CHAPTER 1

Windows 2000 Debugging Support

Because much of the information found in this book is of the so-called undocumented kind, some of it is available only by peeking inside the operating system code. The Windows 2000 Device Driver Kit (DDK) provides a powerful debugger that does a great job in this respect. This chapter begins with detailed step-by-step instructions to set up a full-fledged debugging environment on your machine. While reading the following chapters, you will frequently go back to the Kernel Debugger to extract operating system internals of various kinds. If you are becoming bored with the Kernel Debugger, you might want to tailor your own debugging tools. Therefore, this chapter also includes information about the documented and undocumented Windows 2000 debugging interfaces, including detailed inside information about Microsoft symbol files. It features two sample libraries with companion applications that list processes, process and system modules, and various kinds of symbol information buried inside the Windows 2000 symbol files. As a special bonus, you will find the first public documentation of the Microsoft Program Database (PDB) file format at the end of this chapter.

SETTING UP A DEBUGGING ENVIRONMENT

"Hey, I don't want to debug a Windows 2000 program. First of all, I want to *write* one!" you might shout out after reading this headline. "Right!" I say, "That's what you are going to do!" But why should you start the voyage by setting up a debugging environment? The answer is simple: The debugger is sort of a backdoor into the operating system. Of course, this has not been the primary intention of the persons who wrote this tool. However, every good debugger must be able to tell you something useful about the system while you are stepping through the execution of your own code or after your application has died unexpectedly. It is not quite acceptable to report an eight-digit crash address that points somewhere into the 4GB address space, leaving you figuring out alone what really happened. The debugger should at least tell you which module's code the offending code was executing last, and, ideally, it should also tell you the name of the function where the application passed away. Therefore, the debugger usually must know much more about the system than is printed in the programming manuals, and you can use this knowledge to explore the internals of the system.

Windows 2000 comes with two native debuggers: `WinDbg.exe` (pronounced like "WindBag") is a Win32 GUI application, and `i386kd.exe` is its console-mode equivalent. I have worked with both versions for some time and finally decided that `i386kd.exe` is the better one because it has a more powerful set of options. Recently, however, it seems that `WinDbg.exe` has improved, causing the people at Open Systems Resources (OSR) to include an article titled "There's a New WinDBG in Town—And It Doesn't Suck Anymore" in the May/June 2000 edition of *The NT Insider* (Open Systems Resources 2000). Nevertheless, all examples in this book that somehow involve a Windows 2000 debugger relate to `i386kd.exe`. As you might have guessed, the `i386` portion of the name refers to the target processor platform (Intel 386 family in this case, including all Pentium versions), and `kd` is short for Kernel Debugger. The Windows 2000 Kernel Debugger is a very powerful tool. For example, it knows how to make use of the symbol files distributed on the Windows 2000 setup CDs, and therefore can give you invaluable symbolic information about almost any address in system memory. Moreover, it will disassemble binary code, list hex dumps of memory contents in various formats, and even show you the layout of some key structures of the kernel. And it gives away this information for free—the debugger's command interface is fully documented in its online help.

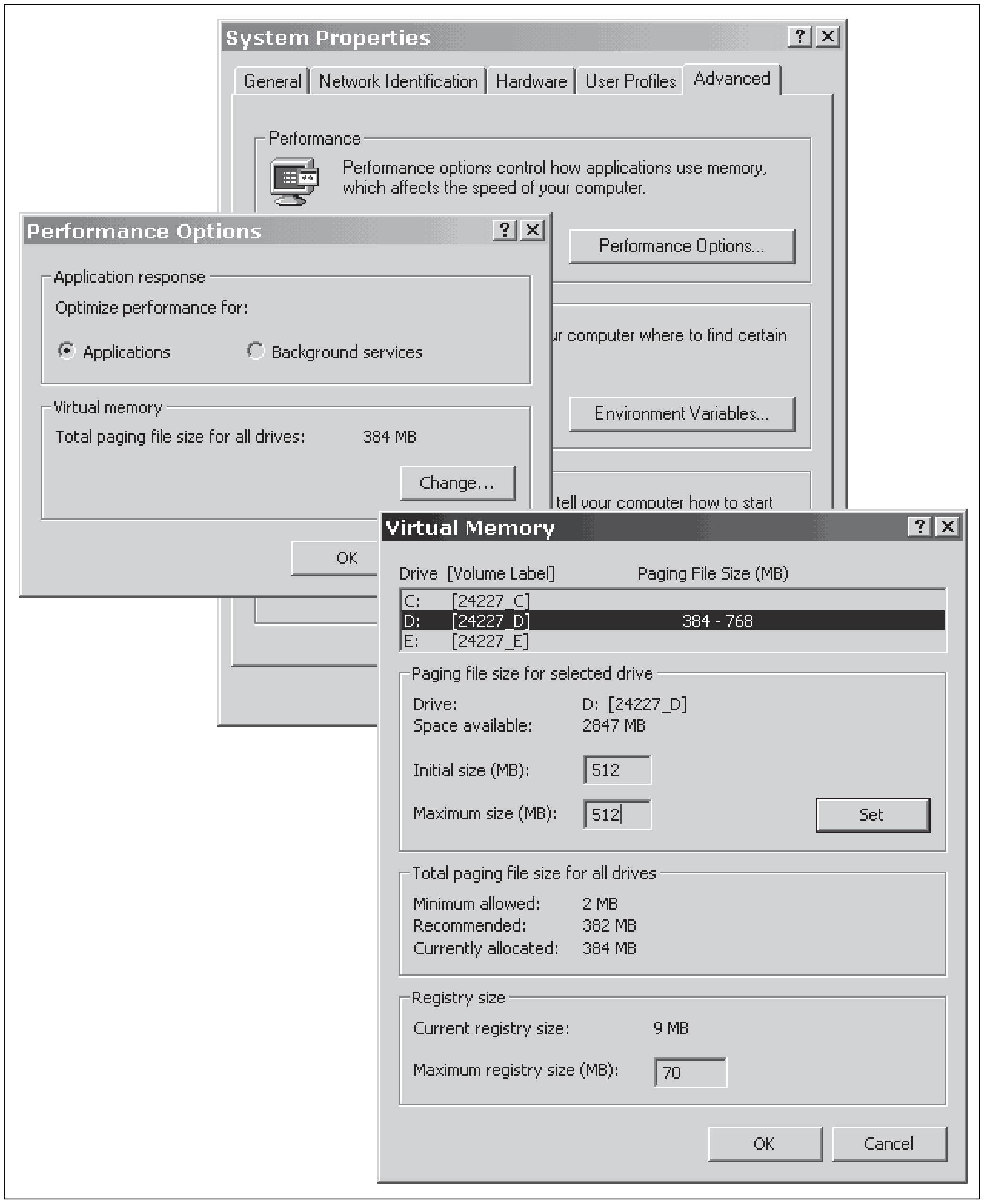
PREPARING FOR A CRASH DUMP

This is the good news. The bad news is that you have to do some preparatory work before the Kernel Debugger will obey you. The first obstacle is that debugging usually involves two separate machines connected by a cable—one running the debugger, the other one hosting the debuggee. However, there is a much easier way, eliminating the necessity of a second machine, if live debugging is not a requirement. For example, if a buggy application throws an unhandled exception causing the infamous NT "Blue Screen Of Death" (BSOD) to pop up, you can choose to save the memory image that was in effect right before the crash to a file and examine this *crash dump* after rebooting. This technique is usually called *post mortem* debugging (*post mortem* is Latin and means "after death"), and it is one of the preferred methods used throughout this book. Our primary task here is to explore system memory, and for most situations, it doesn't matter whether the memory under examination is alive or a snapshot of the last breath of a dead system. However, some interesting insights can be gained by peeking into the innards of a live system using a kernel-mode driver, but this is a topic to be saved for later chapters.

A crash dump is simply a copy of the current memory contents flushed to a disk file. Therefore, the size of a complete crash dump file is (almost) the same as the amount of physical RAM installed on the machine—in fact, it is a bit less than that. The crash dump is written by a special routine inside the kernel in the course of handling the fatal exception. However, this handler doesn't write the memory contents immediately to the target file. This is a good idea, because the disk file system might not be in good health after the crash. Instead, the image is copied to the page file storage, which is part of the system's memory manager. Therefore, you should increase the total size of your page files to at least twice the size of physical memory. Twice? Wouldn't the same size be enough? Of course—just enough for the crash dump. However, the system will attempt to copy the crash dump image to a real disk file during bootstrap, and this means that the system might run out of virtual memory if it can't free the page file memory occupied by the image in time. Usually, the system will cope with this situation, just throwing some annoying "low on virtual memory" warnings at you while thrashing the disk, but you can save a lot of time by making the page file large enough whenever you are expecting an increased probability of a Blue Screen.

