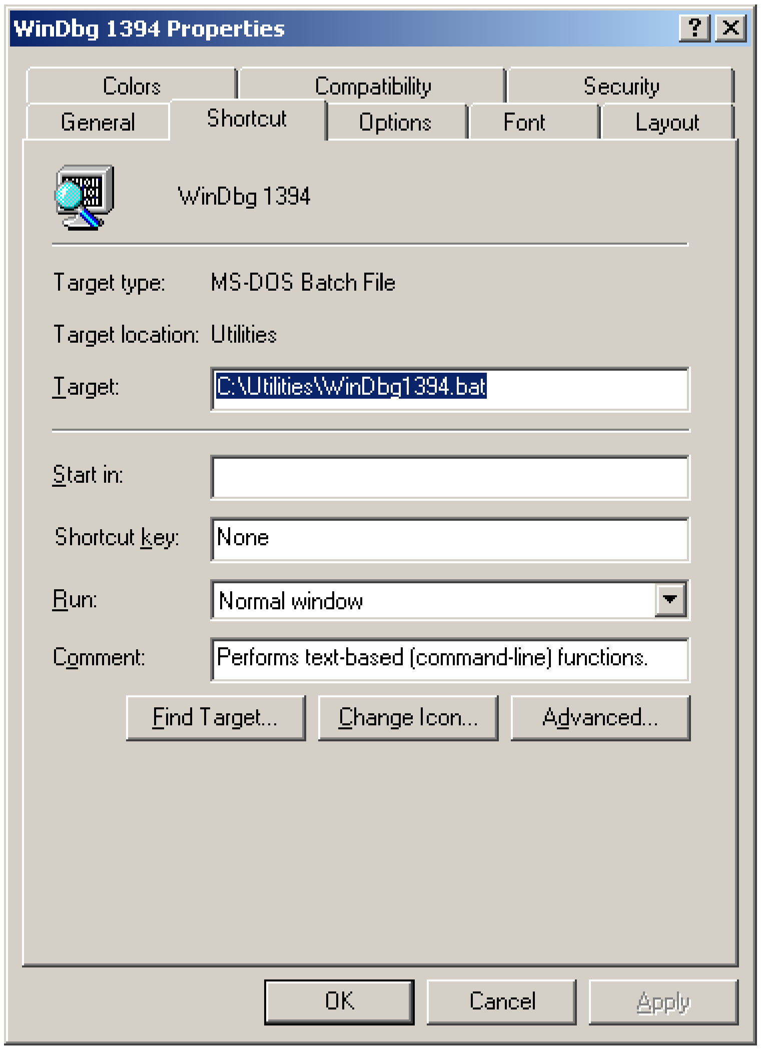
Workspaces

You've not begun actual debugging, yet you have done a good deal of typing already. A lot of the settings can be kept in a workspace. So you might use **File→Save** to save the settings in a workspace, perhaps one you name kernel1394Win2003. After that, you could start WinDbg with that workspace:

```
windbg -W kernel1394Win2003 -k 1394:channel=1
```

where **–W** specifies a workspace and **–k** gives the communication protocol (refer to the Debugging Tools for Windows help file under "WinDbg Command-Line Options"). Note: You should be careful to preserve lower and upper case in the command-line options you give to WinDbg or KD.

To make things easier, you can put a shortcut on your desktop to start WinDbg with the desired workspace, for example, with a 1394 connection:



WinDbg 1394

Target type: MS-DOS Batch File

Target location: Utilities

Target: C:\Utilities\WinDbg1394.bat

Start in:

Shortcut key: None

Run: Normal window

Comment: Performs text-based (command-line) functions.

The contents of the file above are:

```
cd /d "d:\Program Files\Debugging Tools for Windows"
start windbg.exe -y
SRV*d:\DebugSymbols*http://msdl.microsoft.com/download/symbols -W
kernel1394Win2003
```

The first line makes the Debugging Tools for Windows installation directory the current directory, to ensure that debugger modules will be found. The second line starts WinDbg, specifying a symbol path (**-y**) and workspace (**-W**).

A sample driver

It will be helpful to exercise WinDbg with the sample driver IoCtl. This can be found in the Windows Device Driver Kit (DDK) and its successor, the Windows Driver Kit (WDK). Install this kit and look in the *src\general\Ioctl* subdirectory. The advantages of IoCtl are that it is simple and that it is a "legacy" driver, that is, one loaded by the Service Control Manager (SCM) and not a part of Plug-and-Play (the in's and out's of PnP are not of interest here).

You should build both the user-space executable (ioctlapp.exe) and the kernel-space driver (sioctl.sys), since the former will load the latter.

Here is something fairly important to understand. The build process is quite smart about optimizing code, and optimization can result in code movement (logic is of course preserved) and in keeping variable values solely in registers. To ensure a straightforward debugging experience, you should produce a checked build with this compiler directive given in the build window or in the appropriate sources file:

```
MSC_OPTIMIZATION=/Od
```

(That is "Oh d" and not "zero d.")

Sometimes the above will result in a build problem with intrinsic functions like memcmp. If you run into that problem, try:

```
MSC_OPTIMIZATION=/Odi
```

Please understand that preventing optimization is not a good choice for a production build. With the above directive, you would not be creating and would not be testing a production-type build. It is nonetheless a good practice to *start* testing with non-optimized builds and, once you are familiar with the code and have eliminated the simpler errors, to advance to production builds. When you have to deal with optimized code, you will find some assistance in dealing with optimized code.

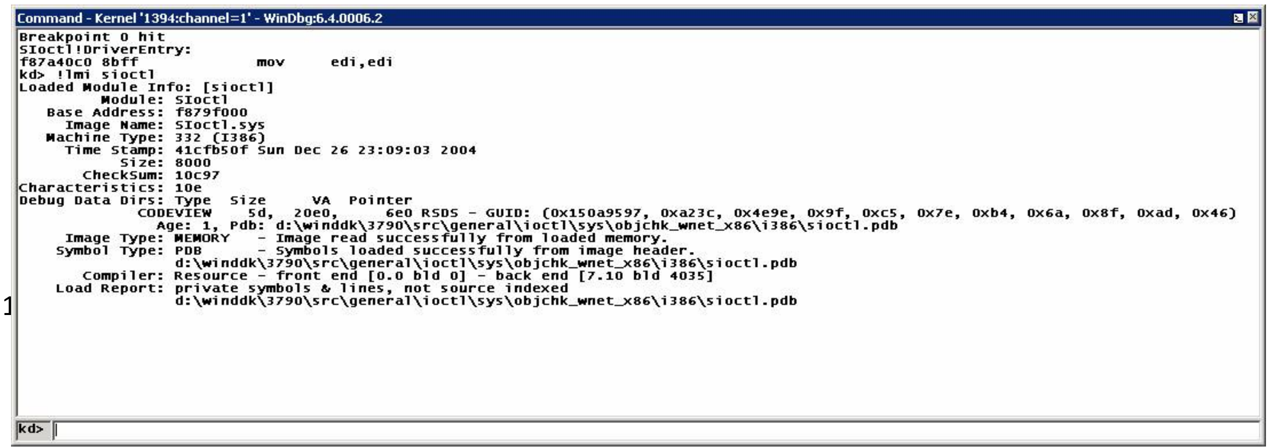
Starting to debug the sample driver

Set a breakpoint in IoCtl at DriverEntry. Before starting the driver, break into the WinDbg command window and type this:

```
bu sioctl!DriverEntry
```

The **bu** ("Breakpoint Unresolved") command defers the actual setting of the breakpoint until the module is loaded; at that point, WinDbg will look in it for "DriverEntry." Since there is nothing more to do now, hit F5 (or you could type **g**, "Go").

Next, copy ioctlapp.exe and sioctl.sys to a place on the target system such as C:\Temp\IOCTL, log in with Administrator privileges on that system, and make C:\Temp\IOCTL the current directory in a command window there. (You need not put this

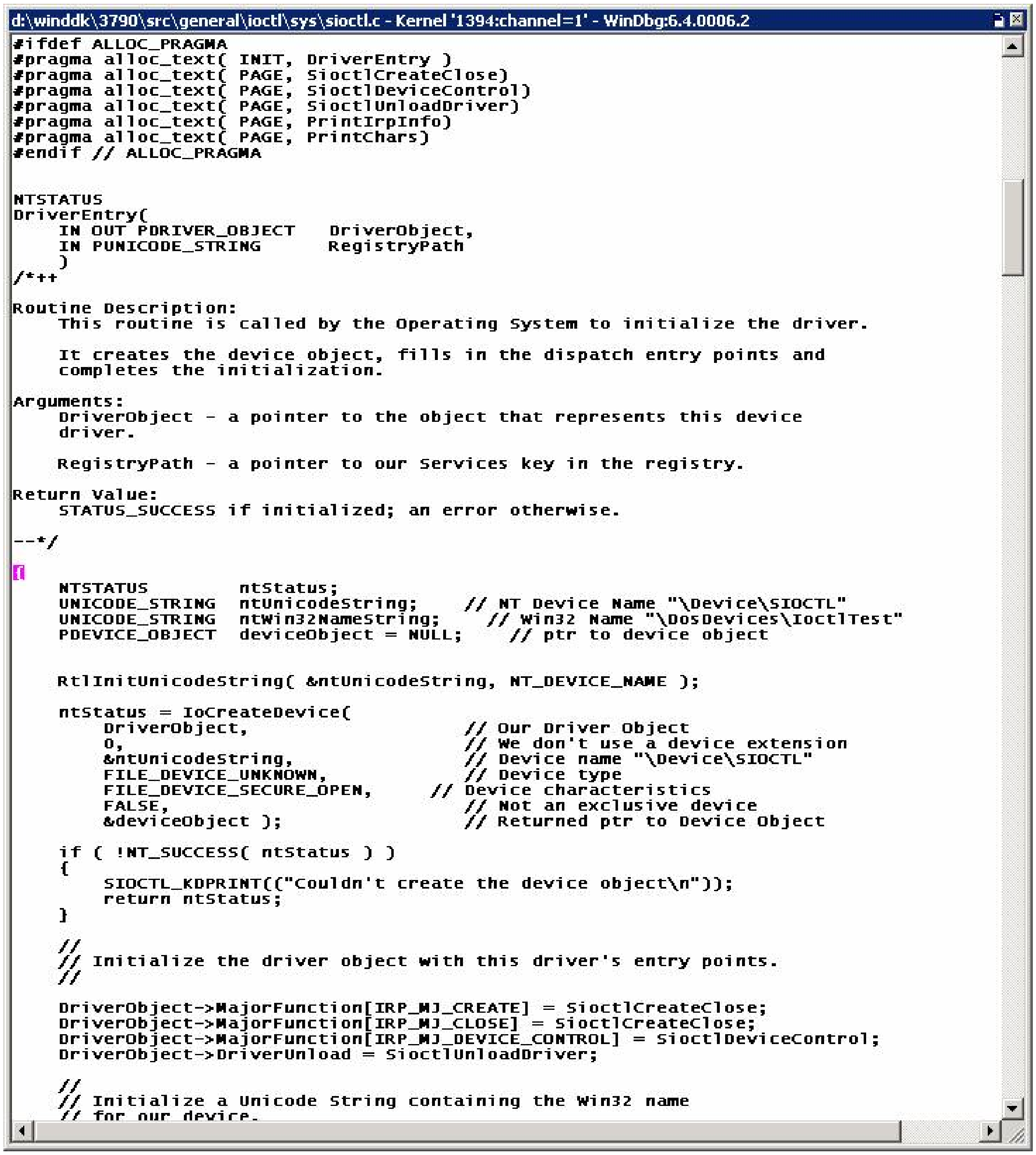


```
Breakpoint 0 hit
SIoctl!DriverEntry:
f87a40c0 8bff            mov     edi,edi
kd> !lmi sioctl
Loaded Module Info: [sioctl]
         Module: SIoctl
   Base Address: f879f000
     Image Name: SIoctl.sys
   Machine Type: 332 (I386)
     Time Stamp: 41cfb50f Sun Dec 26 23:09:03 2004
           Size: 8000
       CheckSum: 10c97
Characteristics: 10e
Debug Data Dirs: Type  Size     VA  Pointer
             CODEVIEW    5d,  20e0,     6e0 RSDS - GUID: (0x150a9597, 0xa23c, 0x4e9e, 0x9f, 0xc5, 0x7e, 0xb4, 0x6a, 0x8f, 0xad, 0x46)
               Age: 1, Pdb: d:\winddk\3790\src\general\ioctl\sys\objchk_wnet_x86\i386\sioctl.pdb
     Image Type: MEMORY   - Image read successfully from loaded memory.
    Symbol Type: PDB      - Symbols loaded successfully from image header.
                 d:\winddk\3790\src\general\ioctl\sys\objchk_wnet_x86\i386\sioctl.pdb
       Compiler: Resource - front end [0.0 bld 0] - back end [7.10 bld 4035]
    Load Report: private symbols & lines, not source indexed
                 d:\winddk\3790\src\general\ioctl\sys\objchk_wnet_x86\i386\sioctl.pdb
```

path into a symbol path or source path in WinDbg.) In that same command window, type **ioctlapp** and press Enter; in WinDbg you will see:

In the above, after execution stopped at the breakpoint, the **!lmi** command showed that WinDbg did pick up the symbols from a DDK build. The timestamp is about what you would expect, and the location of the symbol file is exactly that expected.

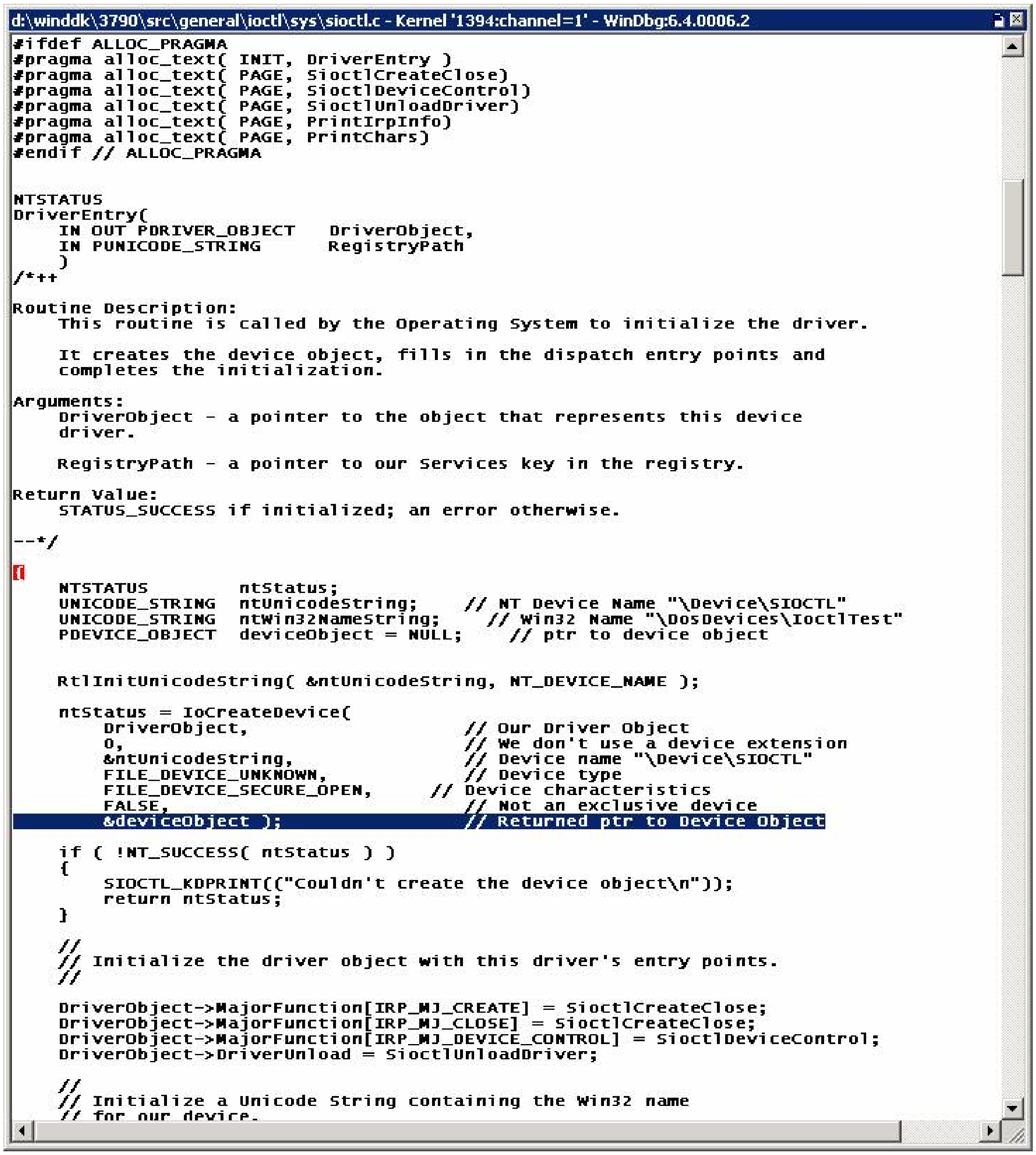
Depending on your docking arrangement, it may not be obvious, since windows can be hidden by other windows, but you will have a source code window somewhere (the key sequence ‘alt-Keypad *’ — without single quotation marks — will bring that window to the front):



d:\winddk\3790\src\general\ioctl\sys\sioctl.c - Kernel '1394:channel=1' - WinDbg:6.4.0006.2

```
#ifdef ALLOC_PRAGMA
#pragma alloc_text( INIT, DriverEntry )
#pragma alloc_text( PAGE, SioctlCreateClose)
#pragma alloc_text( PAGE, SioctlDeviceControl)
#pragma alloc_text( PAGE, SioctlUnloadDriver)
#pragma alloc_text( PAGE, PrintIrpInfo)
#pragma alloc_text( PAGE, PrintChars)
#endif // ALLOC_PRAGMA


NTSTATUS
DriverEntry(
    IN OUT PDRIVER_OBJECT   DriverObject,
    IN PUNICODE_STRING      RegistryPath
    )
/*++

Routine Description:
    This routine is called by the Operating System to initialize the driver.

    It creates the device object, fills in the dispatch entry points and
    completes the initialization.

Arguments:
    DriverObject - a pointer to the object that represents this device
    driver.

    RegistryPath - a pointer to our Services key in the registry.

Return Value:
    STATUS_SUCCESS if initialized; an error otherwise.

--*/

{
    NTSTATUS        ntStatus;
    UNICODE_STRING  ntUnicodeString;    // NT Device Name "\Device\SIOCTL"
    UNICODE_STRING  ntWin32NameString;    // Win32 Name "\DosDevices\IoctlTest"
    PDEVICE_OBJECT  deviceObject = NULL;    // ptr to device object


    RtlInitUnicodeString( &ntUnicodeString, NT_DEVICE_NAME );

    ntStatus = IoCreateDevice(
        DriverObject,                   // Our Driver Object
        0,                              // We don't use a device extension
        &ntUnicodeString,               // Device name "\Device\SIOCTL"
        FILE_DEVICE_UNKNOWN,            // Device type
        FILE_DEVICE_SECURE_OPEN,     // Device characteristics
        FALSE,                          // Not an exclusive device
        &deviceObject );                // Returned ptr to Device Object

    if ( !NT_SUCCESS( ntStatus ) )
    {
        SIOCTL_KDPRINT(("Couldn't create the device object\n"));
        return ntStatus;
    }

    //
    // Initialize the driver object with this driver's entry points.
    //

    DriverObject->MajorFunction[IRP_MJ_CREATE] = SioctlCreateClose;
    DriverObject->MajorFunction[IRP_MJ_CLOSE] = SioctlCreateClose;
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = SioctlDeviceControl;
    DriverObject->DriverUnload = SioctlUnloadDriver;

    //
    // Initialize a Unicode String containing the Win32 name
    // for our device.
```

The line marked in pink (the Debugging Tools for Windows help file calls it purple) is where the breakpoint was set and is where execution stopped. Have execution proceed down to IoCreateDevice (controlling execution says how this might be accomplished):


d:\winddk\3790\src\general\ioctl\sys\sioctl.c - Kernel '1394:channel=1' - WinDbg:6.4.0006.2

```
#ifdef ALLOC_PRAGMA
#pragma alloc_text( INIT, DriverEntry )
#pragma alloc_text( PAGE, SioctlCreateClose)
#pragma alloc_text( PAGE, SioctlDeviceControl)
#pragma alloc_text( PAGE, SioctlUnloadDriver)
#pragma alloc_text( PAGE, PrintIrpInfo)
#pragma alloc_text( PAGE, PrintChars)
#endif // ALLOC_PRAGMA


NTSTATUS
DriverEntry(
    IN OUT PDRIVER_OBJECT   DriverObject,
    IN PUNICODE_STRING      RegistryPath
    )
/*++

Routine Description:
    This routine is called by the Operating System to initialize the driver.

    It creates the device object, fills in the dispatch entry points and
    completes the initialization.

Arguments:
    DriverObject - a pointer to the object that represents this device
    driver.

    RegistryPath - a pointer to our Services key in the registry.

Return Value:
    STATUS_SUCCESS if initialized; an error otherwise.

--*/
{
    NTSTATUS        ntStatus;
    UNICODE_STRING  ntUnicodeString;    // NT Device Name "\Device\SIOCTL"
    UNICODE_STRING  ntWin32NameString;    // Win32 Name "\DosDevices\IoctlTest"
    PDEVICE_OBJECT  deviceObject = NULL;    // ptr to device object


    RtlInitUnicodeString( &ntUnicodeString, NT_DEVICE_NAME );

    ntStatus = IoCreateDevice(
        DriverObject,                   // Our Driver Object
        0,                              // We don't use a device extension
        &ntUnicodeString,               // Device name "\Device\SIOCTL"
        FILE_DEVICE_UNKNOWN,            // Device type
        FILE_DEVICE_SECURE_OPEN,     // Device characteristics
        FALSE,                          // Not an exclusive device
        &deviceObject );                // Returned ptr to Device Object

    if ( !NT_SUCCESS( ntStatus ) )
    {
        SIOCTL_KDPRINT(("Couldn't create the device object\n"));
        return ntStatus;
    }

    //
    // Initialize the driver object with this driver's entry points.
    //

    DriverObject->MajorFunction[IRP_MJ_CREATE] = SioctlCreateClose;
    DriverObject->MajorFunction[IRP_MJ_CLOSE] = SioctlCreateClose;
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = SioctlDeviceControl;
    DriverObject->DriverUnload = SioctlUnloadDriver;

    //
    // Initialize a Unicode String containing the Win32 name
    // for our device.
```

Here you see again the original breakpoint (highlighted in red, since now control is stopped there), and you see the current statement marked in dark blue.

Fundamentals

That was a “test drive” in a debugging session. Here are the debugger’s basic operations.

Commands, extensions, etc.

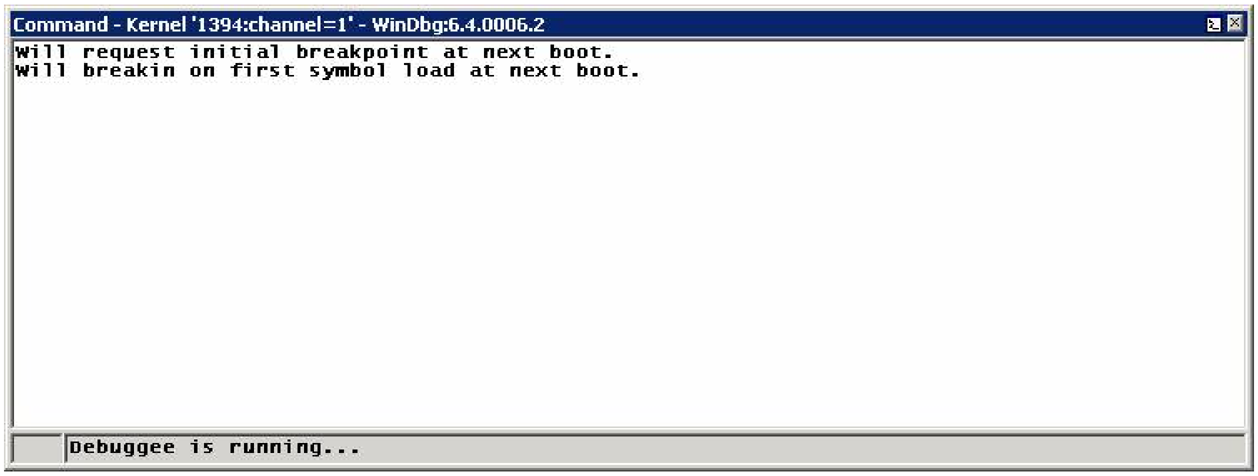
Commands come in several families: Plain (unadorned), ones beginning with a period (“.”) and ones beginning with an exclamation point (“!”). The Debugging Tools for Windows help file describes these families as commands, meta-commands and extension commands, respectively. For present purposes, the families are pretty much the same.

Breakpoints

Causing an interruption in execution is one of the basic capabilities of the debugger. Here are some ways that can be done.

- Breakin at operating system startup.

  To get a break at the earliest point in operating system startup, be sure WinDbg is properly connected, and do the key sequence CTRL-ALT-K repeatedly until you see:



Command - Kernel '1394:channel=1' - WinDbg:6.4.0006.2

```
Will request initial breakpoint at next boot.
Will breakin on first symbol load at next boot.
```

```
Debuggee is running...
```

  At the next boot, the operating system will halt very shortly after *ntoskrnl* has started but before any drivers have been loaded, and WinDbg will get control. This is when you may wish to define breakpoints for drivers to be started at boot time.

- Ordinary breakpoints

The simplest breakpoints are those set via the **bp** ("Breakpoint") command. For example:

```
bp MyDriver!xyz
bp f89adeaa
```

In the first, the breakpoint is at a name in a module (<module>!<name>); in the second, it is at the given address. When execution reaches one of those points, the operating system halts and gives control to WinDbg. (You will see how to get the address for the second command in finding a name.)

Note: The syntax of the first command assumes that the operating system has loaded the module and that enough information is available through a symbol file or through external names to identify *xyz*. If *xyz* cannot be found in the module, the debugger will say so.

- Deferred breakpoints.

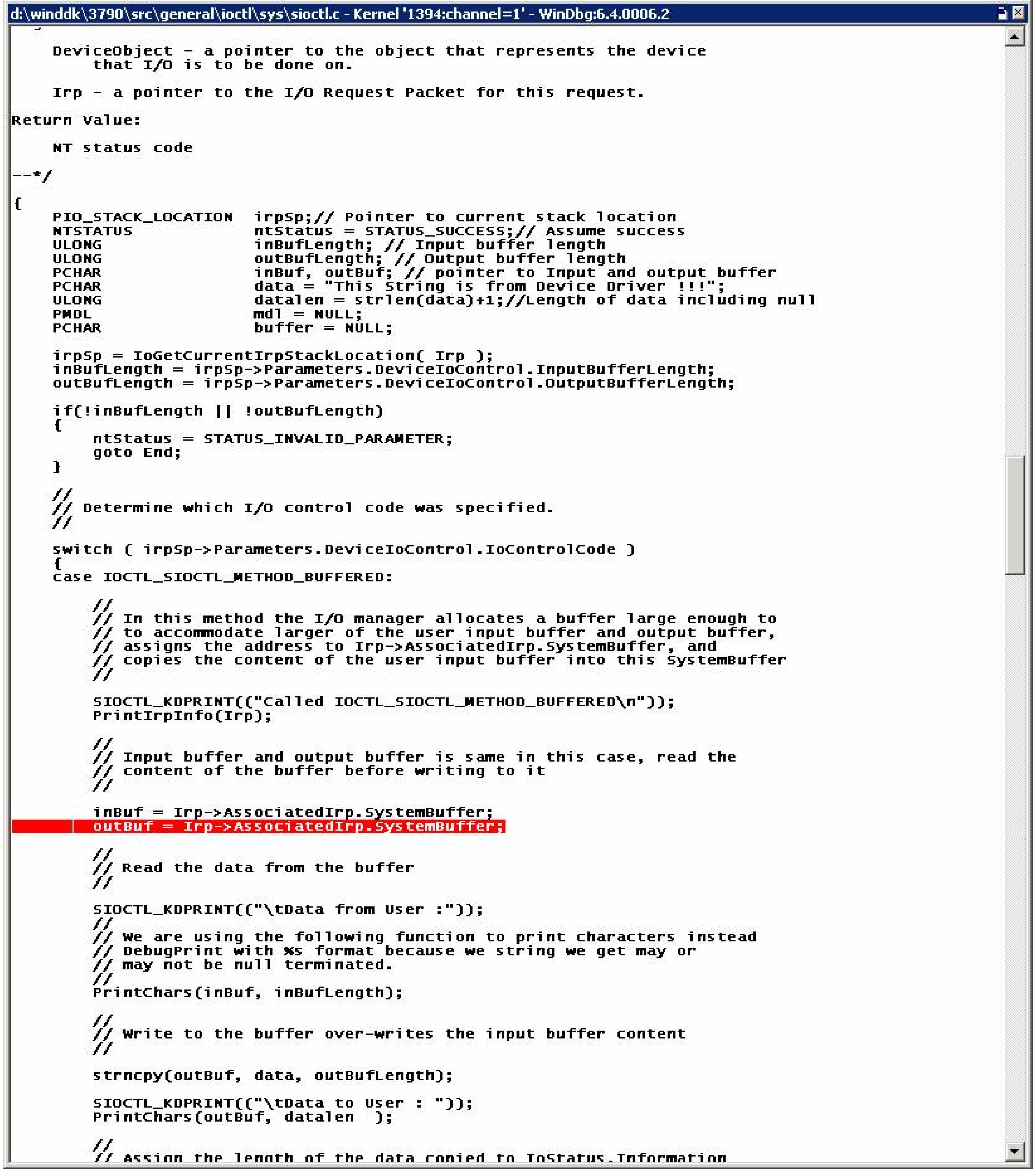
  Speaking of drivers that haven't loaded, your very first breakpoint, set by **bu** (see starting to debug the sample driver above), was a "deferred" breakpoint. The **bu** command takes as a parameter the module and a name in it, for example:

  ```
  bu sioctl!SioctlDeviceControl
  ```

  where SioctlDeviceControl is an entry point or other name in the module sioctl.sys. This form assumes that when the module is loaded, enough information will be available to identify SioctlDeviceControl so that the breakpoint can be set. (If the module is already loaded and the name is found, the breakpoint is set immediately.) If the operating system cannot find SioctlDeviceControl, the debugger will say so, and there will be *no* halt at SioctlDeviceControl.