That said, you should proceed now by starting the Windows 2000 Control Panel utility and changing the following settings:

- Increase the overall size of your page files to at least twice the amount of installed RAM. To this end, open the **System** applet, select the **Advanced** tab of the **System Properties** dialog, and click the **Performance Options**... button. In the **Virtual memory** frame, click the **Change...** button, and change the value in the **Maximum size (MB)** field if it doesn't match your physical memory configuration. Figure 1-1 is a sample snapshot taken on the system on which I am currently writing these lines. I have 256 MB of RAM inside my tower, so 512 MB is just enough. Click **Set** after changing the settings, and confirm all open dialogs except the **System Properties** by pressing their **OK** buttons.
- Next, configure the system to write a crash dump file on every Blue Screen. In the **System Properties** dialog, click the **Startup and Recovery...** button, and examine the **Write Debugging Information** options. You should select the **Complete Memory Dump** option from the drop-down list to get a faithful copy of the entire memory contents. In the **Dump File** box, enter the path and name of the file where the dump will be copied to from the page file. `%SystemRoot%\MEMORY.DMP` is a commonly used setting (Figure 1-2). Check or uncheck the **Overwrite any existing file** option according to your own preference, and confirm all open dialogs.



Performance options control how applications use memory, which affects the speed of your computer.

ur computer where to find certain

tell your computer how to start

Performance Options

Application response

Optimize performance for:

[✓ Applications] Background services

Virtual memory

Total paging file size for all drives: 384 MB

Virtual Memory

Drive [Volume Label]		Paging File Size (MB)
C:	[24227_C]	
D:	[24227_D]	384 - 768
E:	[24227_E]	

Paging file size for selected drive

Drive:	D: [24227_D]
Space available:	2847 MB
Initial size (MB):	512
Maximum size (MB):	512

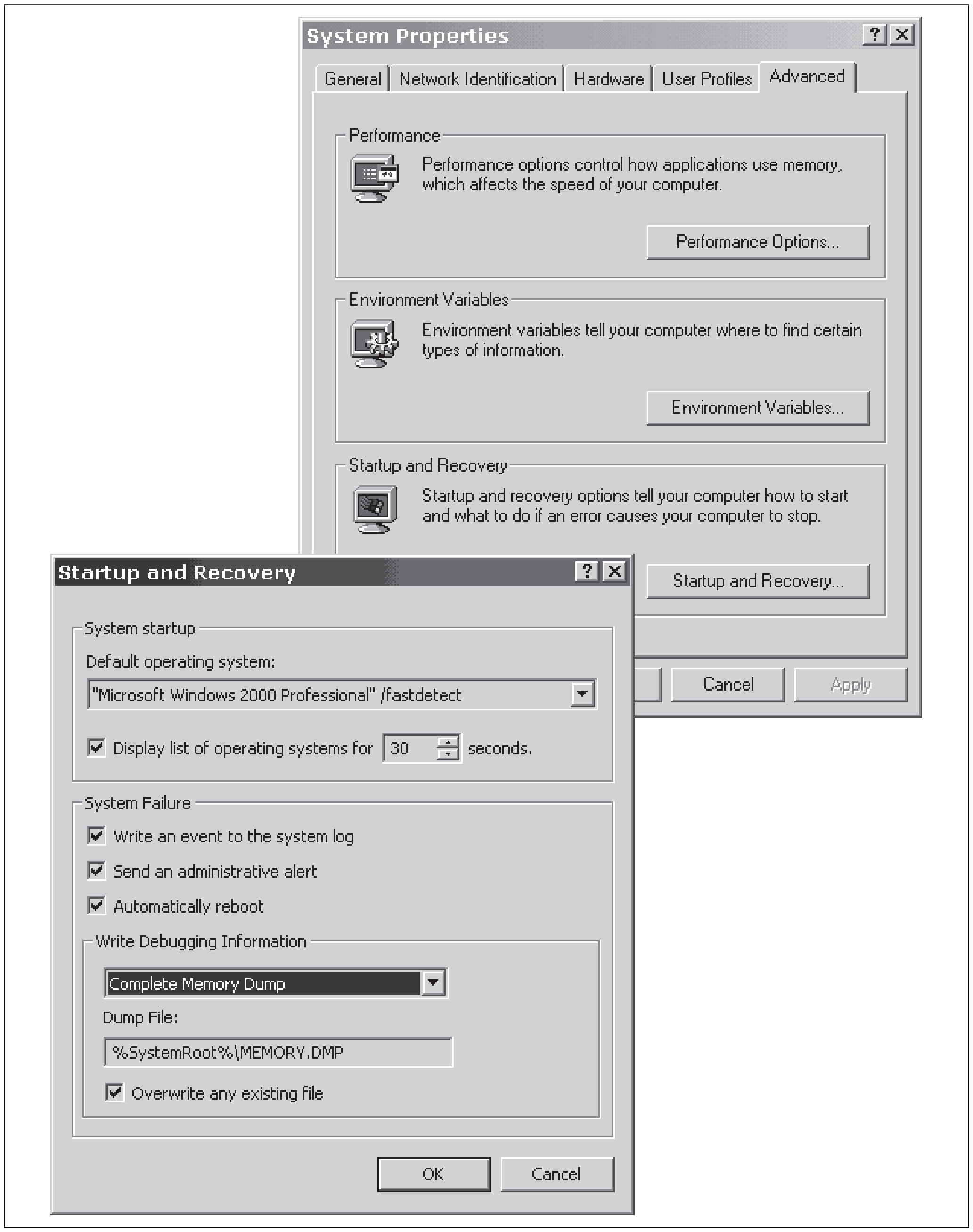
Total paging file size for all drives

Minimum allowed:	2 MB
Recommended:	382 MB
Currently allocated:	384 MB

Registry size

Current registry size:	9 MB
Maximum registry size (MB):	70

FIGURE 1-1. *Setting the Size of the Page File Storage*



System Properties

General | Network Identification | Hardware | User Profiles | Advanced

Performance

Performance options control how applications use memory, which affects the speed of your computer.

Performance Options...

Environment Variables

Environment variables tell your computer where to find certain types of information.

Environment Variables...

Startup and Recovery

Startup and recovery options tell your computer how to start and what to do if an error causes your computer to stop.

Startup and Recovery...

Cancel | Apply

Startup and Recovery

System startup

Default operating system:

"Microsoft Windows 2000 Professional" /fastdetect

☑ Display list of operating systems for 30 seconds.

System Failure

☑ Write an event to the system log

☑ Send an administrative alert

☑ Automatically reboot

Write Debugging Information

Complete Memory Dump

Dump File:

%SystemRoot%\MEMORY.DMP

☑ Overwrite any existing file

OK | Cancel

FIGURE 1-2. *Choosing Crash Dump Options*

Crashing the System

After having set up the system for a crash dump, it is time to do the most horrible thing in the life of a Windows 2000 system programmer: Let's crash the system! Usually, you will get the dreaded Blue Screen whenever Damocles' sword is hanging above your head—typically when a production deadline is due in a few hours. Now that you are willing to crash the system, you are probably unable to find any unstable piece of software that will do the job. Try David Solomon's neat trick described in the second edition of *Inside Windows NT*. This is his proposal:

> *"How can you reliably generate a crash dump? Just kill the Win32 subsystem process (csrss.exe) or the Windows NT logon process (winlogon.exe) with the Windows NT Resource Kit tool kill.exe. (You must have administrator privileges to do this.)" (Solomon [1998], p. 23.)*

Surprise, surprise! This trick doesn't work anymore on Windows 2000! On first sight, that's bad luck, but on the other hand, it is good news. What do you think about an operating system that can be trashed so easily by a tiny and simple tool officially distributed by Microsoft? In fact, it is good that Microsoft has closed this security gap. However, we are now in need of an alternative way to tear down the system. At this point, it is time for an old and simple NT rule: "If anything seems to be impossible in the Win32 world, just write a kernel-mode driver, and it will work out all right!" Windows 2000 manages Win32 applications very carefully. It constructs a wall between the application and the kernel, and anyone trying to cross this border will be shot without mercy. This is good for the overall stability of the system, but bad for programmers who need to write code that has to touch hardware. Contrary to DOS, where any application was allowed to do anything to the hardware, Windows 2000 is very picky in this respect. This doesn't mean that accessing hardware on Windows 2000 is impossible. Instead, this kind of access is restricted to a special kind of module called kernel-mode driver.

I can tell you now that I will present a short introduction to kernel-mode driver programming in Chapter 3. For now, it should suffice to say that crashing the system is one of the easiest things a kernel-mode driver can do. Windows 2000 doesn't provide an error recovery mechanism for drivers going berserk—even the faintest attempt to perform an illegal operation is immediately answered with a Blue Screen. Of course, the simplest and least dangerous violation of the rules is reading from an invalid memory address. Because the system explicitly catches all memory accesses through a NULL pointer, which is probably one of the most common errors in C programming, a NULL pointer read is the ideal operation to force a benign system crash. This is exactly what the `w2k_kill.sys` driver on the sample CD does. This very simple piece of software will also be one of the first kernel-mode driver projects presented in this book.