  A useful characteristic of deferred breakpoints is that they operate on modules!names, whilst ordinary breakpoints operate on addresses or on modules!names immediately resolved into addresses. Another feature of deferred breakpoints is that they are remembered across boots (that wouldn't make sense for address-expressed breakpoints). Yet another characteristic is that a deferred breakpoint is remembered if the associated module is unloaded, whilst an ordinary breakpoint is removed when the module is unloaded.

- Another way to set an ordinary breakpoint is via a source window. Return to sioctl.sys. When you broke at DriverEntry, you could have scrolled the window down to some place where you wished to stop, placed the cursor on the desired line and hit F9:



d:\winddk\3790\src\general\ioctl\sys\sioctl.c - Kernel '1394:channel=1' - WinDbg:6.4.0006.2

```
    DeviceObject - a pointer to the object that represents the device
        that I/O is to be done on.

    Irp - a pointer to the I/O Request Packet for this request.

Return Value:

    NT status code

--*/

{
    PIO_STACK_LOCATION  irpSp;// Pointer to current stack location
    NTSTATUS            ntStatus = STATUS_SUCCESS;// Assume success
    ULONG               inBufLength; // Input buffer length
    ULONG               outBufLength; // Output buffer length
    PCHAR               inBuf, outBuf; // pointer to Input and output buffer
    PCHAR               data = "This String is from Device Driver !!!";
    ULONG               datalen = strlen(data)+1;//Length of data including null
    PMDL                mdl = NULL;
    PCHAR               buffer = NULL;

    irpSp = IoGetCurrentIrpStackLocation( Irp );
    inBufLength = irpSp->Parameters.DeviceIoControl.InputBufferLength;
    outBufLength = irpSp->Parameters.DeviceIoControl.OutputBufferLength;

    if(!inBufLength || !outBufLength)
    {
        ntStatus = STATUS_INVALID_PARAMETER;
        goto End;
    }

    //
    // Determine which I/O control code was specified.
    //

    switch ( irpSp->Parameters.DeviceIoControl.IoControlCode )
    {
    case IOCTL_SIOCTL_METHOD_BUFFERED:

        //
        // In this method the I/O manager allocates a buffer large enough to
        // to accommodate larger of the user input buffer and output buffer,
        // assigns the address to Irp->AssociatedIrp.SystemBuffer, and
        // copies the content of the user input buffer into this SystemBuffer
        //

        SIOCTL_KDPRINT(("Called IOCTL_SIOCTL_METHOD_BUFFERED\n"));
        PrintIrpInfo(Irp);

        //
        // Input buffer and output buffer is same in this case, read the
        // content of the buffer before writing to it
        //

        inBuf = Irp->AssociatedIrp.SystemBuffer;
        outBuf = Irp->AssociatedIrp.SystemBuffer;

        //
        // Read the data from the buffer
        //

        SIOCTL_KDPRINT(("\tData from User :"));
        //
        // We are using the following function to print characters instead
        // DebugPrint with %s format because we string we get may or
        // may not be null terminated.
        //
        PrintChars(inBuf, inBufLength);

        //
        // Write to the buffer over-writes the input buffer content
        //

        strncpy(outBuf, data, outBufLength);

        SIOCTL_KDPRINT(("\tData to User : "));
        PrintChars(outBuf, datalen  );

        //
        // Assign the length of the data copied to IoStatus.Information
```

The line in red is the breakpoint set via F9.

- To see all the breakpoints you've defined, use the **bl** ("Breakpoint List") command:

```
kd> bl

 0 e [d:\winddk\3790\src\general\ioctl\sys\sioctl.c @ 123]
0001 (0001) SIoctl!DriverEntry

 1 e [d:\winddk\3790\src\general\ioctl\sys\sioctl.c @ 338]
0001 (0001) Sioctl!SioctlDeviceControl+0x103
```

Notice a couple of things: Each breakpoint has a number, and breakpoint status is shown, “e” for “enabled” or “d” for “disabled.”

- Now suppose you wish to take a breakpoint temporarily out of operation. **bd** (“Disable Breakpoint”) would do the trick. You have to specify just the breakpoint number:

```
kd> bd 1

kd> bl

 0 e [d:\winddk\3790\src\general\ioctl\sys\sioctl.c @ 123]
0001 (0001) SIoctl!DriverEntry

 1 d [d:\winddk\3790\src\general\ioctl\sys\sioctl.c @ 338]
0001 (0001) SIoctl!SioctlDeviceControl+0x103
```

- In a similar way, to remove breakpoint number 1 permanently, use **bc 1** (“Clear Breakpoint”). Now that breakpoint will be gone from the breakpoint list.
- Well and good. Sometimes, however, a breakpoint is set at a very busy place in the operating system or a driver, and you may want to apply some conditions or conditional actions so that the breakpoint stops only under those circumstances. Here is the basic form:

```
bp SIoctl!SioctlDeviceControl+0x103 "j (@@(Irp)=0xffb5c4f8) '';
'g'"
```

The meaning of this is: Break only if the pointer *Irp* equals the address 0xFFB5C4F8; if that condition is not met, proceed.

To go deeper into the above, the breakpoint is not *itself* conditional. Rather, the breakpoint has an action clause (in double quotation marks); in that clause, the command **j** (“Execute IF/ELSE”) is a conditional action. **j**'s function is to execute its TRUE|FALSE clauses (in single quotation marks). In the above, the TRUE clause (first) is empty, so that when the breakpoint fires and the TRUE condition is met, WinDbg would do nothing more than the default of halting. If the FALSE clause (second) is met, execution would proceed, by executing **g**. One action or the other would be done, depending on the condition in force.

Consider next this slightly more elaborate form of the above:

```
bp SIoctl!SioctlDeviceControl+0x103 "j (@@(Irp)=0xffb5c4f8)
'.echo Found the interesting IRP' ; '.echo Skipping an IRP of no
interest; g' "
```

Here the TRUE clause puts out a message and stops. The FALSE clause puts out a message and continues (a message is helpful because WinDbg would evaluate the condition to FALSE and would otherwise go on silently).

Something to note: The following breakpoint, where register Eax is tested (you will find registers treated more thoroughly in registers), won't work as you might expect:

```
bp SIoctl!SioctlDeviceControl+0x103 "j (@eax=0xffb5c4f8) '.echo
Here!' ; '.echo Skipping; g' "
```

The reason is that evaluation may "sign-extend" the register's value to 64 bits, i.e., to 0xFFFFFFFF`FFB5C4F8, and that would not match 0x00000000`FFB5C4F8. This will be a issue only if the highest bit of the 32-bit value is 1 and if some other conditions (for example, a 32-bit register) apply. "Sign Extension" in the Debugging Tools for Windows help file gives details (see also "Setting a Conditional Breakpoint" there).

A breakpoint may have conditions, with or without conditional actions. One condition is "one-shot" firing: The breakpoint is to fire only once (it is cleared after being hit). This is handy for a very high-traffic piece of code where you are interested in the first hit only.

```
bp /1 SIoctl!SioctlDeviceControl+0x103
```

Another useful condition is a test for a process or thread:

```
bp /p 0x81234000 SIoctl!SioctlDeviceControl+0x103

bp /t 0xff234000 SIoctl!SioctlDeviceControl+0x103
```

which mean, respectively, Stop at the indicated point only if the process block (EPROCESS) is at 0x81234000, and Stop at the indicated point only if the thread block (ETHREAD) is at 0xFF234000.

Conditions can be combined:

```
bp /1 /C 4 /p 0x81234000 SIoctl!SioctlDeviceControl+0x103
```

The meaning here is, Break once but only when the call stack depth is greater than four (here capitalization of "C" is significant, because "c" means "less than") and the process block is at 0x81234000.

- A different kind of breakpoint is one that is specified for *access*. For example,

```
ba w4 0xffb5c4f8+0x18+0x4
```

> The address was taken from the IRP whose address you saw above, and at the offset 0x18+0x4 into the IRP is the IoStatus.Information member. So the breakpoint will fire when something attempts an update of the four bytes constituting IoStatus.Information in that IRP. Such breakpoints are called *data breakpoints* (because they are triggered by data access) or *processor breakpoints* (because they are implemented by the processor, not the debugger itself).

Expressions: MASM versus C++

You will probably agree that it is handy to use a variable in a driver program to provide a parameter value like process address. Doing that, however, demands that you understand something of debugger expressions.

The debugger has two ways of evaluating expressions, referred to "MASM" (Microsoft Macro Assembler) and "C++." To quote the Debugging Tools for Windows help file under "MASM Expressions vs. C++ Expressions":

> In a MASM expression, the numerical value of any symbol is its memory address. In a C++ expression, the numerical value of a variable is its actual value, not its address.

Reading and re-reading that section will repay your time well.

An expression will be evaluated by MASM rules, C++ rules or a combination. In brief,

1. The default expression type is MASM.
2. The default type can be changed by **.expr** (see the Debugging Tools for Windows help file).
3. Certain commands always use C++ evaluation.
4. Evaluation of a specific expression (or part of an expression) can be changed to the form *opposite* to the usual expression type by prefixing the expression with "@@."

Even this summary is fairly involved, and you should refer to "Evaluating Expressions" in the Debugging Tools for Windows help file for the details. For now, here are a few examples to give a sense of how evaluation works.

Earlier you were stopped at Sioctl!SioctlDeviceControl+0x103, so use **dv** to look at a variable known there (see the **dv** command for more information):

```
kd> dv Irp

            Irp = 0xff70fbc0
```

The response means, The variable *Irp* contains 0xFF70FBC0. Further, **dv** is interpreting its parameter in C++ terms, in that the response is based on the variable's content and not on its address. You can confirm that so:

```
kd> ?? Irp

      struct _IRP * 0xff70fbc0
```

since **??** always functions on the basis of C++ (see the **??** command). For the MASM type of evaluation, try **?** (see the **?** command):

```
kd> ? Irp

      Evaluate expression: -141181880 = f795bc48
```

which means, The variable *Irp* is located at 0XF795BC48. You can confirm that the variable at that location truly does contain the value 0xFF70FBC0 by displaying memory via **dd** (see the **dd** command):

```
kd> dd f795bc48 l1

f795bc48  ff70fbc0
```

And the storage pointed to is:

```
kd> dd 0xff70fbc0

ff70fbc0  00940006 00000000 00000070 ff660c30

ff70fbd0  ff70fbd0 ff70fbd0 00000000 00000000

ff70fbe0  01010001 04000000 0006fdc0 00000000

ff70fbf0  00000000 00000000 00000000 04008f20

ff70fc00  00000000 00000000 00000000 00000000

ff70fc10  ff73f4d8 00000000 00000000 00000000

ff70fc20  ff70fc30 ffb05b90 00000000 00000000

ff70fc30  0005000e 00000064 0000003c 9c402408
```

This indeed looks like an IRP, as **dt** shows (see the **dt** command), since the Type and Size members have plausible values:

```
kd> dt Irp

Local var @ 0xf795bc48 Type _IRP*

0xff70fbc0

   +0x000 Type             : 6

   +0x002 Size             : 0x94

   +0x004 MdlAddress       : (null)
   +0x008 Flags            : 0x70

   +0x00c AssociatedIrp    : __unnamed

   +0x010 ThreadListEntry  : _LIST_ENTRY [ 0xff70fbd0 - 0xff70fbd0 ]

   +0x018 IoStatus         : _IO_STATUS_BLOCK

   +0x020 RequestorMode    : 1 ''

   +0x021 PendingReturned  : 0 ''

   +0x022 StackCount       : 1 ''

   +0x023 CurrentLocation  : 1 ''

   +0x024 Cancel           : 0 ''

   +0x025 CancelIrql       : 0 ''

   +0x026 ApcEnvironment   : 0 ''

   +0x027 AllocationFlags  : 0x4 ''

   +0x028 UserIosb         : 0x0006fdc0

   +0x02c UserEvent        :  (null)

   +0x030 Overlay          : __unnamed

   +0x038 CancelRoutine    :  (null)

   +0x03c UserBuffer       : 0x04008f20

   +0x040 Tail             : __unnamed
```

Sometimes you will want to employ C++ evaluation inside a MASM expression. The "@@" prefix achieves that. Since extensions always use parameters as MASM expressions, you can see the effect of @@ when it is employed with the extension**!irp** (see IRPs):

```
kd> !irp @@(Irp)

Irp is active with 1 stacks 1 is current (= 0xff70fc30)

 No Mdl System buffer = ff660c30 Thread ff73f4d8:  Irp stack trace.

     cmd  flg cl Device   File     Completion-Context

>[  e, 0]   5  0 82361348 ffb05b90 00000000-00000000

             \Driver\SIoctl

                        Args: 00000064 0000003c 9c402408 00000000
```

To repeat, without the @@ prefix to the variable *Irp* above, **!irp** would have used the address of the variable rather than the value of the variable. To make the point concrete, if the variable were located at 0xF795BC48 and contained the value 0xFF70FBC0, doing **!irp Irp** instead of **!irp @@(Irp)** would ask WinDbg to format the IRP stack for an IRP supposedly located at 0xF795BC48.

A further thing to understand: The @@ prefix is rather versatile, for its true meaning is, Use the evaluation method other than that currently being used in the surrounding expression. If the overall evaluation is MASM, @@ means C++, and if it is C++, @@ means MASM.

A final bit of advice: If you cannot get an expression to work as you expect, consider whether you're asking the debugger to understand it in MASM or C++ terms.

Displaying and setting memory, variables, registers and so forth

There are quite a few ways to display and change things.

- To display the variables in the current routine (the current "scope"), use **dv** ("Display Variables"). For example, if stopped at Sioctl!SioctlDeviceControl+0x103:

```
kd> dv

   DeviceObject = 0x82361348

            Irp = 0xff70fbc0

   outBufLength = 0x64

         buffer = 0x00000000 ""

          irpSp = 0xff70fc30

           data = 0xf886b0c0 "This String is from Device Driver
!!!"

       ntStatus = 0

            mdl = 0x00000000

    inBufLength = 0x3c

        datalen = 0x26

         outBuf = 0x00000030 ""

          inBuf = 0xff660c30 "This String is from User
Application; using METHOD_BUFFERED"
```

  This is a list of parameter variables and local variables known at the breakpoint. "Known" is an important qualifier. For example, if a variable is optimized into a register, it will not be displayed, although one can dip into disassembly (**View**(**Disassembly** brings up the disassembly window) and examine registers.

If only a single variable is of interest, you can do:

```
kd> dv outBufLength

   outBufLength = 0x64
```

- Another useful command is **dt** ("Display Type"). For example, continuing to use the breakpoint at Sioctl!SioctlDeviceControl+0x103:

```
kd> dt Irp

Local var @ 0xf795bc48 Type _IRP*

0xff70fbc0

   +0x000 Type             : 6

   +0x002 Size             : 0x94

   +0x004 MdlAddress       : (null)

   +0x008 Flags            : 0x70

   +0x00c AssociatedIrp    : __unnamed

   +0x010 ThreadListEntry  : _LIST_ENTRY [ 0xff70fbd0 -
0xff70fbd0 ]

   +0x018 IoStatus         : _IO_STATUS_BLOCK

   +0x020 RequestorMode    : 1 ''

   +0x021 PendingReturned  : 0 ''

   +0x022 StackCount       : 1 ''

   +0x023 CurrentLocation  : 1 ''

   +0x024 Cancel           : 0 ''

   +0x025 CancelIrql       : 0 ''

   +0x026 ApcEnvironment   : 0 ''

   +0x027 AllocationFlags  : 0x4 ''

   +0x028 UserIosb         : 0x0006fdc0

   +0x02c UserEvent        : (null)

   +0x030 Overlay          : __unnamed
   +0x038 CancelRoutine    : (null)

   +0x03c UserBuffer       : 0x04008f20

   +0x040 Tail             : __unnamed
```

The above says that the variable *Irp* is at 0xF795BC48 and that it contains 0xFF70FBC0; since **dt** knows the variable to be an IRP pointer (“Type _IRP*”), the area at 0xFF70FBC0 is formatted as an IRP.

To expand the structure one level:

```
kd> dt -r1 Irp

Local var @ 0xf795bc48 Type _IRP*

0xff70fbc0

   +0x000 Type             : 6

   +0x002 Size             : 0x94

   +0x004 MdlAddress       : (null)

   +0x008 Flags            : 0x70

   +0x00c AssociatedIrp    : __unnamed

      +0x000 MasterIrp        : 0xff660c30

      +0x000 IrpCount         : -10089424

      +0x000 SystemBuffer     : 0xff660c30

   +0x010 ThreadListEntry  : _LIST_ENTRY [ 0xff70fbd0 -
0xff70fbd0 ]

      +0x000 Flink            : 0xff70fbd0  [ 0xff70fbd0 -
0xff70fbd0 ]

      +0x004 Blink            : 0xff70fbd0  [ 0xff70fbd0 -
0xff70fbd0 ]

   +0x018 IoStatus         : _IO_STATUS_BLOCK

      +0x000 Status           : 0

      +0x000 Pointer          : (null)

      +0x004 Information      : 0

   +0x020 RequestorMode    : 1 ''

   +0x021 PendingReturned  : 0 ''
   +0x022 StackCount       : 1 ''
   +0x023 CurrentLocation  : 1 ''
   +0x024 Cancel           : 0 ''
   +0x025 CancelIrql       : 0 ''
   +0x026 ApcEnvironment   : 0 ''
   +0x027 AllocationFlags  : 0x4 ''
   +0x028 UserIosb         : 0x0006fdc0
      +0x000 Status           : 67142040
      +0x000 Pointer          : 0x04008198
      +0x004 Information      : 0x2a
   +0x02c UserEvent        : (null)
   +0x030 Overlay          : __unnamed
      +0x000 AsynchronousParameters : __unnamed
      +0x000 AllocationSize   : _LARGE_INTEGER 0x0
   +0x038 CancelRoutine    : (null)
   +0x03c UserBuffer       : 0x04008f20
   +0x040 Tail             : __unnamed
      +0x000 Overlay          : __unnamed
      +0x000 Apc              : _KAPC
                  +0x000 CompletionKey    : (null)
```

It is possible to display some structures even when they aren't in scope (assuming, that is, that the memory in question has not been reused for some other purpose):

```
kd> dt nt!_IRP 0xff70fbc0
   +0x000 Type             : 6
   +0x002 Size             : 0x94
   +0x004 MdlAddress       : (null)
   +0x008 Flags            : 0x70
   +0x00c AssociatedIrp    : __unnamed
   +0x010 ThreadListEntry  : _LIST_ENTRY [ 0xff70fbd0 -
0xff70fbd0 ]

   +0x018 IoStatus         : _IO_STATUS_BLOCK

   +0x020 RequestorMode    : 1 ''

   +0x021 PendingReturned  : 0 ''

   +0x022 StackCount       : 1 ''

   +0x023 CurrentLocation  : 1 ''

   +0x024 Cancel           : 0 ''

   +0x025 CancelIrql       : 0 ''

   +0x026 ApcEnvironment   : 0 ''

   +0x027 AllocationFlags  : 0x4 ''

   +0x028 UserIosb         : 0x0006fdc0

   +0x02c UserEvent        : (null)

   +0x030 Overlay          : __unnamed

   +0x038 CancelRoutine    : (null)

   +0x03c UserBuffer       : 0x04008f20

   +0x040 Tail             : __unnamed
```

The command above exploits your knowledge that there is an IRP at 0xFF70FBC0 and the fact that there is a mapping of the IRP structure in *ntoskrnl*.

- What if you're interested in a single field of a structure with many member fields? Take the member Size, for example:

```
kd> dt nt!_IRP Size 0xff70fbc0

unsigned short 0x94
```

A rather more intuitive way is the **??** ("Evaluate C++ Expression") command:

```
kd> ?? Irp->Size

unsigned short 0x94
```

That is, **??** understands that its parameter is a pointer to a member of the appropriate structure.

- To display memory without the formatting above, commands like **dd**, **dw** and **db** ("Display Memory") are available:

```
kd> dd 0xff70fbc0 l0x10

ff70fbc0  00940006 00000000 00000070 ff660c30

ff70fbd0  ff70fbd0 ff70fbd0 00000000 00000000

ff70fbe0  01010001 04000000 0006fdc0 00000000

ff70fbf0  00000000 00000000 00000000 04008f20


kd> dw 0xff70fbc0 l0x20

ff70fbc0  0006 0094 0000 0000 0070 0000 0c30 ff66

ff70fbd0  fbd0 ff70 fbd0 ff70 0000 0000 0000 0000

ff70fbe0  0001 0101 0000 0400 fdc0 0006 0000 0000

ff70fbf0  0000 0000 0000 0000 0000 0000 8f20 0400


kd> db 0xff70fbc0 l0x40

ff70fbc0  06 00 94 00 00 00 00 00-70 00 00 00 30 0c 66 ff  ........p...0.f.

ff70fbd0  d0 fb 70 ff d0 fb 70 ff-00 00 00 00 00 00 00 00  ..p...p.........

ff70fbe0  01 00 01 01 00 00 00 04-c0 fd 06 00 00 00 00 00  ................

ff70fbf0  00 00 00 00 00 00 00 00-00 00 00 00 20 8f 00 04  ............ ...
```

  (Note: The second parameter in each of the three commands above is a length, given by *l* (the letter "l") immediately followed by a value such as 0x10.)

  The first displays 16 doublewords (four bytes each, or 64 bytes total) as double words. The second displays the same as words. The third, the same as bytes.

- What about changing a variable? Continuing at Sioctl!SioctlDeviceControl+0x103, you would see that the intuitive form

```
kd> outBufLength = 00

     ^ Syntax error in 'outBufLength = 00'
```

  doesn't work. But **??** does the job:

```
kd> ?? outBufLength = 0
```

```
unsigned long 0
```

Now go back to the IRP you used with **dt** above:

```
kd> dt Irp

Local var @ 0xf795bc48 Type _IRP*

0xff70fbc0

   +0x000 Type             : 6

   +0x002 Size             : 0x94

   +0x004 MdlAddress       : (null)

   +0x008 Flags            : 0x70

   +0x00c AssociatedIrp    : __unnamed

   +0x010 ThreadListEntry  : _LIST_ENTRY [ 0xff70fbd0 -
0xff70fbd0 ]

   +0x018 IoStatus         : _IO_STATUS_BLOCK

   +0x020 RequestorMode    : 1 ''

   +0x021 PendingReturned  : 0 ''

   +0x022 StackCount       : 1 ''

   +0x023 CurrentLocation  : 1 ''

   +0x024 Cancel           : 0 ''

   +0x025 CancelIrql       : 0 ''

   +0x026 ApcEnvironment   : 0 ''

   +0x027 AllocationFlags  : 0x4 ''

   +0x028 UserIosb         : 0x0006fdc0

   +0x02c UserEvent        : (null)

   +0x030 Overlay          : __unnamed

   +0x038 CancelRoutine    : (null)

   +0x03c UserBuffer       : 0x04008f20

   +0x040 Tail             : __unnamed
```

To change the first word (two bytes, that is) via **ew** (“Enter Values”):

```
kd> ew 0xff70fbc0 3


kd> dt Irp

Local var @ 0xf795bc48 Type _IRP*

0xff70fbc0

   +0x000 Type             : 3

   +0x002 Size             : 0x94

   +0x004 MdlAddress       : (null)

   +0x008 Flags            : 0x70

   +0x00c AssociatedIrp    : __unnamed

   +0x010 ThreadListEntry  : _LIST_ENTRY [ 0xff70fbd0 -
0xff70fbd0 ]

   +0x018 IoStatus         : _IO_STATUS_BLOCK

   +0x020 RequestorMode    : 1 ''

   +0x021 PendingReturned  : 0 ''

   +0x022 StackCount       : 1 ''

   +0x023 CurrentLocation  : 1 ''

   +0x024 Cancel           : 0 ''

   +0x025 CancelIrql       : 0 ''

   +0x026 ApcEnvironment   : 0 ''

   +0x027 AllocationFlags  : 0x4 ''

   +0x028 UserIosb         : 0x0006fdc0

   +0x02c UserEvent        : (null)

   +0x030 Overlay          : __unnamed

   +0x038 CancelRoutine    : (null)

   +0x03c UserBuffer       : 0x04008f20

   +0x040 Tail             : __unnamed
```

Of course, the following is probably more natural than **ew**:

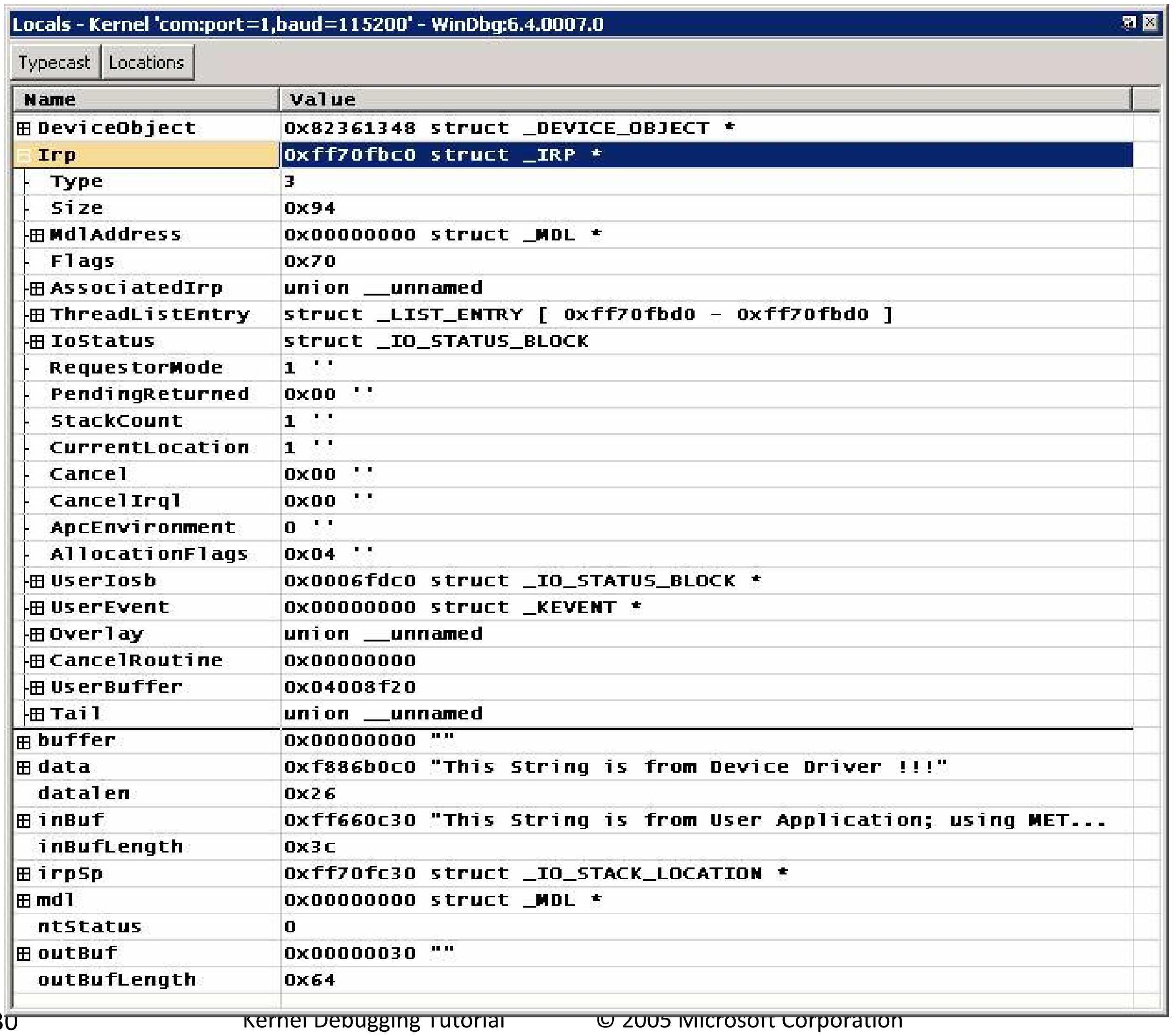
```
kd> ?? irp->type = 3

Type does not have given member error at 'type = 3'

kd> ?? irp->Type = 3

short 3


kd> dt irp

ioctlapp!Irp

Local var @ 0xf795bc48 Type _IRP*

0xff70fbc0

   +0x000 Type             : 3

   +0x002 Size             : 0x94

   +0x004 MdlAddress       : (null)

   +0x008 Flags            : 0x70

   +0x00c AssociatedIrp    : __unnamed

   +0x010 ThreadListEntry  : _LIST_ENTRY [ 0xff70fbd0 -
0xff70fbd0 ]

   +0x018 IoStatus         : _IO_STATUS_BLOCK

   +0x020 RequestorMode    : 1 ''

   +0x021 PendingReturned  : 0 ''

   +0x022 StackCount       : 1 ''

   +0x023 CurrentLocation  : 1 ''

   +0x024 Cancel           : 0 ''

   +0x025 CancelIrql       : 0 ''

   +0x026 ApcEnvironment   : 0 ''

   +0x027 AllocationFlags  : 0x4 ''

   +0x028 UserIosb         : 0x0006fdc0

   +0x02c UserEvent        : (null)

   +0x030 Overlay          : __unnamed
```

```
+0x038 CancelRoutine    : (null)

+0x03c UserBuffer       : 0x04008f20

+0x040 Tail             : __unnamed
```

There are a couple of things to notice in the above. First, the case of a member of a structure is significant, as shown by WinDbg's claim that there was no "type" member of *Irp*. Second, **dt irp** was ambiguous, but WinDbg helpfully displayed the instances it thought were likely fits, one in ioctlapp.exe and one in sioctl.sys. Since case can be significant, you should employ it whenever you know it.

In addition to **ew**, there are other "Enter Values" commands: **eb** for a byte, **ed** for a doubleword, **eq** for quadword (8 bytes) and so forth. Refer to the Debugging Tools for Windows help file under "Enter Values."