Listing 1-1 is a tiny excerpt from `w2k_kill.c`, containing nothing but the bad code that triggers the Blue Screen. When writing senseless code such as this, be aware that the brilliant optimizer built into Visual C/C++ might counteract your efforts. It tracks all code and tends to eliminate any instructions that don't have permanent side effects. In the example below, the optimizer's hands are tied because the `DriverEntry()` function insists on returning the value found at address zero as its return value. This means that this value has to be moved to CPU register `EAX`, and the easiest way to do this is by means of the `MOV EAX, [0]` instruction, which will throw the exception we have been waiting for.

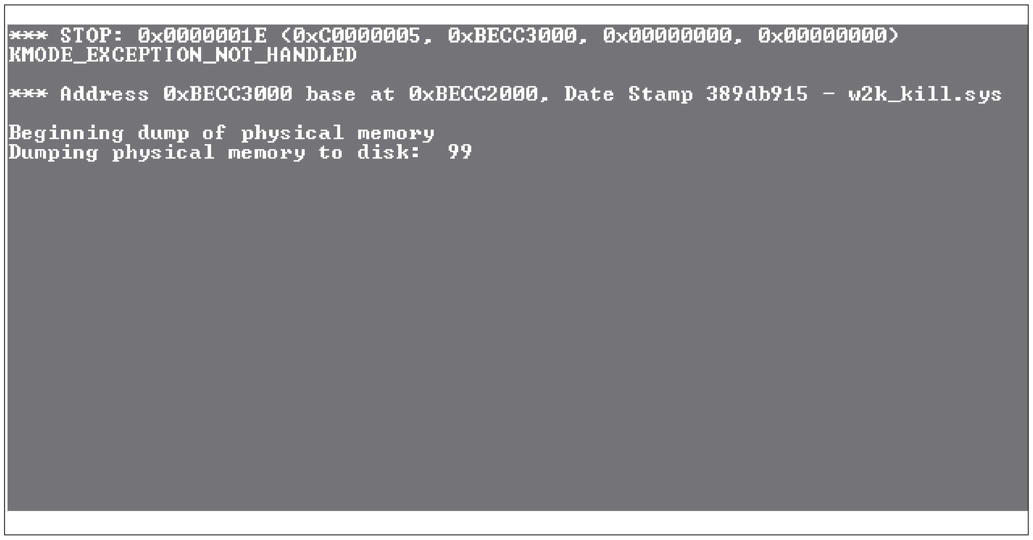
The `w2k_load.exe` application presented in Chapter 3 can be used to load and start the `w2k_kill.sys` driver. If you are mentally ready to kill your Windows 2000 system, proceed as follows:

- Close all applications.
- Insert the accompanying sample CD.
- Choose **Run…** from the **Start** menu.
- Enter `d:\bin\w2k_load w2k_kill.sys` into the edit box, replacing `d:` with the drive letter of your CD-ROM drive, and click **OK.**

After this click, `w2k_load.exe` will attempt to load the `w2k_kill.sys` file located in the CD's `\bin` directory. As soon as the `DriverEntry()` routine is executed, the Blue Screen will appear, with a message similar to the one shown in Figure 1-3, and you will see a counter on the screen being incremented from 0 to 100 (or so) while the memory contents are dumped to the page file storage. If you have checked the **Automatically reboot** option in the **Startup and Recovery** dialog (see Figure 1-2), the system will reboot immediately after the crash dump is finished. When the system is ready for logon, wait for some time until the disk LED is no longer flashing. It takes some time to copy the crash dump image from the page file storage to the target disk file defined in the **Startup and Recovery** options (see Figure 1-2), especially if you have plenty of physical memory. Disturbing the system in this phase, for example, by shutting it down too early, might yield an invalid crash dump file that will be refused by the Kernel Debugger.

```
NTSTATUS DriverEntry (PDRIVER_OBJECT pDriverObject,
                      PUNICODE_STRING puRegistryPath)
    {
    return *((NTSTATUS *) 0); // read through NULL pointer
    }
```

LISTING 1-1. *A NULL Pointer Read Operation in Kernel-Mode Crashes the System*



```
*** STOP: 0x0000001E (0xC0000005, 0xBECC3000, 0x00000000, 0x00000000)
KMODE_EXCEPTION_NOT_HANDLED

*** Address 0xBECC3000 base at 0xBECC2000, Date Stamp 389db915 - w2k_kill.sys

Beginning dump of physical memory
Dumping physical memory to disk:  99
```

FIGURE 1-3. *Execution of* `w2k_kill.sys` *Yields a Nice Blue Screen*

In Figure 1-3, the system displays the name of the module that contains the offending code (`w2k_kill.sys`), as well as the address of the instruction that caused the exception (`0xBECC3000`). This address will probably be different on your system, because it varies with the hardware configuration. Driver load addresses generally are not deterministic, similar to DLL load addresses. Please write down the indicated address—you will need it later after installing and configuring the Kernel Debugger.

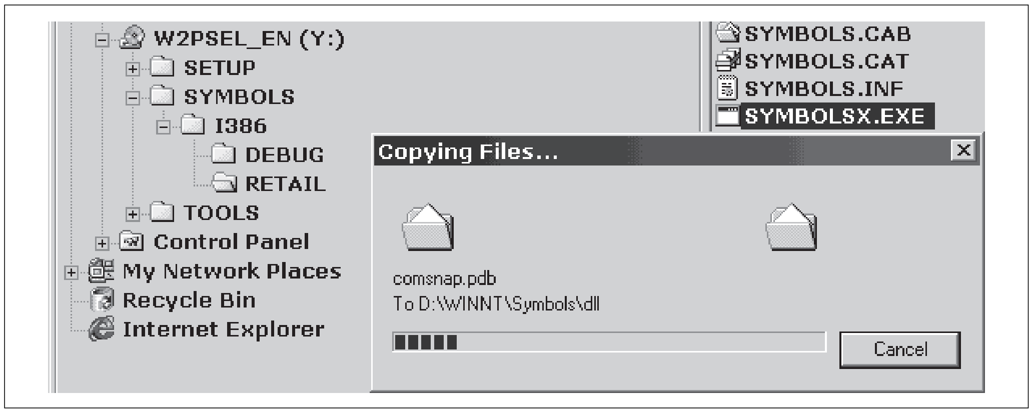
A short note of caution is appropriate here: Crashing the system intentionally is not something you should do every day. Although the offending `w2k_kill.sys` code itself is benign, the time of its execution might be unfortunate. If the NULL pointer read occurs while another thread is in the course of doing something important, the system might shut down before this thread has a chance to clean up. For example, the active desktop tends to complain after the reboot that something horrible has happened and that it needs to be restored. Therefore, carefully check that the machine isn't working on precious data and that all cached data has been flushed to disk before you crash the system. The best time is when the disk has calmed down after a bootstrap. Note that neither the author nor the publisher of this book shall be liable for any damages resulting from system crashes forced by the `w2k_kill.sys` driver.

INSTALLING THE SYMBOL FILES

After rebooting, you have a snapshot of a Windows 2000 system, including a bad kernel-mode driver, caught in the course of a NULL pointer read. Peeking into this file is as good as examining the memory of a live system. Of course, this snapshot is like a dead animal body—it can't react anymore to external stimuli, but that shouldn't worry you now. What comes next is the setup of the symbol files that shall be used by the Kernel Debugger while you are dissecting the crash dump.

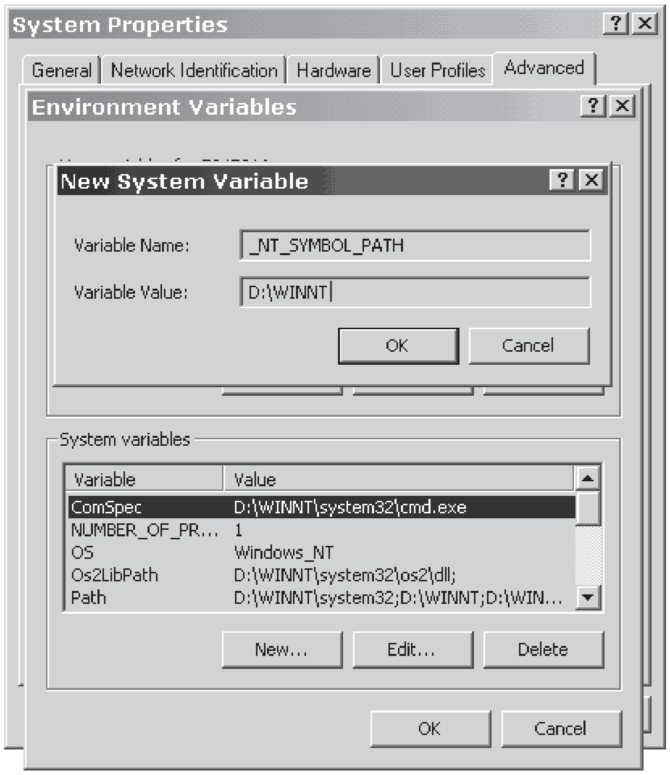
MSDN subscribers have to look for the symbol files on the CD named *Windows 2000 Customer Support—Diagnostic Tools,* which is part of the dark green *Development Platform (English)* CD set. Inserting the CD into the drive will start the Windows 2000 Internet Explorer with a file named `\DBG.HTM`. Here you can click on various setup options. If you are running the free build of Windows 2000, **Install retail symbols** is the correct choice. For the checked build, choose **Install debug symbols** instead. You can also use the classic symbol file setup by opening the Explorer and double-clicking the files `\SYMBOLS\I386\RETAIL\SYMBOLSX.EXE` (Figure 1-4) or `\SYMBOLS\I386\DEBUG\SYMBOLSX.EXE`, which are exactly the actions attached to the setup hyperlinks embedded in the `\DBG.HTM` file. The setup utility will copy several `.dbg` and `.pdb` files from the `SYMBOLS.CAB` archive to various subdirectories of the system's symbol root, which is named `%systemroot%\Symbols` by default. The `%systemroot%` token symbolizes the value of the environment variable `systemroot`, indicating the installation directory of the Windows 2000 system. In the example below, it is the `D:\WINNT` directory.