- The Locals window may be easier to use for things like a structure with imbedded pointers to structures:



Locals - Kernel 'com:port=1,baud=115200' - WinDbg:6.4.0007.0

Name	Value
⊞ DeviceObject	0x82361348 struct _DEVICE_OBJECT *
⊟ Irp	0xff70fbc0 struct _IRP *
Type	3
Size	0x94
⊞ MdlAddress	0x00000000 struct _MDL *
Flags	0x70
⊞ AssociatedIrp	union __unnamed
⊞ ThreadListEntry	struct _LIST_ENTRY [0xff70fbd0 - 0xff70fbd0]
⊞ IoStatus	struct _IO_STATUS_BLOCK
RequestorMode	1 ''
PendingReturned	0x00 ''
StackCount	1 ''
CurrentLocation	1 ''
Cancel	0x00 ''
CancelIrql	0x00 ''
ApcEnvironment	0 ''
AllocationFlags	0x04 ''
⊞ UserIosb	0x0006fdc0 struct _IO_STATUS_BLOCK *
⊞ UserEvent	0x00000000 struct _KEVENT *
⊞ Overlay	union __unnamed
⊞ CancelRoutine	0x00000000
⊞ UserBuffer	0x04008f20
⊞ Tail	union __unnamed
⊞ buffer	0x00000000 ""
⊞ data	0xf886b0c0 "This String is from Device Driver !!!"
datalen	0x26
⊞ inBuf	0xff660c30 "This String is from User Application; using MET...
inBufLength	0x3c
⊞ irpSp	0xff70fc30 struct _IO_STACK_LOCATION *
⊞ mdl	0x00000000 struct _MDL *
ntStatus	0
⊞ outBuf	0x00000030 ""
outBufLength	0x64

You can, not so incidentally, overtype values in the Locals window to change values.

- Registers (as well as segment pointers and flags) can be displayed and altered. For example:

```
kd> r

eax=81478f68 ebx=00000000 ecx=814243a8 edx=0000003c esi=81778ea0
edi=81478f68

eip=f8803553 esp=f7813bb4 ebp=f7813c3c iopl=0         nv up ei ng
nz ac pe nc

cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000
efl=00000292
```

Or just:

```
kd> r eax

eax=81478f68
```

Sometimes you will want to alter a register. Eax, for instance, is often used at exit from a routine to carry the return value, so, just before exit from a routine:

```
r eax = 0xc0000001
```

which happens to denote the status value STATUS_UNSUCCESSFUL.

Here are some other examples:

```
r eip = poi(@esp)

r esp = @esp + 0xc
```

which mean, respectively, Set Eip (the instruction pointer) to the value found at offset 0x0 from the stack pointer, and Add 0xC to Esp (the stack pointer), effectively unwinding the stack pointer. The Debugging Tools for Windows help file, under "Register Syntax," explains **poi** and why register names have to be prefixed with a single "@" in some places.

You may be asking yourself how the above register-setting commands could be useful. Consider the case where there is a "bad" driver whose DriverEntry will cause a bug check ("blue screen") — due to an access violation, perhaps. You could deal with the problem by setting a deferred breakpoint when *ntoskrnl* loads. The following command must be typed on a single line:

```
bu sioctl!DriverEntry "r eip = poi(@esp); r eax = 0xc0000001; r
esp = @esp + 0xc; .echo sioctl!DriverEntry entered; g"
```

The meaning is: At sioctl.sys's DriverEntry, 1) set the instruction pointer (Eip) thus; 2) set the return code (Eax) thus; 3) set the stack pointer (Esp) thus; 4) announce that DriverEntry has been entered; and 5) proceed. (Of course, this technique merely removes the possibility of DriverEntry causing a blowup such as an access violation. If the operating system expects the driver to be supplying function, that function is not going to be available, and down the road there may be other problems.)

In case you're wondering whether it is possible to use a register to set a variable, it is. For example, returning yet again to the IoCtl dispatch routine:

```
kd> r

eax=00000000 ebx=00000000 ecx=81a88f18 edx=81a88ef4 esi=ff9e18a8
edi=ff981e7e

eip=f87a40fe esp=f88fac78 ebp=f88fac90 iopl=0         nv up ei pl
zr na po nc

cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000
efl=00000246



kd> ?? ntStatus = @ecx

long -2119659752

kd> dd &ntStatus l1

f88fac78  81a88f18
```

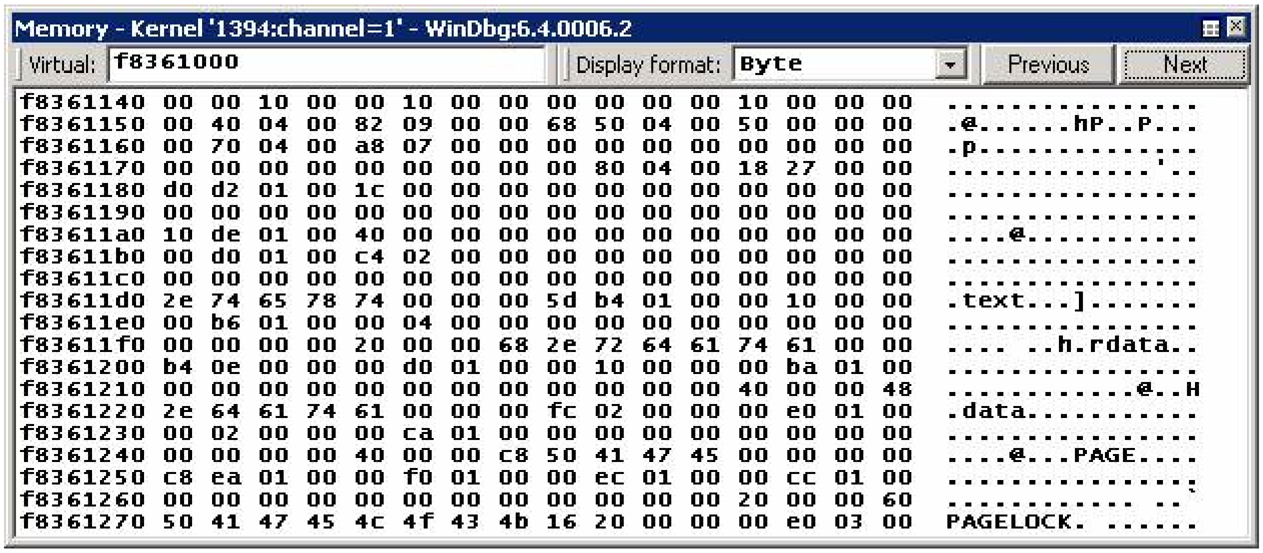
In this case, the form *@ecx* should be employed, to ensure WinDbg knows you're referring to a register.

There are more registers than those shown by default. To see the full complement, use the **rM** command ("M" must be uppercase; this is actually the **r** command with the parameter *M*, with no space allowed between command and parameter):

```
kd> rM 0xff

eax=00000001 ebx=0050e2a3 ecx=80571780 edx=000003f8 esi=000000c0
edi=d87a75a8

eip=804df1c0 esp=8056f564 ebp=8056f574 iopl=0         nv up ei pl
nz na pe nc
```

```
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000
efl=00000202

fpcw=0000: rn 24 ------  fpsw=0000: top=0 cc=0000 --------
fptw=0000

fopcode=6745  fpip=2301:a0020000  fpdp=dcfe:efcdab89

st0= 5.143591243081972142170e-4932  st1=
0.001025530551233493990e-4933

st2= 0.000000002357022271740e-4932  st3=
2.471625214254630491460e-4906

st4= 3.370207406893238285120e-4932  st5=-
7.461339669368745455450e+4855

st6= 6.698191557136036873700e-4932  st7=-
2.455410815115332972380e-4906

mm0=c3d2e1f010325476  mm1=0000ffdff1200000

mm2=000000018168d902  mm3=f33cffdff1200000

mm4=804efc868056f170  mm5=7430804efb880000

mm6=ff02740200000000  mm7=f1a48056f1020000

xmm0=0 9.11671e-041 3.10647e+035 -1.154e-034

xmm1=-7.98492e-039 -2.83455e+038 -2.91106e+038 5.85182e-042

xmm2=1.77965e-043 -1.17906e-010 -4.44585e-038 -7.98511e-039

xmm3=-7.98511e-039 0 0 -7.98504e-039

xmm4=-7.98503e-039 1.20545e-040 -1.47202e-037 -1.47202e-037

xmm5=-2.05476e+018 -452.247 -1.42468e-037 -8.60834e+033

xmm6=2.8026e-044 -1.47202e-037 -452.247 0

xmm7=8.40779e-045 -7.98503e-039 0 -7.98511e-039

cr0=8001003b cr2=d93db000 cr3=00039000

dr0=00000000 dr1=00000000 dr2=00000000

dr3=00000000 dr6=ffff0ff0 dr7=00000400 cr4=000006d9
```

- If you don't like to use commands to change things, you can bring up a memory window (**View→Memory**), variable window (**View→Locals**) or register window (**View→Registers**) and overtype values as you like. For example,



```
f8361140 00 00 10 00 00 10 00 00 00 00 00 00 10 00 00 00  ................
f8361150 00 40 04 00 82 09 00 00 68 50 04 00 50 00 00 00  .@......hP..P...
f8361160 00 70 04 00 a8 07 00 00 00 00 00 00 00 00 00 00  .p..............
f8361170 00 00 00 00 00 00 00 00 00 80 04 00 18 27 00 00  .............'..
f8361180 d0 d2 01 00 1c 00 00 00 00 00 00 00 00 00 00 00  ................
f8361190 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
f83611a0 10 de 01 00 40 00 00 00 00 00 00 00 00 00 00 00  ....@...........
f83611b0 00 d0 01 00 c4 02 00 00 00 00 00 00 00 00 00 00  ................
f83611c0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
f83611d0 2e 74 65 78 74 00 00 00 5d b4 01 00 00 10 00 00  .text...].......
f83611e0 00 b6 01 00 00 04 00 00 00 00 00 00 00 00 00 00  ................
f83611f0 00 00 00 00 20 00 00 68 2e 72 64 61 74 61 00 00  .... ..h.rdata..
f8361200 b4 0e 00 00 00 d0 01 00 00 10 00 00 00 ba 01 00  ................
f8361210 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 48  ............@..H
f8361220 2e 64 61 74 61 00 00 00 fc 02 00 00 00 e0 01 00  .data...........
f8361230 00 02 00 00 00 ca 01 00 00 00 00 00 00 00 00 00  ................
f8361240 00 00 00 00 40 00 00 c8 50 41 47 45 00 00 00 00  ....@...PAGE....
f8361250 c8 ea 01 00 00 f0 01 00 00 ec 01 00 00 cc 01 00  ................
f8361260 00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 60  ............ ..`
f8361270 50 41 47 45 4c 4f 43 4b 16 20 00 00 00 e0 03 00  PAGELOCK. ......
```

In the above, you could overtype the hexadecimal values.

Controlling execution

Earlier (see IoCreateDevice) you were asked to let execution proceed from one point to the next, without being told how. There are several ways to control execution. All the below, except the first, assume that execution is currently halted.

- Break in (CTRL-BREAK) — This sequence will always interrupt a system, so long as that system is running and is in communication with WinDbg (the sequence in KD is CTRL-C).

- Step over (F10) — This causes execution to proceed one statement (if C or C++ and WinDbg is in "source mode," toggled by **Debug→Source Mode**) or one instruction at a time, with the provision that if a call is encountered, execution passes over the call without entering the called routine.

- Step in (F11) — This is like step-over, except that execution of a call does go into the called routine.

- Step out (SHIFT-F11) — This causes execution to run to an exit from the current routine (current place in the call stack). Useful if you've seen enough of the routine.

- Run to cursor (F7 or CRTL-F10) — Place the cursor in a source or disassembly window where you want execution to break, then hit F7; execution will run to that point. A word of caution, however: If the flow of execution were not to reach that point (e.g., an IF statement isn't executed), WinDbg would *not* break, because execution did not come to the indicated point!

- Run (F5) — Run until a breakpoint is encountered or an event like a bug check occurs. You may think of Run as normal execution mode.

- Set instruction to current line (CTRL-SHIFT-I) — In a source window, you can put the cursor on a line, do that keystroke sequence, and execution will start from that point as soon as you let it proceed (e.g., F5 or F10). This is handy if you want to retry a sequence. But it requires some care. For example, registers and variables are not set to what they would be if execution had reached that line naturally.

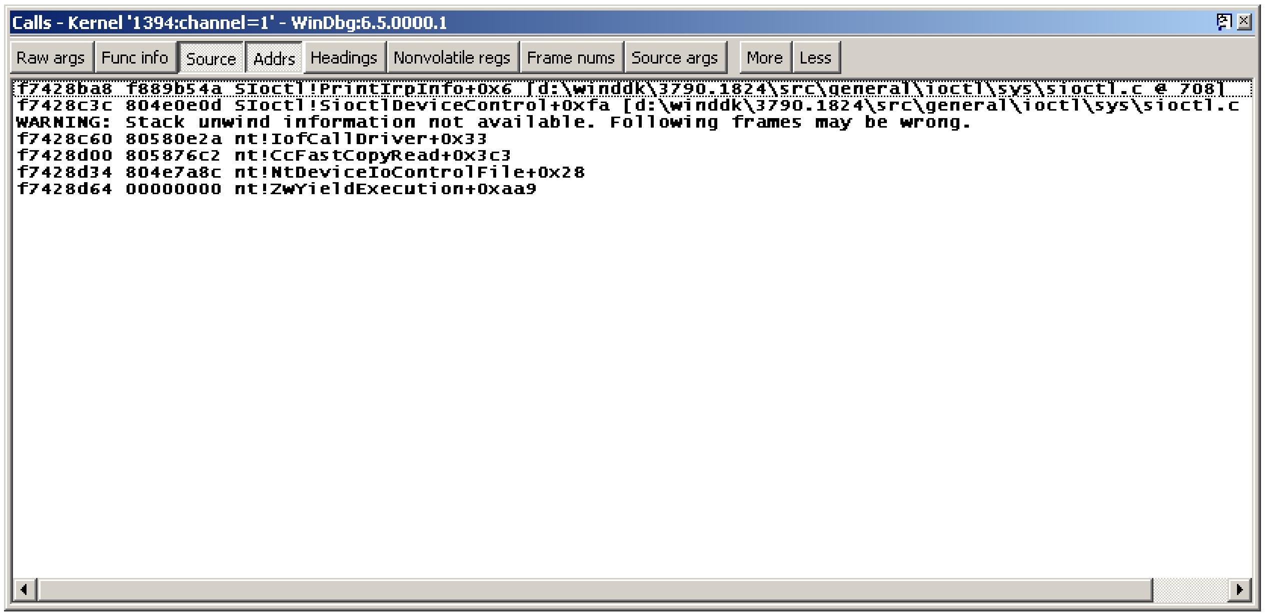
- Direct setting of Eip — You can put a value into the Eip register, and as soon as you hit F5 (or F10 or whatever), execution commences from that address. As should be obvious, this is like setting instruction to the cursor-designated current line, except that you specify the address of an Assembly instruction.

The call stack

At almost any point in execution, there is an area of storage that is used as the stack; the stack is where local state, including parameters and return addresses, is saved. There is a kernel stack, if execution is in kernel space, and a user stack for execution in user space. When you hit a breakpoint, it is likely there will be several routines on the current stack, and there can be quite a few. For example, if instruction has stopped because of a breakpoint in the routine PrintIrpInfo in sioctl.sys, use **k** ("Stack Backtrace"):

```
kd> k

ChildEBP RetAddr

f7428ba8 f889b54a SIoctl!PrintIrpInfo+0x6
[d:\winddk\3790.1824\src\general\ioctl\sys\sioctl.c @ 708]

f7428c3c 804e0e0d SIoctl!SioctlDeviceControl+0xfa
[d:\winddk\3790.1824\src\general\ioctl\sys\sioctl.c @ 337]

WARNING: Stack unwind information not available. Following frames may
be wrong.

f7428c60 80580e2a nt!IofCallDriver+0x33

f7428d00 805876c2 nt!CcFastCopyRead+0x3c3

f7428d34 804e7a8c nt!NtDeviceIoControlFile+0x28

f7428d64 00000000 nt!ZwYieldExecution+0xaa9
```

The topmost line (newest) is the stack frame where control stopped. You can see earlier callers, too, but if you don't have symbols, they may not be represented correctly. Since you enjoy access to the symbols for sioctl.sys, you are presented with file and line number information for each frame involving the driver.



```
f7428ba8 f889b54a SIoctl!PrintIrpInfo+0x6 [d:\winddk\3790.1824\src\general\ioctl\sys\sioctl.c @ 708]
f7428c3c 804e0e0d SIoctl!SioctlDeviceControl+0xfa [d:\winddk\3790.1824\src\general\ioctl\sys\sioctl.c
WARNING: Stack unwind information not available. Following frames may be wrong.
f7428c60 80580e2a nt!IofCallDriver+0x33
f7428d00 805876c2 nt!CcFastCopyRead+0x3c3
f7428d34 804e7a8c nt!NtDeviceIoControlFile+0x28
f7428d64 00000000 nt!ZwYieldExecution+0xaa9
```

You can readily switch to the source window for IoCtl's IRP handler, but suppose you want to see an earlier routine? You bring up the calls window (**View→Call stack**), so:

You can double-click on an entry and be taken to the source file, if that source file can be located.

If you are interested only in variables pertaining to a routine on the stack, you can make the routine current by double-clicking on its line above, or you can do **kn** (a sibling of **k**) and then **.frame**. For example, to get information about the dispatch routine that called PrintIrpInfo:

```
kd> kn

 # ChildEBP RetAddr

00 f7428ba8 f889b54a SIoctl!PrintIrpInfo+0x6
[d:\winddk\3790.1824\src\general\ioctl\sys\sioctl.c @ 708]

01 f7428c3c 804e0e0d SIoctl!SioctlDeviceControl+0xfa
[d:\winddk\3790.1824\src\general\ioctl\sys\sioctl.c @ 337]

WARNING: Stack unwind information not available. Following frames may
be wrong.

02 f7428c60 80580e2a nt!IofCallDriver+0x33

03 f7428d00 805876c2 nt!CcFastCopyRead+0x3c3

04 f7428d34 804e7a8c nt!NtDeviceIoControlFile+0x28

05 f7428d64 00000000 nt!ZwYieldExecution+0xaa9
```

```
kd> .frame 1

01 f7428c3c 804e0e0d SIoctl!SioctlDeviceControl+0xfa
[d:\winddk\3790.1824\src\general\ioctl\sys\sioctl.c @ 337]
```

Having set the frame number, you're able to display (or change, if you wish) variables known in that frame and registers belonging to that frame:

```
kd> dv

   DeviceObject = 0x80f895e8

            Irp = 0x820572a8

   outBufLength = 0x64

         buffer = 0x00000000 ""

          irpSp = 0x82057318

           data = 0xf889b0c0 "This String is from Device Driver !!!"

       ntStatus = 0

            mdl = 0x00000000

    inBufLength = 0x3c

        datalen = 0x26

         outBuf = 0x82096b20 "This String is from User Application;
using METHOD_BUFFERED"

          inBuf = 0x82096b20 "This String is from User Application;
using METHOD_BUFFERED"

kd> r

eax=00000000 ebx=00000000 ecx=80506be8 edx=820572a8 esi=81fabda0
edi=820572a8

eip=f889bcf6 esp=f7428ba4 ebp=f7428ba8 iopl=0         nv up ei ng nz ac
pe nc

cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000
efl=00000292

SIoctl!PrintIrpInfo+0x6:

f889bcf6 8b4508           mov     eax,[ebp+0x8]
ss:0010:f7428bb0=820572a8
```

Finding a name in a module

The **x** ("Examine Symbols") command locates symbols in modules. For example, if you want to put a breakpoint at the Ioctl routine to handle DeviceIoControl IRPs but don't quite remember the routine's name, you might do:

```
kd> x sioctl!*ioctl*

f8883080 SIoctl!SioctlUnloadDriver (struct _DRIVER_OBJECT *)

f8883010 SIoctl!SioctlCreateClose (struct _DEVICE_OBJECT *, struct _IRP
*)

f8883450 SIoctl!SioctlDeviceControl (struct _DEVICE_OBJECT *, struct
_IRP *)
```

This means, Tell me all the symbols in the module sioctl that contain "ioctl."

That may seem trivial. Consider, however, this message seen in the debugger in an actual support case:

```
PopPolicyWorkerAction: action request 2 failed c000009a
```

On a guess that the routine PopPolicyWorkerAction is in *ntoskrnl*, you might look there:

```
kd> x nt!PopPolicy*

805146c0 nt!PopPolicyWorkerThread = <no type information>

8064e389 nt!PopPolicySystemIdle = <no type information>

805b328d nt!PopPolicyWorkerNotify = <no type information>

8056e620 nt!PopPolicyLock = <no type information>

8064d5f8 nt!PopPolicyWorkerActionPromote = <no type information>

805c7d10 nt!PopPolicyWorkerMain = <no type information>

8064d51b nt!PopPolicyWorkerAction = <no type information>

80561c70 nt!PopPolicy = <no type information>

8056e878 nt!PopPolicyIrpQueue = <no type information>

80561a98 nt!PopPolicyLockThread = <no type information>

8064e74a nt!PopPolicyTimeChange = <no type information>

8056e8b0 nt!PopPolicyWorker = <no type information>
```

With that information, you might put a breakpoint in the routine shown in red.

Dealing with optimized code

If an executable was built in a free build and/or with some optimization, it may be difficult to follow execution in a source window, and local variables may not be available or may be shown with incorrect values. With x86 code you might try following execution in the source window and also in the disassembly window (it helps to place those windows side by side). You don't have to understand very much about x86 Assembly to follow the flow of control; look primarily for test (e.g., *test* or *cmp*) and branch (e.g., *jnz*) instructions in order to follow flow.

Selected techniques

That covers fundamental operations. Although the overarching focus here is not on how to investigate particular areas, there are nonetheless a number of substantive debugger commands — technically, they are extensions and are written as DLLs — as well as techniques that deserve mention because they are used over and over in many areas.

Processes and threads

To see the current process (at a stopped point):

```
kd> !process

PROCESS 816fc3c0  SessionId: 1  Cid: 08f8    Peb: 7ffdf000  ParentCid: 0d8c

    DirBase: 10503000  ObjectTable: e1afeaa8  HandleCount:  19.

    Image: ioctlapp.exe

    VadRoot 825145e0 Vads 22 Clone 0 Private 38. Modified 0. Locked 0.

    DeviceMap e10d0198

    Token                             e1c8e030

    ElapsedTime                       00:00:00.518

    UserTime                          00:00:00.000

    KernelTime                        00:00:00.109

    QuotaPoolUsage[PagedPool]         9096

    QuotaPoolUsage[NonPagedPool]      992

    Working Set Sizes (now,min,max)  (263, 50, 345) (1052KB, 200KB, 1380KB)

    PeakWorkingSetSize                263

    VirtualSize                       6 Mb
```

```
        PeakVirtualSize                  6 Mb

        PageFaultCount                   259

        MemoryPriority                   BACKGROUND

        BasePriority                     8

        CommitCharge                     48


                  THREAD 825d2020  Cid 08f8.0708  Teb: 7ffde000
          Win32Thread: 00000000 RUNNING on processor 0
```

The addresses of the process block (EPROCESS) and thread block (ETHREAD) have been marked in red. You could use these in a conditional breakpoint.

To see all the processes in summary form:

```
kd> !process 0 0

**** NT ACTIVE PROCESS DUMP ****

PROCESS 826af478  SessionId: none  Cid: 0004    Peb: 00000000
ParentCid: 0000

    DirBase: 02c20000  ObjectTable: e1001e60  HandleCount: 363.

    Image: System


PROCESS 82407d88  SessionId: none  Cid: 0158    Peb: 7ffdf000
ParentCid: 0004

    DirBase: 1fbe8000  ObjectTable: e13ff740  HandleCount:  24.

    Image: smss.exe


PROCESS 82461d88  SessionId: 0  Cid: 0188    Peb: 7ffdf000  ParentCid:
0158

    DirBase: 1f14d000  ObjectTable: e15e8958  HandleCount: 408.

    Image: csrss.exe


...
```

To see a particular process with a thread summary, give the process block's address and ask for detail via the second parameter (see the Debugging Tools for Windows help file for information about the detail parameter):

```
kd> !process 826af478 3

PROCESS 826af478  SessionId: none  Cid: 0004    Peb: 00000000
ParentCid: 0000

    DirBase: 02c20000  ObjectTable: e1001e60  HandleCount: 362.

    Image: System

    VadRoot 81a43840 Vads 4 Clone 0 Private 3. Modified 18884. Locked
0.

    DeviceMap e1002868

    Token                             e1002ae0

    ElapsedTime                       07:19:11.250

    UserTime                          00:00:00.000

    KernelTime                        00:00:11.328

    QuotaPoolUsage[PagedPool]         0

    QuotaPoolUsage[NonPagedPool]      0

    Working Set Sizes (now,min,max)  (54, 0, 345) (216KB, 0KB, 1380KB)

    PeakWorkingSetSize                497

    VirtualSize                       1 Mb

    PeakVirtualSize                   2 Mb

    PageFaultCount                    4179

    MemoryPriority                    BACKGROUND

    BasePriority                      8

    CommitCharge                      7


        THREAD 826af1f8  Cid 0004.0008  Teb: 00000000 Win32Thread:
00000000 WAIT: (WrFreePage) KernelMode Non-Alertable

            80580040  SynchronizationEvent

            80581140  NotificationTimer
        THREAD 826aea98  Cid 0004.0010  Teb: 00000000 Win32Thread:
00000000 WAIT: (WrQueue) KernelMode Non-Alertable

            80582d80  QueueObject


        THREAD 826ae818  Cid 0004.0014  Teb: 00000000 Win32Thread:
00000000 WAIT: (WrQueue) KernelMode Non-Alertable

            80582d80  QueueObject


...
```

To see a particular thread in maximum detail, use **!thread** with 0xFF as the detail parameter:

```
kd> !thread 826af1f8 0xff

THREAD 826af1f8  Cid 0004.0008  Teb: 00000000 Win32Thread: 00000000
WAIT: (WrFreePage) KernelMode Non-Alertable

    80580040  SynchronizationEvent

    80581140  NotificationTimer

Not impersonating

DeviceMap                 e1002868

Owning Process            826af478       Image:         System

Wait Start TickCount      1688197        Ticks: 153 (0:00:00:02.390)

Context Switch Count      9133

UserTime                  00:00:00.0000

KernelTime                00:00:03.0406

Start Address nt!Phase1Initialization (0x806fb790)

Stack Init f88b3000 Current f88b2780 Base f88b3000 Limit f88b0000 Call
0

Priority 0 BasePriority 0 PriorityDecrement 0

ChildEBP RetAddr

f88b2798 804edb2b nt!KiSwapContext+0x26 (FPO: [EBP 0xf88b27c0] [0,0,4])
```

```
f88b27c0 804f0e7a nt!KiSwapThread+0x280 (FPO: [Non-Fpo]) (CONV:
fastcall)

f88b27f4 80502fc2 nt!KeWaitForMultipleObjects+0x324 (FPO: [Non-Fpo])
(CONV: stdcall)
```

Driver and device objects

If you are writing a driver, you will often be looking at device stacks. You might begin by finding devices belonging to a certain driver and then examine the stack a device is in. Suppose you are interested in the ScsiPort miniport driver aic78xx.sys. Start with **!drvobj**:

```
kd> !drvobj aic78xx

Driver object (82627250) is for:

 \Driver\aic78xx

Driver Extension List: (id , addr)

(f8386480 8267da38)

Device Object list:

82666030  8267b030  8263c030  8267ca40
```

There are four device objects here. Use To look at the first, use **!devobj** to get some information about the device and **!devstack** to show the device-object stack to which the device object belongs:

```
kd> !devobj 82666030

Device object (82666030) is for:

 aic78xx1Port2Path0Target1Lun0 \Driver\aic78xx DriverObject 82627250

Current Irp 00000000 RefCount 0 Type 00000007 Flags 00001050

Dacl e13bb39c DevExt 826660e8 DevObjExt 82666d10 Dope 8267a9d8 DevNode
8263cdc8

ExtensionFlags (0000000000)

AttachedDevice (Upper) 826bb030 \Driver\Disk

Device queue is not busy.

kd> !devstack 82666030

  !DevObj   !DrvObj            !DevExt   ObjectName

  826bbe00  \Driver\PartMgr    826bbeb8
  826bb030  \Driver\Disk       826bb0e8  DR2

> 82666030  \Driver\aic78xx    826660e8  aic78xx1Port2Path0Target1Lun0

!DevNode 8263cdc8 :

  DeviceInst is
"SCSI\Disk&Ven_QUANTUM&Prod_VIKING_II_4.5WLS&Rev_5520\5&375eb691&1&010"

  ServiceName is "disk"
```

IRPs

The most common method of communication amongst drivers is the I/O Request Packet, or IRP. To see an IRP's I/O completion stack, as for example at Sioctl!SioctlDeviceControl+0x103:

```
kd> !irp @@(Irp)

Irp is active with 1 stacks 1 is current (= 0xff70fc30)

 No Mdl System buffer = ff660c30 Thread ff73f4d8:  Irp stack trace.

     cmd  flg cl Device   File     Completion-Context

>[  e, 0]   5  0 82361348 ffb05b90 00000000-00000000

             \Driver\SIoctl

            Args: 00000064 0000003c 9c402408 00000000
```