On startup, the Windows 2000 Kernel Debugger will try to locate the symbol files by evaluating the environment variable `_NT_SYMBOL_PATH` (note the leading underscore), so it is a good idea to define this variable right now. Again, you have to start the **System** applet from the Control Panel and select the **Advanced** tab, this time clicking the **Environment Variables…** button. Next, click the **New...** button in the **System variables** frame, and enter the **Variable Name:** and **Variable Value:** as shown in Figure 1-5, replacing `D:\WINNT` by the `%systemroot%` path of your system. After confirming all dialogs, symbol setup is complete.



comsnap.pdb
To D:\WINNT\Symbols\dll

FIGURE 1-4. *Installing the Windows 2000 Retail Symbols*



Variable Name: _NT_SYMBOL_PATH

Variable Value: D:\WINNT

Variable	Value
ComSpec	D:\WINNT\system32\cmd.exe
NUMBER_OF_PR...	1
OS	Windows_NT
Os2LibPath	D:\WINNT\system32\os2\dll;
Path	D:\WINNT\system32;D:\WINNT;D:\WIN...

FIGURE 1-5. *Defining the Environment Variable* _NT_SYMBOL_PATH

The Microsoft documentation is somewhat unclear about which directory path must be assigned to the _NT_SYMBOL_PATH variable. The kernel-mode debugging chapters of the DDK say that the Symbols subdirectory has to be included, yielding a value of d:\winnt\symbols or equivalent. In the Platform Software Development Kit (SDK) documentation of the dbghelp.dll library, the symbol path setup is described a bit differently:

> *"The library uses the symbol search path to locate debug symbols* (.dbg file) *for* .dll, .exe, *and* .sys *files by appending "*\symbols*" and "*\dll*" or "*\exe*" or "*\sys*" to the path. For example, the typical location of symbol files for* .dll *files is* c:\mysymbols\symbols\dll. *For* .exe *files, the location is* c:\mysymbols\symbols\exe.*"*
> *[...] "If you set the* _NT_SYMBOL_PATH or _NT_ALT_SYMBOL_PATH *environment variable, the symbol handler searches for symbol files in the following order:*

1. The current working directory of the application.
2. The `_NT_SYMBOL_PATH` environment variable.
3. The `_NT_ALT_SYMBOL_PATH` environment variable.
4. The `SYSTEMROOT` environment variable."
`(MSDN Library – April 2000 \ Platform SDK \ Base Services \ Debugging and Error Handling \ Debug Help Library \ About DbgHelp \ Symbol Handling \ Symbol Paths)`

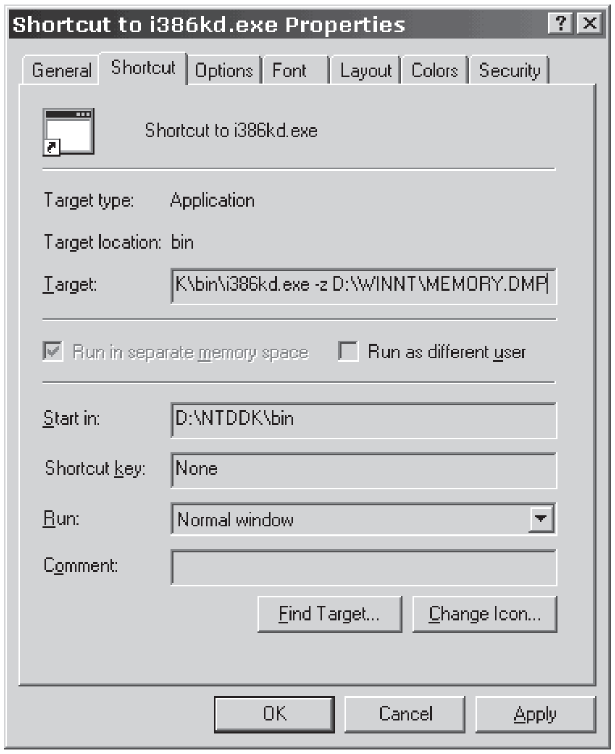
This sounds more like setting `_NT_SYMBOL_PATH` to `d:\winnt` rather than `d:\winnt\symbols`. To find out which point of view is correct, I tried both variants and was glad to see that it doesn't matter which one you choose. The Kernel Debugger finds the symbol files one way or another. If you suspect now that the `_NT_SYMBOL_PATH` value doesn't matter at all, try to set it to an invalid path—the debugger will refuse to run.

SETTING UP THE KERNEL DEBUGGER

The last step in the debugging environment setup is the installation and configuration of the Kernel Debugger. If you have already installed the Windows 2000 DDK, you can use the debuggers found in the `\NTDDK\bin` directory. The Kernel Debugger executable is named `i386kd.exe`. An alternative way is to install the debugging tools from the MSDN CD *Windows 2000 Customer Support—Diagnostic Tools,* from which you have already taken the symbol files. Just click on the **Install Debugging Tools** link on the setup page `\DBG.HTM`, or start the setup in the classic way by double-clicking `\TOOLS\I386\DBGPLUS.EXE` in the Explorer panel. This setup utility will copy the tools to a directory named `\Program Files\Debuggers\bin`.

After installing the Kernel Debugger, it is a good idea to create a shortcut that invokes `i386kd.exe` with the parameters you need. If you want to examine the crash dump file generated after the `w2k_kill.sys` Blue Screen, you can use the `-z` command line switch to specify the path of this file, directing the debugger to load this memory image at startup. Figure 1-6 illustrates typical shortcut properties.

Now everything is set up for the first debugging session. If you double-click the debugger's shortcut, you should see a console window like the one shown in Figure 1-7. The `kd>` prompt in front of the flashing cursor indicates that the Kernel Debugger is ready to accept commands. Before doing anything else, please check that the symbol search path displayed below the copyright banner is set to the correct location. If not, there is probably a typo in the environment variable specifying this path (see Figure 1-5). The start message also shows that the debugger has loaded three extension DLLs. `i386kd.exe` features a powerful extension mechanism that allows the basic command set to be augmented by custom commands implemented in a separate DLL. Because these additional commands have to be preceded by the "bang" character "!" to distinguish them from the built-in set, they usually are called *bang commands*. Some of them are extremely useful, as you will see later.



Shortcut to i386kd.exe

Target type: Application

Target location: bin

Target: K\bin\i386kd.exe -z D:\WINNT\MEMORY.DMP

[✓] Run in separate memory space [ ] Run as different user

Start in: D:\NTDDK\bin

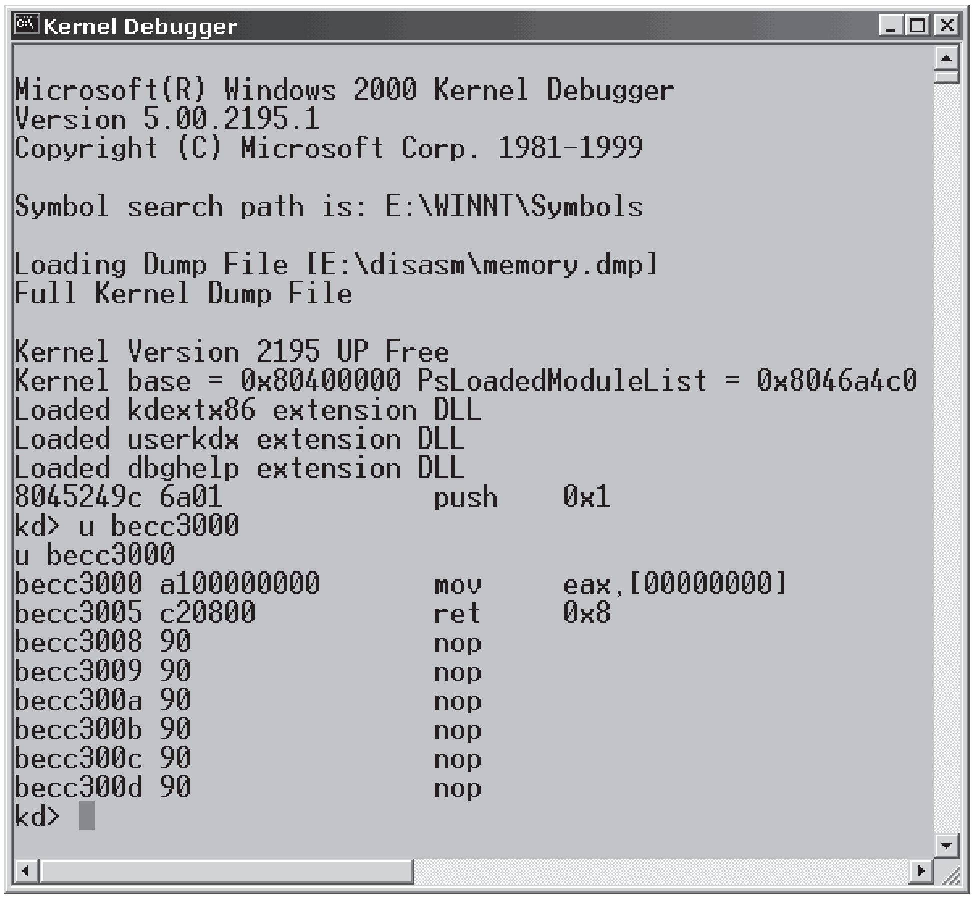
Shortcut key: None

Run: Normal window

Comment:

FIGURE 1-6. *Creating a Kernel Debugger Shortcut*

In Figure 1-7, I have entered one of the built-in commands: `u becc3000`. The `u` mnemonic means, of course, "unassemble," and `becc3000` is the hexadecimal start address where disassembly begins. By default, the number radix is 16, but you can change this setting with the `n` command, for example, `n 10` if you prefer decimal notation. You can always force a number to be interpreted as a hexadecimal by using the `0x` prefix borrowed from the C language. The address `becc3000` is the memory location where the `w2k_kill.sys` crash dump occurred (see Figure 1-3). Please try the `u` command with the address reported by your system after crashing. You should get a `mov eax,[00000000]` instruction, too, as shown in Figure 1-7, although the address is probably different. Otherwise, you are probably peeking into the wrong crash dump file—please check your Kernel Debugger shortcut in this case (see Figure 1-6). The `mov eax,[00000000]` instruction, loads a 32-bit value from the virtual address `0x00000000` to CPU register `EAX`, so it is obviously the implementation of the C expression `return *((NTSTATUS *) 0)` in Listing 1-1, and constitutes a NULL-pointer read operation. There is no special exception handler installed for this type of error, therefore, the system reports a `KMODE_EXCEPTION_NOT_HANDLED` error on the Blue Screen, as demonstrated by Figure 1-3. If you want, you can learn more about this common error code in *The NT Insider* (Open Systems Resources 1999b).



```
Microsoft(R) Windows 2000 Kernel Debugger
Version 5.00.2195.1
Copyright (C) Microsoft Corp. 1981-1999

Symbol search path is: E:\WINNT\Symbols

Loading Dump File [E:\disasm\memory.dmp]
Full Kernel Dump File

Kernel Version 2195 UP Free
Kernel base = 0x80400000 PsLoadedModuleList = 0x8046a4c0
Loaded kdextx86 extension DLL
Loaded userkdx extension DLL
Loaded dbghelp extension DLL
8045249c 6a01             push    0x1
kd> u becc3000
u becc3000
becc3000 a100000000       mov     eax,[00000000]
becc3005 c20800           ret     0x8
becc3008 90               nop
becc3009 90               nop
becc300a 90               nop
becc300b 90               nop
becc300c 90               nop
becc300d 90               nop
kd>
```

FIGURE 1-7. *Initiating a Kernel Debugger Session*

KERNEL DEBUGGER COMMANDS

Although the debugger commands are intended to be mnemonic, it is sometimes hard to recall them at the right time. Therefore, I have collected them in Appendix A, Table A-1, as a quick reference. This table is an edited version of the debugger's help output generated by the `?` command. The various types of arguments required for the commands are compiled in Table A-2.

As already mentioned, the Kernel Debugger can execute external commands known as *bang commands* that are implemented in one or more associated extension DLLs. Whenever a command name is prefixed by an exclamation mark (the so-called *bang* character), this name is looked up in the export lists of the loaded extension DLLs. If a match is found, the command is handed over to the DLL. Figure 1-7 shows that the Kernel Debugger loads the extensions `kdextx86.dll`, `userkdx.dll`, and `dbghelp.dll`, in this order. The latter is located in the same directory as the `i386kd.exe` application; the former pair is available in four versions: free versus checked build for Windows NT 4.0 (subdirectories `nt4fre` and `nt4chk`), and free versus checked build for Windows 2000 (subdirectories `w2kfre` and `w2kchk`), respectively. Normally, the debugger will use a default search order when locating the handler of a bang command. However, you can override the default by specifying a module name before the command name, separated by a dot. For example, both the `kdextx86.dll` and `userkdx.dll` extensions export a `help` command. Typing `!help` will yield the help screen of the `kdextx86.dll` module by default. To execute the help command of `userkdx.dll`, you have to type `!userkdx.help` (or `!userkdx.help -v` if you need more verbose help). By the way, you can write your own debugger extensions if you know the rules. An excellent how-to article can be found in *The NT Insider* (Open Systems Resources 1999a). It is targeting `WinDbg.exe` rather than `i386kd.exe`, but because both debuggers use the same extension DLLs, most of the information is applicable to `i386kd.exe` as well.

Tables A-3 and A-4 in Appendix A show the output generated by the `help` commands of `kdextx86.dll` and `userkdx.dll`, respectively, slightly corrected and heavily edited for better readability. You will notice that these tables list far more commands than documented in the Microsoft DDK, and some commands obviously have additional optional parameters not mentioned in the DDK documentation.

THE TOP TEN DEBUGGING COMMANDS

Tables A-1 to A-4 demonstrate in an impressive way that the Kernel Debugger and its standard extensions offer a large number of commands. Therefore, I will discuss in detail some of the commands that are most useful for the exploration of Windows 2000 internals.

`u`: *Unassemble Machine Code*

You have already used the `u` command after starting the Kernel Debugger to check whether the loaded crash dump file is OK. The `u` command has three forms:

1. `u <from>` disassembles eight machine instructions, starting at address `<from>`.
2. `u <from> <to>` starts disassembly at address `<from>`, and continues until reaching or transcending address `<to>`. The instruction at this address, if any, is not included in the listing.
3. `u` (without arguments) restarts disassembly from the address where a previous `u` command stopped (no matter whether it had arguments or not).

Of course, disassembling large code portions with this command is quite annoying, but it comes in handy if you just need to know what is occurring at a specific address. Perhaps the most interesting feature of the `u` command is its ability to resolve symbols referenced by the code—even internal symbols not exported by the target module. However, in disassembling complete Windows 2000 executables, using the `Multi-Format Visual Disassembler` on the companion CD is much more fun. More on this product will follow later in this chapter.

`db, dw`, and `dd`: Dump Memory `BYTEs, WORDs,` and `DWORDs`

If the memory contents you are currently interested in are binary data rather than machine code, the debugger's hex dump commands do a great job. Depending on the data types you are expecting at the source address, one of the variants `db` (for `BYTEs`), `dw` (for `WORDs`), or `dd` (for `DWORDs`) applies.

- `db` dumps a memory range in two panels. On the left-hand side, the contents are displayed as two-digit 8-bit hexadecimal quantities; the right-hand panel shows the same data in ASCII format.
- dw displays the contents of a memory range as four-digit 16-bit hexadecimal quantities. An ASCII panel is not included.
- `dd` displays the contents of a memory range as eight-digit 32-bit hexadecimal quantities. An ASCII panel is not included.

For this command set, the same arguments as for the `u` command can be used. Note, however, that the data located at the `<to>` address are always included in the hex dump listing. If no arguments are specified, the next 128-byte block is displayed.

`x`: Examine Symbols

The `x` command is very important. It can create lists of symbols compiled from the installed symbol files. It is typically used in one of the following three forms:

1. `x *!*` displays a list of all modules for which symbols can be browsed. After startup, only the `ntoskrnl.exe` symbols are available by default. The symbols of other modules can be added by issuing the `.reload` command.
2. `x <module>!<filter>` displays a list of symbols found in the symbol file of `<module>`, applying a `<filter>` that may contain the wildcards `?` and `*`. The `<module>` name must be one of the list yielded by the `x *!*` command. For example, `x nt!*` lists all symbols found in the kernel's symbol file `ntoskrnl.dbg`, and `x win32k!*` lists the symbols provided by `win32k.dbg`. If the debugger reports "Couldn't resolve 'x ...'", try the command again after reloading all symbols by means of the `.reload` command.

3. `x <filter>` displays a subset of all available symbols, matched against a `<filter>` expression. Essentially, this is a variant of the `x <module>!<filter>` command, in which the `<module>!` part has been omitted.

Along with the symbol names, the associated virtual addresses are shown. For function names, this is the function's entry point. For variables, it is a pointer to the base address of the variable. The most notable thing about this command is that its output includes many internal symbols, not just those found in the executable's export table.

`ln`: List Nearest Symbols

The `ln` command is certainly my favorite, because it gives quick and easy access to the installed symbol files. It is the ideal complement to the `x` command. Whereas the latter is great if you need an address listing of various operating system symbols, the `ln` command is used to look up individual symbols by address or name.