To get the full IRP plus its stack, ask for detail:

```
kd> !irp @@(Irp) 1

Irp is active with 1 stacks 1 is current (= 0xff70fc30)

 No Mdl System buffer = ff660c30 Thread ff73f4d8:  Irp stack trace.

Flags = 00000070

ThreadListEntry.Flink = ff70fbd0

ThreadListEntry.Blink = ff70fbd0

IoStatus.Status = 00000000

IoStatus.Information = 00000000

RequestorMode = 00000001

Cancel = 00

CancelIrql = 0
```

```
ApcEnvironment = 00

UserIosb = 0006fdc0

UserEvent = 00000000

Overlay.AsynchronousParameters.UserApcRoutine = 00000000

Overlay.AsynchronousParameters.UserApcContext = 00000000

Overlay.AllocationSize = 00000000 - 00000000

CancelRoutine = 00000000

UserBuffer = 04008f20

&Tail.Overlay.DeviceQueueEntry = ff70fc00

Tail.Overlay.Thread = ff73f4d8

Tail.Overlay.AuxiliaryBuffer = 00000000

Tail.Overlay.ListEntry.Flink = 00000000

Tail.Overlay.ListEntry.Blink = 00000000

Tail.Overlay.CurrentStackLocation = ff70fc30

Tail.Overlay.OriginalFileObject = ffb05b90

Tail.Apc = 00000000

Tail.CompletionKey = 00000000

     cmd  flg cl Device   File     Completion-Context

>[  e, 0]   5  0 82361348 ffb05b90 00000000-00000000

             \Driver\SIoctl

                  Args: 00000064 0000003c 9c402408 00000000
```

To get only the IRP proper, with its first-level members:

```
kd> dt nt!_IRP @@(Irp)

   +0x000 Type             : 6

   +0x002 Size             : 0x94

   +0x004 MdlAddress       : (null)

   +0x008 Flags            : 0x70

   +0x00c AssociatedIrp    : __unnamed
   +0x010 ThreadListEntry  : _LIST_ENTRY [ 0xff70fbd0 - 0xff70fbd0 ]

   +0x018 IoStatus         : _IO_STATUS_BLOCK

   +0x020 RequestorMode    : 1 ''

   +0x021 PendingReturned  : 0 ''

   +0x022 StackCount       : 1 ''

   +0x023 CurrentLocation  : 1 ''

   +0x024 Cancel           : 0 ''

   +0x025 CancelIrql       : 0 ''

   +0x026 ApcEnvironment   : 0 ''

   +0x027 AllocationFlags  : 0x4 ''

   +0x028 UserIosb         : 0x0006fdc0

   +0x02c UserEvent        : (null)

   +0x030 Overlay          : __unnamed

   +0x038 CancelRoutine    : (null)

   +0x03c UserBuffer       : 0x04008f20

   +0x040 Tail             : __unnamed
```

IRQL

A command (available for targets that are Windows Server 2003 or later) for occasional use is **!irql**, because it shows the current IRQL on the concerned processor. Breaking at Sioctl!SioctlDeviceControl+0x0:

```
kd> !irql

Debugger saved IRQL for processor 0x0 -- 0 (LOW_LEVEL)
```

For an example of a higher IRQL, suppose you added the following code (in curly braces) to Sioctl!SioctlDeviceControl just before the break statement at the end of the IOCTL_SIOCTL_METHOD_BUFFERED clause:

```
Irp->IoStatus.Information =
(outBufLength<datalen?outBufLength:datalen);

{ /* Begin added code */

 KIRQL saveIrql;
```

```
         ULONG i = 0;


         KeRaiseIrql(DISPATCH_LEVEL, &saveIrql);

         i++;

         KeLowerIrql(saveIrql);

        } /* End added code */

        break;
```

Now, if you set a breakpoint at the statement after **KeRaiseIrql** and hit that breakpoint:

```
kd> !irql

Debugger saved IRQL for processor 0x0 -- 2 (DISPATCH_LEVEL)
```

Note, by the way, that the **!pcr** command will not ordinarily show the IRQL you're interested in, namely, the IRQL in force at the point a breakpoint caused a halt.

Dump Files

There is little to say about dump files that is peculiar to them. A few things are worth mentioning.

- There are three kinds of kernel dump files. A full-memory dump is best, but the somewhat less voluminous kernel dump suffices for most purposes. There is also the small-memory dump, which is 64KB in size (and so will be generated more quickly than the two other types). Since a small-memory dump does not have full information about executables, it may be necessary to employ the **.exepath** command to point to executable images if investigating them is required. Windows can be configured to create one of these dump files if a crash (bug check) occurs.
- To investigate a dump file, start WinDbg but do not specify a protocol for a target system. When in WinDbg, open the dump file with **File→Open Crash Dump**. It will help if the symbol path and possibly the source path are already set.
- Now, in the WinDbg command window, do **!analyze –v**, to get a summary. The command will probably suggest an execution context (**.cxr**); setting that context will give you access to the call stack at the time the bug check was issued (which, hopefully, will be close to the actual error). You may need to proceed to processes and threads (**!process** and **!thread**), to look at the list of kernel modules (**lmnt**), from that list to look at selected driver objects (**!drvobj**) and possibly to look at device nodes (**!devnode**), device objects (**!devobj**) and device stacks (**!devstack**). But beyond **!analyze –v**, there are no simple prescriptions in working with a dump file.

If a kernel-mode dump file has been created after a bug check has occurred, debugging this file is similar to debugging a bug check that occurs when a debugger is attached. The following section shows an example of live debugging, but it is similar to the analysis of a dump file.

Debugging a Bug Check

Here is how to begin analyzing a bug check. In this example, the kernel debugger is attached when the crash occurs, but the procedure when analyzing a kernel-mode dump file is similar.

In this example, the sample driver *Sioctl.sys* is loaded, and a breakpoint is set at Sioctl!DriverEntry. When the debugger stops at this breakpoint, set Eip to 0. This is never a valid value, since the instruction pointer cannot be zero. Then let execution proceed via F5. A kernel error will occur, and a bug check will be issued. You can then use the **!analyze** extension command to investigate:

```
kd> !analyze -v

*************************************************************************
********

*
*

*                         Bugcheck Analysis
*

*
*

*************************************************************************
********



SYSTEM_THREAD_EXCEPTION_NOT_HANDLED (7e)

This is a very common bugcheck.  Usually the exception address
pinpoints

the driver/function that caused the problem.  Always note this address

as well as the link date of the driver/image that contains this
address.

Arguments:

Arg1: c0000005, The exception code that was not handled

Arg2: 00000000, The address that the exception occurred at
```

```
Arg3: f88f2bd8, Exception Record Address

Arg4: f88f2828, Context Record Address


Debugging Details:

------------------


EXCEPTION_CODE: (NTSTATUS) 0xc0000005 - The instruction at "0x%08lx"
referenced memory at "0x%08lx". The memory could not be "%s".


FAULTING_IP:

+0

00000000 ??               ???


EXCEPTION_RECORD:  f88f2bd8 -- (.exr ffffffff88f2bd8)

ExceptionAddress: 00000000

   ExceptionCode: c0000005 (Access violation)

  ExceptionFlags: 00000000

NumberParameters: 2

   Parameter[0]: 00000000

   Parameter[1]: 00000000

Attempt to read from address 00000000


CONTEXT:  f88f2828 -- (.cxr ffffffff88f2828)

eax=ffff99ea ebx=00000000 ecx=0000bb40 edx=8055f7a4 esi=e190049e
edi=81e826e8

eip=00000000 esp=f88f2ca0 ebp=f88f2cf0 iopl=0         nv up ei pl nz na
pe nc

cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000
efl=00010202

00000000 ??               ???
```

```
Resetting default scope

DEFAULT_BUCKET_ID:  DRIVER_FAULT

CURRENT_IRQL:  0

ERROR_CODE: (NTSTATUS) 0xc0000005 - The instruction at "0x%08lx"
referenced memory at "0x%08lx". The memory could not be "%s".

READ_ADDRESS:  00000000

BUGCHECK_STR:  0x7E

LAST_CONTROL_TRANSFER:  from 805b9cbb to 00000000

STACK_TEXT:
WARNING: Frame IP not in any known module. Following frames may be
wrong.
f88f2c9c 805b9cbb 81e826e8 8123a000 00000000 0x0
f88f2d58 805b9ee5 80000234 8123a000 81e826e8 nt!IopLoadDriver+0x5e1
f88f2d80 804ec5c8 80000234 00000000 822aeda0
nt!IopLoadUnloadDriver+0x43
f88f2dac 805f1828 f7718cf4 00000000 00000000 nt!ExpWorkerThread+0xe9
f88f2ddc 8050058e 804ec50d 00000001 00000000
nt!PspSystemThreadStartup+0x2e
00000000 00000000 00000000 00000000 00000000 nt!KiThreadStartup+0x16


FAILED_INSTRUCTION_ADDRESS:
+0
```

```
00000000 ??               ???


FOLLOWUP_IP:

nt!IopLoadDriver+5e1

805b9cbb 3bc3             cmp     eax,ebx


SYMBOL_STACK_INDEX:  1


SYMBOL_NAME:  nt!IopLoadDriver+5e1


MODULE_NAME:  nt


IMAGE_NAME:  ntoskrnl.exe


DEBUG_FLR_IMAGE_TIMESTAMP:  3e800a79


STACK_COMMAND:  .cxr ffffffff88f2828 ; kb


FAILURE_BUCKET_ID:  0x7E_NULL_IP_nt!IopLoadDriver+5e1


BUCKET_ID:  0x7E_NULL_IP_nt!IopLoadDriver+5e1

kd> .cxr ffffffff88f2828

eax=ffff99ea ebx=00000000 ecx=0000bb40 edx=8055f7a4 esi=e190049e
edi=81e826e8

eip=00000000 esp=f88f2ca0 ebp=f88f2cf0 iopl=0         nv up ei pl nz na
pe nc

cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000
efl=00010202

00000000 ??               ???
```

```
kd> kb

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

ChildEBP RetAddr  Args to Child

WARNING: Frame IP not in any known module. Following frames may be
wrong.

f88f2c9c 805b9cbb 81e826e8 8123a000 00000000 0x0

f88f2d58 805b9ee5 80000234 8123a000 81e826e8 nt!IopLoadDriver+0x5e1

f88f2d80 804ec5c8 80000234 00000000 822aeda0
nt!IopLoadUnloadDriver+0x43

f88f2dac 805f1828 f7718cf4 00000000 00000000 nt!ExpWorkerThread+0xe9

f88f2ddc 8050058e 804ec50d 00000001 00000000
nt!PspSystemThreadStartup+0x2e

00000000 00000000 00000000 00000000 00000000 nt!KiThreadStartup+0x16
```

The topmost entry in the above stack looks wrong. That is something you might encounter in a dump file. How would you work with that if you didn't know how the bug check had been produced?

1. Do **.frame 1**, to get to the frame for the caller of the mystery routine, nt!IopLoadDriver.

2. Go to the disassembly window, where the call by nt!IopLoadDriver is conveniently displayed, in instructions:

   ```
   8062da9e ff572c           call    dword ptr [edi+0x2c]

   8062daa1 3bc3             cmp     eax,ebx
   ```

3. The call was to the address in a doubleword pointed to by the Edi register's value plus 0x2C. That is the address you need. So, display the Edi register:

   ```
   kd> r edi

   Last set context:

   edi=81a2bb18
   ```

4. A little arithmetic:

   ```
   kd> ? 81a2bb18+0x2c

   Evaluate expression: -2120041660 = 81a2bb44
   ```

5. So the address is in storage at 0x81A2BB44:

```
kd> dd 81a2bb44 l1

81a2bb44  f87941a3
```

6. What's at that address?

```
kd> dt f87941a3

GsDriverEntry

 SIoctl!GsDriverEntry+0(

                  _DRIVER_OBJECT*,

                  _UNICODE_STRING*)
```

Thus you would have determined the real routine at the top of the stack.

Pseudo-registers

Pseudo-registers are variables that you can employ for various purposes. A number of pseudo-registers are predefined in meaning: $ra for the return point in the current entry on the call stack, $ip for the instruction pointer, $scopeip for the address of the current scope (local context, which makes available local variables in the current routine), $proc that points to the current EPROCESS, and so forth. These might be useful in conditional statements.

There are also pseudo-registers with operator-defined meaning, $t0 through $t19. These can be turned to various purposes, like counting breakpoints. A pseudo-register was put to that use in a real-life case where there were many updates to a piece of storage:

```
ba w4 81b404d8-18 "r$t0=@$t0+1;as /x ${/v:$$t0} @$t0;.block {.echo hit
# $$t0};ad ${/v:$$t0};dd 81b404d8-18 l1;k;!thread -1 0;!process -1 0"
```

The approximate meaning of the above is, When the doubleword at 0x81B404D8 is updated, update the pseudo-register $t0 as a hit counter, say what is the number of the hit and show the value at 0x81B404D8, the call stack, the current process and the current thread. (To understand details in the above, refer to aliasing below.)

Another illustrative use is from a support case where it was necessary to track the activity of Atapi.sys's DPC routine (Atapi.sys is a standard operating system driver). This routine was entered extremely often, but the investigating engineer knew that at a specific point there would be an interesting IRP about to be completed, and the variable *irp* pointed to that IRP. In another routine in Tape.sys (another standard operating system driver), there was a variable named *Irp* that pointed to the same IRP. The engineer wanted to stop in Tape.sys at just the right time, so he started by setting a one-time breakpoint in the Atapi.sys DPC:

```
bp /1 Atapi!IdeProcessCompletedRequest+0x3bd "dv irp; r$t0=@@(irp)"
```

That breakpoint's action was to set the pseudo-register $t0 to the value of the variable *irp*, which was the address of the IRP of interest. (It also displayed the value of *irp*.)

When that one-time breakpoint was hit, the engineer did this:

```
bp Tape!TapeIoCompleteAssociated+0x1c6 "j (@@(Irp)=$t0) '.echo stopped
at TAPE!TapeIoCompleteAssociated+0x1c6; dv Irp' ; 'g'"
```

This means: When this second breakpoint in Tape.sys is hit, if the local variable *Irp* matches $t0, put out some eye-catching information and display the value of *Irp*. If, on the other hand, *Irp* does not match $t0, just go. When this second breakpoint caused execution to stop, control had halted where the engineer desired.

Aliasing

It may be convenient to replace a set of characters with another set in a command string. One use is to define a short string for a long set of commands. For example,

```
kd> as Demo r; !process -1 0; k; !thread -1 0

kd> al

  Alias            Value

 -------          -------

 Demo             r; !process -1 0; k; !thread -1 0

kd> demo

Couldn't resolve error at 'emo'

kd> Demo

eax=00000001 ebx=001a6987 ecx=80571780 edx=ffd11118 esi=0000003e
edi=f8bcc776

eip=804df1c0 esp=8056f564 ebp=8056f574 iopl=0         nv up ei pl nz na
pe nc

cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000
efl=00000202

nt!RtlpBreakWithStatusInstruction:

804df1c0 cc               int     3

PROCESS 80579f60  SessionId: none  Cid: 0000    Peb: 00000000
ParentCid: 0000
```

```
    DirBase: 00039000  ObjectTable: e1000e78  HandleCount: 234.

    Image: Idle


ChildEBP RetAddr

8056f560 804e8682 nt!RtlpBreakWithStatusInstruction

8056f560 804e61ce nt!KeUpdateSystemTime+0x132

80579f60 00000000 nt!KiIdleLoop+0xe

THREAD 80579d00  Cid 0000.0000  Teb: 00000000 Win32Thread: 00000000
RUNNING on processor 0
```

Note that the alias and what replaces it are case-sensitive.