- `ln <address>` displays the name of the symbol found at or preceding the given `<address>`, as well as the next known symbol following this address.
- `ln <symbol>` resolves the given `<symbol>` name to its virtual address and then proceeds like the `ln <address>` command.

Like with the `x` command, the debugger is aware of all exported and several nonexported internal symbols. Therefore, it is an important aid for anyone who tries to make sense of unknown pointers occurring somewhere in a disassembly listing or hex dump. Note that the `u`, `db`, `dw`, and `dd` commands also accept symbols where addresses are expected.

`!processfields`: List `EPROCESS` Members

As the bang character preceding the name imples, this is a command from a debugger extension module—`kdextx86.dll`, in this case. This command displays the names and offsets of all members of the—formally undocumented—`EPROCESS` structure used by the kernel to represent processes, as shown in Example 1-1.

```
kd> !processfields
!processfields
 EPROCESS structure offsets:
    Pcb:                                       0x0
    ExitStatus:                                0x6c
    LockEvent:                                 0x70    LockCount:      0x80
    CreateTime:                                0x88
    ExitTime:                                  0x90
```

```
      LockOwner:                                   0x98
      UniqueProcessId:                             0x9c
      ActiveProcessLinks:                          0xa0
      QuotaPeakPoolUsage[0]:                       0xa8
      QuotaPoolUsage[0]:                           0xb0
      PagefileUsage:                               0xb8
      CommitCharge:                                0xbc
      PeakPagefileUsage:                           0xc0
      PeakVirtualSize:                             0xc4
      VirtualSize:                                 0xc8
      Vm:                                          0xd0
      DebugPort:                                   0x120
      ExceptionPort:                               0x124
      ObjectTable:                                 0x128
      Token:                                       0x12c
      WorkingSetLock:                              0x130
      WorkingSetPage:                              0x150
      ProcessOutswapEnabled:                       0x154
      ProcessOutswapped:                           0x155
      AddressSpaceInitialized:                     0x156
      AddressSpaceDeleted:                         0x157
      AddressCreationLock:                         0x158
      ForkInProgress:                              0x17c
      VmOperation:                                 0x180
      VmOperationEvent:                            0x184
      PageDirectoryPte:                            0x1f0
      LastFaultCount:                              0x18c
      VadRoot:                                     0x194
      VadHint:                                     0x198
      CloneRoot:                                   0x19c
      NumberOfPrivatePages:                        0x1a0
      NumberOfLockedPages:                         0x1a4
      ForkWasSuccessful:                           0x182
      ExitProcessCalled:                           0x1aa
      CreateProcessReported:                       0x1ab
      SectionHandle:                               0x1ac
      Peb:                                         0x1b0
      SectionBaseAddress:                          0x1b4
      QuotaBlock:                                  0x1b8
      LastThreadExitStatus:                        0x1bc
      WorkingSetWatch:                             0x1c0
   InheritedFromUniqueProcessId:                   0x1c8
      GrantedAccess:                               0x1cc
      DefaultHardErrorProcessing                   0x1d0
      LdtInformation:                              0x1d4
      VadFreeHint:                                 0x1d8
      VdmObjects:                                  0x1dc
      DeviceMap:                                   0x1e0
      ImageFileName[0]:                            0x1fc
      VmTrimFaultValue:                            0x20c
      Win32Process:                                0x214
      Win32WindowStation:                          0x1c4
```

EXAMPLE 1-1. *Cracking the* EPROCESS *Structure*

Although this command shows the members' offsets only, you can easily guess the corresponding types. For example, the `LockEvent` member is located at offset *0×70*, and the next member follows at offset *0×80*, so this member requires 16 bytes, which looks rather like a `KEVENT` structure. Don't worry if you don't know what a `KEVENT` is—I will discuss kernel object structures in Chapter 7.

`!threadfields`: List *ETHREAD* Members

This command is another great option offered by the `kdextx86.dll` debugger extension. Like the `!processfields` command, it displays the member names and offsets of yet another formally undocumented structure named `ETHREAD`, which represents threads. Example 1-2 shows a sample output.

`!drivers`: List Loaded Drivers

The `kdextx86.dll` goodie `!drivers` shows detailed information about all currently running kernel and file system modules. If a crash dump image is examined, this list reflects the system state at the time of the crash. Example 1-3 is an excerpt of a sample run on my machine. Note that the last line before the summary shows our bad Windows 2000 killer device at base address `0xBECC2000`, which is obviously one of the hexadecimal numbers reported on the Blue Screen after the `w2k_kill.sys` crash (see Figure 1-3).

```
kd> !threadfields
!threadfields
 ETHREAD structure offsets:
    Tcb:                                    0x0
    CreateTime:                             0x1b0
    ExitTime:                               0x1b8
    ExitStatus:                             0x1c0
    PostBlockList:                          0x1c4
    TerminationPortList:                    0x1cc
    ActiveTimerListLock:                    0x1d4
    ActiveTimerListHead:                    0x1d8
    Cid:                                    0x1e0
    LpcReplySemaphore:                      0x1e8
    LpcReplyMessage:                        0x1fc
    LpcReplyMessageId:                      0x200
    ImpersonationInfo:                      0x208
    IrpList:                                0x20c
    TopLevelIrp:                            0x214
    ReadClusterSize:                        0x21c
```

```
     ForwardClusterOnly:                              0x220
     DisablePageFaultClustering:                      0x221
     DeadThread:                                      0x222
     HasTerminated:                                   0x224
     GrantedAccess:                                   0x228
     ThreadsProcess:                                  0x22c
     StartAddress:                                    0x230
     Win32StartAddress:                               0x234
     LpcExitThreadCalled:                             0x238
     HardErrorsAreDisabled:                           0x239
```

EXAMPLE 1-2. *Cracking the* ETHREAD *Structure*

```
kd> !drivers
!drivers
Loaded System Driver Summary

Base       Code     Size       Data     Size       Driver Name    Creation Time
80400000   142dc0   (1291 kb)   4d680    (309 kb)  ntoskrnl.exe   Wed Dec 08 00:41:11 1999
80062000    13c40   (  79 kb)    34e0    ( 13 kb)       hal.dll   Sun Oct 31 00:48:14 1999
f0810000     1760   (   5 kb)    1000    (  4 kb)   BOOTVID.DLL   Thu Nov 04 02:24:33 1999
f0400000     bdc0   (  47 kb)    22a0    (  8 kb)       pci.sys   Thu Oct 28 01:11:08 1999
f0410000     99c0   (  38 kb)    18e0    (  6 kb)    isapnp.sys   Sat Oct 02 22:00:35 1999
f09c8000      760   (   1 kb)     520    (  1 kb)  intelide.sys   Fri Oct 29 01:20:03 1999
f0680000     42e0   (  16 kb)     e80    (  3 kb)   PCIIDEX.SYS   Thu Oct 28 01:02:19 1999
f0688000     64a0   (  25 kb)     a20    (  2 kb)  MountMgr.sys   Sat Oct 23 00:48:06 1999
bffe3000    192c0   ( 100 kb)    2b00    ( 10 kb)    ftdisk.sys   Mon Nov 22 20:36:23 1999
f0900000     12e0   (   4 kb)     640    (  1 kb)  Diskperf.sys   Fri Oct 01 02:30:40 1999
[...]
bf255000     fc40   (  63 kb)    2120    (  8 kb)    wdmaud.sys   Wed Oct 27 20:40:45 1999
f0670000     9520   (  37 kb)    1f40    (  7 kb)  sysaudio.sys   Mon Oct 25 21:28:14 1999
f094c000      d40   (   3 kb)     860    (  2 kb)    ParVdm.SYS   Tue Sep 28 05:28:16 1999
f0958000      a00   (   2 kb)     480    (  1 kb)   PfModNT.sys   Thu Dec 16 05:14:08 1999
bf0dd000    35520   ( 213 kb)    59e0    ( 22 kb)        rv.sys   Tue Nov 30 08:38:21 1999
bf191000     d820   (  54 kb)    1280    (  4 kb)      Cdfs.SYS   Mon Oct 25 21:23:52 1999
bed9a000    11f20   (  71 kb)    2ac0    ( 10 kb)     ipsec.sys   Tue Nov 30 08:08:54 1999
beaaf000        0   (   0 kb)       0    (  0 kb)     ATMFD.DLL   Header Paged Out
be9eb000    16f60   (  91 kb)    ccc0    ( 51 kb)    kmixer.sys   Wed Nov 10 07:52:30 1999
becc2000      200   (   0 kb)     a00    (  2 kb)  w2k_kill.sys   Sun Feb 06 19:10:29 2000
TOTAL:      79c660  (7793 kb)  15c160   (1392 kb)       (    0 kb     0 kb)
```

EXAMPLE 1-3. *Displaying Information about System Modules*

`!sel:` Examine Selector Values

If issued without arguments, the `!sel` command implemented by `kdextx86.dll` dumps the parameters of 16 consecutive memory selectors in ascending order. You can issue this command repeatedly until "Selector is invalid" is reported to get a list of all valid selectors (Example 1-4). Memory selector handling will be covered extensively in Chapter 4, and I will present sample code there that demonstrates how you can crack selectors in your own applications.

```
kd> !sel
!sel
0000  Bas=00000000 Lim=00000000 Bytes DPL=0 NP
0008  Bas=00000000 Lim=000fffff Pages DPL=0  P Code  RE A
0010  Bas=00000000 Lim=000fffff Pages DPL=0  P Data  RW A
0018  Bas=00000000 Lim=000fffff Pages DPL=3  P Code  RE A
0020  Bas=00000000 Lim=000fffff Pages DPL=3  P Data  RW A
0028  Bas=80244000 Lim=000020ab Bytes DPL=0  P TSS32    B
0030  Bas=ffdff000 Lim=00000001 Pages DPL=0  P Data  RW A
0038  Bas=00000000 Lim=00000fff Bytes DPL=3  P Data  RW A
0040  Bas=00000400 Lim=0000ffff Bytes DPL=3  P Data  RW
0048  Bas=00000000 Lim=00000000 Bytes DPL=0 NP
0050  Bas=80470040 Lim=00000068 Bytes DPL=0  P TSS32    A
0058  Bas=804700a8 Lim=00000068 Bytes DPL=0  P TSS32    A
0060  Bas=00022ab0 Lim=0000ffff Bytes DPL=0  P Data  RW A
0068  Bas=000b8000 Lim=00003fff Bytes DPL=0  P Data  RW
0070  Bas=ffff7000 Lim=000003ff Bytes DPL=0  P Data  RW
0078  Bas=80400000 Lim=0000ffff Bytes DPL=0  P Code  RE
kd> !sel
!sel
0080  Bas=80400000 Lim=0000ffff Bytes DPL=0  P Data  RW
0088  Bas=00000000 Lim=00000000 Bytes DPL=0  P Data  RW
0090  Bas=00000000 Lim=00000000 Bytes DPL=0 NP
0098  Bas=00000000 Lim=00000000 Bytes DPL=0 NP
00a0  Bas=814985a8 Lim=00000068 Bytes DPL=0  P TSS32    A
00a8  Bas=00000000 Lim=00000000 Bytes DPL=0 NP
00b0  Bas=00000000 Lim=00000000 Bytes DPL=0 NP
00b8  Bas=00000000 Lim=00000000 Bytes DPL=0 NP
00c0  Bas=00000000 Lim=00000000 Bytes DPL=0 NP
00c8  Bas=00000000 Lim=00000000 Bytes DPL=0 NP
00d0  Bas=00000000 Lim=00000000 Bytes DPL=0 NP
00d8  Bas=00000000 Lim=00000000 Bytes DPL=0 NP
00e0  Bas=f0430000 Lim=0000ffff Bytes DPL=0  P Code  RE A
00e8  Bas=00000000 Lim=0000ffff Bytes DPL=0  P Data  RW
00f0  Bas=8042dce8 Lim=000003b7 Bytes DPL=0  P Code  EO
00f8  Bas=00000000 Lim=0000ffff Bytes DPL=0  P Data  RW
```

EXAMPLE 1-4. *Displaying Selector Parameters*

SHUTTING DOWN THE DEBUGGER

You can kick the Kernel Debugger out of the system by simply closing the console window it is running in. However, the clean way to shut it down is using its `q` command, where "q" stands for—you guessed it—"quit."

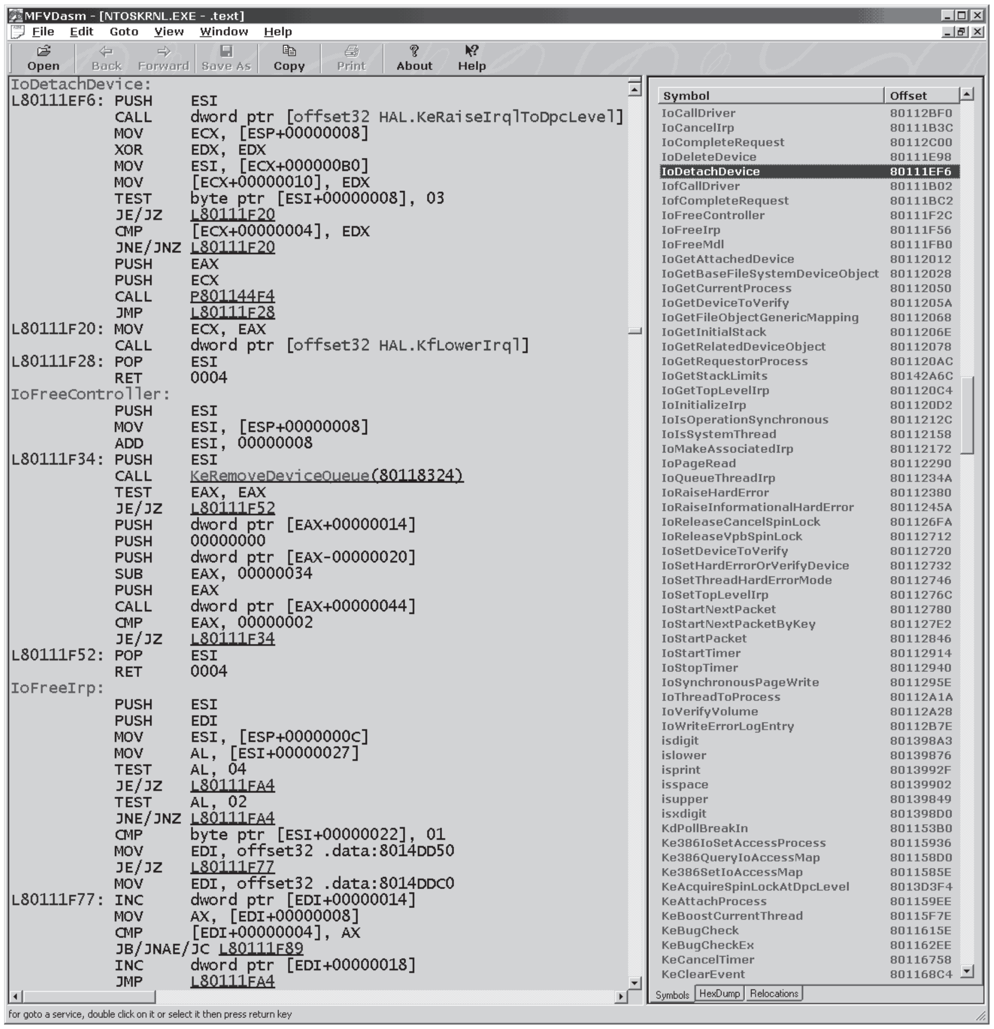
MORE DEBUGGING TOOLS

On the book's companion CD, you will find another pair of valuable debugging tools contributed by two "e-friends" of mine. I am very glad that they allowed me to put fully functional versions of their great tools onto the CD. Wayne J. Radburn's *PE and COFF File Viewer* (PEview) is a special FreeWare edition for the readers of this book. Jean-Louis Seigné's *Multi-Format Visual Disassembler* (MFVDasm) comes in an uncrippled but timed demo version. This section is a short introduction to both tools.

MFVDASM: THE MULTI-FORMAT VISUAL DISASSEMBLER

MFVDasm is not just a simple assembly listing generator. In fact, it is more an assembly code browser with several nice navigation features. Figure 1-8 shows a snapshot of an MFVDasm session in which I examined the Windows 2000 I/O Manager function `IoDetachDevice()`. Figure 1-8 does not show the color you would see on the screen. For example, all function labels, as well as jumps and calls to named destinations, are displayed red. Jumps and calls to anonymous addresses (i.e., addresses that are not associated with an exported symbol) are blue, and references to symbols dynamically imported from other modules are violet. All reachable destinations are underlined, indicating that you can click on them to scroll the code pane to the address. Using the **Back** and **Forward** buttons on the toolbar, you can navigate through the history of branches, much like flipping through the visited pages in an Internet browser.

In the right-hand pane, you can randomly select a symbol or target address to which you can jump. Of course, this list can be sorted by clicking on the column header buttons. On the lower edge of this pane, MFVDasm has tabs that allow switching between **Symbols, HexDump,** and **Relocations.** The hex dump view can be quite useful if you are disassembling a code section that contains embedded strings. MFVDasm doesn't choke on very large files such as `ntoskrnl.exe`, as some other popular disassemblers do, and, of course, the assembly code can be saved to a text file. Many more options are accessible via the main menu and the context menus that appear if you right-click on one of the window panes. If you need more information, visit Jean-Louis Seigné's MFVDasm home site at http://redirect.to/MFVDasm.