If you look back at pseudo-registers, you will now recognize that alias commands were used in the first example there. To begin, suppose that this was the actual command:

```
bp SioctlDeviceControl "r$t0=@$t0+1;as /x ${/v:$$t0} @$t0;.block {.echo
hit # $$t0};ad ${/v:$$t0};k ;!thread -1 0;!process -1 0;g"
```

Here's what would be put out in the WinDbg command window, with the effect of aliasing shown in red:

```
hit # 0x1

ChildEBP RetAddr

f747dc20 80a2675c SIoctl!SioctlDeviceControl

f747dc3c 80c70bed nt!IofCallDriver+0x62

f747dc54 80c71b0d nt!IopSynchronousServiceTail+0x159

f747dcf4 80c673aa nt!IopXxxControlFile+0x665

f747dd28 80afbbf2 nt!NtDeviceIoControlFile+0x28

f747dd28 7ffe0304 nt!_KiSystemService+0x13f

0006fdc8 04003bcb SharedUserData!SystemCallStub+0x4

0006fde8 04002314 ioctlapp!_ftbuf+0x1b

0006ff78 04002e02 ioctlapp!main+0x1e4

0006ffc0 77e4f38c ioctlapp!mainCRTStartup+0x14d
```

```
WARNING: Frame IP not in any known module. Following frames may be
wrong.

0006fff0 00000000 0x77e4f38c

THREAD feca2b88  Cid 0714.0e2c  Teb: 7ffde000 Win32Thread: 00000000
RUNNING on processor 0

PROCESS ff877b50  SessionId: 1  Cid: 0714    Peb: 7ffdf000  ParentCid:
0d04

    DirBase: 048f0000  ObjectTable: e2342440  HandleCount:  19.

    Image: ioctlapp.exe

...

hit # 0x2

...

hit # 0x3

...
```

The basic trick above is to define the command block associated with hitting the breakpoint in such a way that the imbedded alias definition isn't evaluated immediately but only when a breakpoint is hit. That is achieved by using the **${}** ("Alias Interpreter") token with the **/v** option to specify an alias not evaluated at specification (in the **bp** command, that is) and the **.block** ("Block") token to cause alias evaluation at the time of hitting the breakpoint and executing the associated commands. Finally, the **/x** option of the **as** command ensures 64-bit values are used, and the **ad** ensures the latest alias is cleaned up.

Script files and other ways to reduce work

It is possible to run a script file to execute a number of WinDbg commands. Consider as an example a dump file from an x64 system. Here the focus is on SCSI Request Blocks (SRBs) from a device driver called xStor.sys:

1. Use **!irpfind** (see the Debugging Tools for Windows help file) to find IRPs in the non-paged pool. You will get lines like this:

   ```
   fffffadfe5df1010 [fffffadfe5ee6760] irpStack: ( 4, 0)
   fffffadfe78cc060 [ \Driver\dmio] 0xfffffadfe6919470
   ```

   The value in red is the address of that specific IRP.

2. Copy those lines into a file.

3. In the file, select all lines containing xStor and put those lines into another file, debugtst1.txt. That yields lines like this one:

   ```
   fffffadfe5e4c9d0 [00000000] irpStack: ( f, 0)  fffffadfe783d050 [
   \Driver\xStor] 0xfffffadfe6919470
   ```

4. Editing debugtst1.txt, change each line:

   ```
   !irp fffffadfe5e4c9d0 1
   ```

   The **!irp** extension displays the IRP at the given address, including the IRP's major fields and its stack. Save debugtst1.txt.

5. Now, in WinDbg, give the command **$$<c:\temp\debugtst1.txt**. You will get a lot of output, starting with:

   ```
   1: kd> $$<c:\temp\debugtst1.txt

   1: kd> !irp fffffadfe5e4c9d0 1

   Irp is active with 2 stacks 2 is current (= 0xfffffadfe5e4cae8)

    Mdl = fffffadfe600f5e0 Thread 00000000:  Irp stack trace.

   Flags = 00000000

   ThreadListEntry.Flink = fffffadfe5e4c9f0

   ThreadListEntry.Blink = fffffadfe5e4c9f0

   IoStatus.Status = c00000bb

   IoStatus.Information = 00000000

   ...

   Tail.Apc = 0326cc00

   Tail.CompletionKey = 0326cc00

        cmd  flg cl Device   File     Completion-Context

    [  0, 0]   0  0 00000000 00000000 00000000-00000000

                                Args: 00000000 00000000 00000000 00000000

   >[  f, 0]   0 e1 fffffadfe783d050 00000000 fffffadfe3ee46d0-
   fffffadfe6869010 Success Error Cancel pending
   ```

```
                    \Driver\xStor   CLASSPNP!TransferPktComplete
                                    Args: fffffadfe6869130 00000000 00000000
        00000000
```

The value in red is, under these conditions, the SRB address for that IRP.

6. To get the SRBs formatted, create debugtst2.txt by finding and copying all lines in the above output that contain 'Args: ffff'. Then change each line to be like this:

   ```
   dt nt!SCSI_REQUEST_BLOCK fffffadfe6869130
   ```

   Note: Since the Microsoft symbol store contains only "public symbols", so nt!SCSI_REQUEST_BLOCK may not be available. For present purposes, think of it simply as a structure defined in a driver for which you have complete symbols.

7. You would save debugtst2.txt. Then in WinDbg you would do **$$<c:\temp\debugtst2.txt**. The output would be like this:

   ```
   1: kd> dt nt!SCSI_REQUEST_BLOCK fffffadfe6869130

      +0x000 Length           : 0x58

      +0x002 Function         : 0 ''

      +0x003 SrbStatus        : 0 ''

      +0x004 ScsiStatus       : 0 ''

      +0x005 PathId           : 0 ''

      +0x006 TargetId         : 0xff ''

      +0x007 Lun              : 0 ''


   ...


      +0x038 SrbExtension     : (null)

      +0x040 InternalStatus   : 0x21044d0

      +0x040 QueueSortKey     : 0x21044d0

      +0x044 Reserved         : 0

      +0x048 Cdb              : [16]  "*"
   ```

Thus, with a few minutes' work, you would have found and displayed all the SRBs of interest. It is possible to write a debugger extension to do the same thing, but for a one-off investigation, a simple script may be the better approach. You can take a script a step further by packaging commands with control logic to form a command program. The control is exercised through flow-operation tokens like **.if** and **.for**. For details about scripts and command programs, see "Run Script File" and "Using Debugger Command Programs" in the Debugging Tools for Windows help file.

A debugger extension is even more powerful, but it takes more time to write. An extension, written in C or C++ and built as a DLL, has available the full power of the debugger and its "engine." Many of commonly used commands such as **!process** are in fact extensions. The specifics of writing an extension are beyond the present scope. Refer to "Debugger Extensions" in the Debugging Tools for Windows help file.

Remote debugging

WinDbg (and KD) can be connected to a target to act as a server for a debugger instance acting as client, via TCP/IP or other protocol. A test system is connected via COMx or 1394 to a debugger, and the debugger is made into a server; then a developer can investigate problems or exercise function at a distance. This setup is very valuable in automated testing, allowing a person to investigate lab problems from his or her desk.

To get this capability, you can start the debugger with a command-line option to indicate its role:

```
windbg -server tcp:port=5555
```

or you can issue a command in WinDbg after it has started:

```
.server tcp:port=5005
```

Either method causes WinDbg to act as a debugging server, listening at TCP/IP port 5005.

In a different instance of WinDbg, do this to connect as a client:

```
.tcp:server=myserver,port=5005
```

To start a WinDbg client from scratch:

```
windbg -remote tcp:server=myserver,port=5005
```

COMx, named-pipe and SSL are other available protocols.

A few things to realize about remote debugging:

- If there is a firewall between the host system in one local or corporate network and the target system in another network, remote debugging is more complicated. See the Debugging Tools for Windows help file for details.
- Access to symbols and to source are based on the permissions of the person logged in at the remote server, not the permissions of the person that is running the client.

- The client sets source file locations by the **.lsrcpath** ("local source path") command, not **.srcpath**.

"Short" call stacks

WinDbg does its best to figure out the call stack, but sometimes it is defeated. Retrieving such a situation is one of the most difficult tasks facing the person who is debugging, since she or he must use her/his own knowledge to supplement WinDbg's expert knowledge. Be warned, then! Tough slogging ahead.

Consider this example from a dump file where there was a double-fault bug check (unexpected kernel-mode trap with first argument 0x00000008):

```
kd> k

ChildEBP RetAddr

be80cff8 00000000 hal!HalpClockInterruptPn+0x84
```

It would seem there was only one routine on the stack, an operating system clock-interrupt routine. It is a bit suspicious that that should have failed. So, begin by looking at the current thread:

```
kd> !thread

THREAD 88f108a0  Cid 8.54c  Teb: 00000000  Win32Thread: 00000000
RUNNING



...



Start Address rpcxdr!SunRpcGetRxStats (0xf7352702)

Stack Init be810000 Current be80fd34 Base be810000 Limit be80d000 Call
0
```

The stack starts at 0xBE810000 and goes down to 0xBE80D000 (three pages, which is normal). The stack base (ChildEBP) for the apparently failing clock routine is 0xBE80CFF8, which is beyond (below) the stack's end. Is it likely that a clock routine would have used more than the standard stack?

The detective work begins with looking for addresses in the stack that may indicate other routines. For this the usual tool is **dds** ("Display Words and Symbols"), to look for stored addresses (**dqs** and **dps** can be used, too; note that all three of these commands are case-sensitive). For present purposes ignore the clock-interrupt routine and instead focus on the stack up to but not including the routine interrupted by the clock routine. But don't ignore the clock routine entirely: You start with the fact that its base stack pointer (ChildEBP above) is 0xBE80CFF8.

Next look at 0xBE80CFF8 and see if anything interesting shows up (annotations below are shown with C-style comment delimiters):

```
2: kd> dds be80cff8 be80cff8+0x100

be80cff8  ????????           /* Invalid storage address. */

be80cffc  ????????           /* Invalid storage address. */

be80d000  00000000


...


be80d034  00000000

be80d038  00000020


...


be80d058  be80d084

be80d05c  00000000

be80d060  bfee03e0 zzznds+0x103e0

be80d064  00000008

be80d068  00000246

be80d06c  8a61c004

be80d070  bfbb7858

be80d074  88c1da28

be80d078  00000000

be80d07c  00000000

be80d080  00000000

be80d084  be80d0d8           /* Saved Ebp of zzznds+0xBED7, as
explained below. */
```

```
be80d088  bfedbed7 zzznds+0xbed7

...
```

Assume that the line with the identification "zzzndx+0x103E0" is the driver routine interrupted by the clock routine. You will notice the earlier line (higher stack address) with the identification "zzznds+0xBED7."

Now look at disassembled instructions a little before zzznds+0xBED7 (the point of call):

```
zzznds+0xbed0:

bfedbed0 53              push    ebx

bfedbed1 57              push    edi

bfedbed2 e8e7420000      call    zzznds+0x101be (bfee01be)
```

Note that the call is to zzznds+0x101BE, which is fairly close to the first identified line. Thus, the disassembly could well be the call to that.

Now disassemble zzznds+0x101BE to see how that call worked:

```
bfee01be 55              push    ebp          /* Save the caller's
EBP.         */

bfee01bf 8bec            mov     ebp,esp      /* Make the current ESP
our EBP.  */

bfee01c1 83ec0c          sub     esp,0xc      /* Adjust ESP by
subtracting 0xC. */

bfee01c4 53              push    ebx
```

Looking back to the output from **dds** above, you can see at 0xBE80D088 the caller's saved Ebp. But the operations of pushing down that Ebp (**push ebp** at 0xBFEE01BE) and of saving it at 0xBE80D088 mean that Esp after the pushing is 0xBE80D084, and since Esp becomes the current Ebp (instruction at 0xBFEE01BF), and since 0xC is subtracted from Esp at 0xBFEE01C1, the resulting Esp value at the instruction at 0XBFEE01C4 has to be 0xBE80D078.

Now you have determined the Ebp, Esp and Eip values for what was called by zzznds+0xBED7, namely, 0xBE80D084, 0xBE80D078 and 0xBFEE01C4, so you supply them to the **k** command for it to use them rather than try to discover values:

```
2: kd> k = 0xBE80D084 0xBE80D078 0xBFEE01C4
```

```
ChildEBP RetAddr

WARNING: Stack unwind information not available. Following frames may
be wrong.

be80d084 bfedbed7 zzznds+0x101c4

be80d0d8 bff6030f zzznds+0xbed7

be80d0fc 8046d778 SCSIPORT!SpStartIoSynchronized+0x139

be80d114 bff60e4f nt!KeSynchronizeExecution+0x28

be80d148 8006627b SCSIPORT!SpBuildScatterGather+0x249

be80d174 8041d30e hal!HalAllocateAdapterChannel+0x11b

be80d18c bff5f8c8 nt!IoAllocateAdapterChannel+0x28

be80d1bc 8041f73f SCSIPORT!ScsiPortStartIo+0x2ea

be80d1e0 bff5f4ec nt!IoStartPacket+0x6f

be80d214 bff601d0 SCSIPORT!ScsiPortFdoDispatch+0x26c

be80d22c bff622f7 SCSIPORT!SpDispatchRequest+0x70

be80d248 bff5e390 SCSIPORT!ScsiPortPdoScsi+0xef

be80d258 8041deb1 SCSIPORT!ScsiPortGlobalDispatch+0x1a


...
```

That is only part, the latest part, of the stack (there was more). But you should have the general idea.

To arrive at the parameters given to **k** immediately above, a good deal of detective work was necessary, involving a search of the stack and looking at code to see how the stack is built at a certain point. The work will vary from case to case. The lesson here is that if WinDbg's stack backtrace looks short, see if the kernel stack allotted to the failing thread is plausibly accounted for. If not, begin to dig in.

Unexpected change of thread context during stepping

If you spend an extended period of time in stepping through kernel code (with F10 or F11, for example), you may notice that control suddenly jumps to a point you didn't expect. This is the more likely if the code is running at an IRQL less than DISPATCH_LEVEL and you employ step-over (F10). If you knew that you were following control running under a specific thread and now checked the running thread, you would know for certain that the thread changed.

This behavior is normal, if disconcerting. The debugger accomplishes stepping by putting a debugging instruction (e.g., int 3 on x86) at the next instruction or the next statement (the debugging instruction isn't ordinarily visible to the person debugging). If the thread's quantum expires in moving from the current instruction/statement to the next, the operating system may dispatch a different thread, and that thread might encounter the debugging instruction, whereupon the debugger would get control. The debugger does not check that the thread is the same as that for the last stepping point but simply stops execution. In such a situation you may observe a jump. This scenario becomes more likely if the stepped-over code involves a great deal of processing, as might happen in stepping over an API that calls an API that calls an API and so forth.

There is no simple way of dealing with this behavior, which, to repeat, is expected. What can work for you is to be on the watch for unexpected jumps in stepping, and to check the current thread if you are suspicious. If you find the thread has switched, you should look back to find the last good point, restart the testing from square 1, set a one-time thread-qualified breakpoint at that last good point, and let things run until that breakpoint is reached. Then you can proceed: You are still exposed to switches, but you are that much further along the path of interest.

Getting more information

This brief essay did not plumb all the debugger's possibilities. For more information, consult the Debugging Tools for Windows help file. If you still have unanswered questions, bring them up in the public newsgroup microsoft.public.windbg (hosted by msnews.microsoft.com), or send email to windbgfb@microsoft.com.