MFVDasm - [NTOSKRNL.EXE - .text]
File Edit Goto View Window Help
Open Back Forward Save As Copy Print About Help
Symbols HexDump Relocations

```
IoDetachDevice:
L80111EF6: PUSH    ESI
           CALL    dword ptr [offset32 HAL.KeRaiseIrqlToDpcLevel]
           MOV     ECX, [ESP+00000008]
           XOR     EDX, EDX
           MOV     ESI, [ECX+000000B0]
           MOV     [ECX+00000010], EDX
           TEST    byte ptr [ESI+00000008], 03
           JE/JZ   L80111F20
           CMP     [ECX+00000004], EDX
           JNE/JNZ L80111F20
           PUSH    EAX
           PUSH    ECX
           CALL    P801144F4
           JMP     L80111F28
L80111F20: MOV     ECX, EAX
           CALL    dword ptr [offset32 HAL.KfLowerIrql]
L80111F28: POP     ESI
           RET     0004
IoFreeController:
           PUSH    ESI
           MOV     ESI, [ESP+00000008]
           ADD     ESI, 00000008
L80111F34: PUSH    ESI
           CALL    KeRemoveDeviceQueue(80118324)
           TEST    EAX, EAX
           JE/JZ   L80111F52
           PUSH    dword ptr [EAX+00000014]
           PUSH    00000000
           PUSH    dword ptr [EAX-00000020]
           SUB     EAX, 00000034
           PUSH    EAX
           CALL    dword ptr [EAX+00000044]
           CMP     EAX, 00000002
           JE/JZ   L80111F34
L80111F52: POP     ESI
           RET     0004
IoFreeIrp:
           PUSH    ESI
           PUSH    EDI
           MOV     ESI, [ESP+0000000C]
           MOV     AL, [ESI+00000027]
           TEST    AL, 04
           JE/JZ   L80111FA4
           TEST    AL, 02
           JNE/JNZ L80111FA4
           CMP     byte ptr [ESI+00000022], 01
           MOV     EDI, offset32 .data:8014DD50
           JE/JZ   L80111F77
           MOV     EDI, offset32 .data:8014DDC0
L80111F77: INC     dword ptr [EDI+00000014]
           MOV     AX, [EDI+00000008]
           CMP     [EDI+00000004], AX
           JB/JNAE/JC L80111F89
           INC     dword ptr [EDI+00000018]
           JMP     L80111FA4
```

Symbol	Offset
IoCallDriver	80112BF0
IoCancelIrp	80111B3C
IoCompleteRequest	80112C00
IoDeleteDevice	80111E98
IoDetachDevice	80111EF6
IofCallDriver	80111B02
IofCompleteRequest	80111BC2
IoFreeController	80111F2C
IoFreeIrp	80111F56
IoFreeMdl	80111FB0
IoGetAttachedDevice	80112012
IoGetBaseFileSystemDeviceObject	80112028
IoGetCurrentProcess	80112050
IoGetDeviceToVerify	8011205A
IoGetFileObjectGenericMapping	80112068
IoGetInitialStack	8011206E
IoGetRelatedDeviceObject	80112078
IoGetRequestorProcess	801120AC
IoGetStackLimits	80142A6C
IoGetTopLevelIrp	801120C4
IoInitializeIrp	801120D2
IoIsOperationSynchronous	8011212C
IoIsSystemThread	80112158
IoMakeAssociatedIrp	80112172
IoPageRead	80112290
IoQueueThreadIrp	8011234A
IoRaiseHardError	80112380
IoRaiseInformationalHardError	8011245A
IoReleaseCancelSpinLock	801126FA
IoReleaseVpbSpinLock	80112712
IoSetDeviceToVerify	80112720
IoSetHardErrorOrVerifyDevice	80112732
IoSetThreadHardErrorMode	80112746
IoSetTopLevelIrp	8011276C
IoStartNextPacket	80112780
IoStartNextPacketByKey	801127E2
IoStartPacket	80112846
IoStartTimer	80112914
IoStopTimer	80112940
IoSynchronousPageWrite	8011295E
IoThreadToProcess	80112A1A
IoVerifyVolume	80112A28
IoWriteErrorLogEntry	80112B7E
isdigit	801398A3
islower	80139876
isprint	8013992F
isspace	80139902
isupper	80139849
isxdigit	801398D0
KdPollBreakIn	801153B0
Ke386IoSetAccessProcess	80115936
Ke386QueryIoAccessMap	801158D0
Ke386SetIoAccessMap	8011585E
KeAcquireSpinLockAtDpcLevel	8013D3F4
KeAttachProcess	801159EE
KeBoostCurrentThread	80115F7E
KeBugCheck	8011615E
KeBugCheckEx	801162EE
KeCancelTimer	80116758
KeClearEvent	801168C4

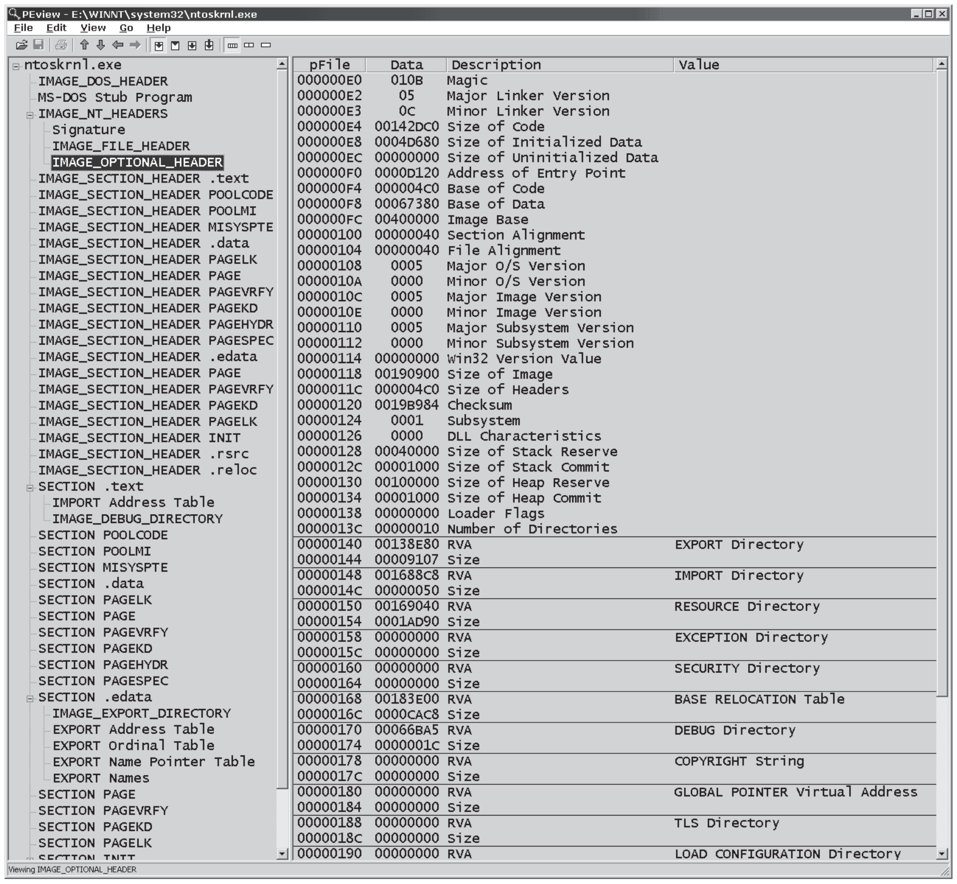
for goto a service, double click on it or select it then press return key

FIGURE 1-8. *MFVDasm Disassembling* `ntoskrnl.Io` DetachDevice()

PEVIEW—THE PE AND COFF FILE VIEWER

Although MFVDasm shows lots of details about the internal structure of a Portable Executable (PE) file, its strength is code browsing. On the other hand, PEview doesn't show you more than a hex dump of a code file section, but is considerably more detailed about the file structure. Figure 1-9 is a snapshot of PEview displaying the various parts of `ntoskrnl.exe` in tree form. If you click on a leaf node in the left-hand pane, the right-hand pane displays everything there is to know about the binary contents of this item. In Figure 1-9, I have selected the `IMAGE_OPTIONAL_HEADER` structure, which is a member of the `IMAGE_NT_HEADERS` structure located near the beginning of the executable.

If you take a closer look to the PEview toolbar, you see navigation arrows that allow scrolling through the file structure (vertical arrows) and the navigation history (horizontal arrows). The main menu and the toolbar offer many more display options that make using this tool a pleasure. Besides applications and DLLs, PEview can dissect several other file formats commonly encountered in debugging situations, such as object files, import libraries, and symbol files. More information is available at Wayne J. Radburn's Web site at http://www.magma.ca/~wjr/.


PEview - E:\WINNT\system32\ntoskrnl.exe
File
Edit
View
Go
Help
Viewing IMAGE_OPTIONAL_HEADER

```
ntoskrnl.exe
  IMAGE_DOS_HEADER
  MS-DOS Stub Program
  IMAGE_NT_HEADERS
    Signature
    IMAGE_FILE_HEADER
    IMAGE_OPTIONAL_HEADER
  IMAGE_SECTION_HEADER .text
  IMAGE_SECTION_HEADER POOLCODE
  IMAGE_SECTION_HEADER POOLMI
  IMAGE_SECTION_HEADER MISYSPTE
  IMAGE_SECTION_HEADER .data
  IMAGE_SECTION_HEADER PAGELK
  IMAGE_SECTION_HEADER PAGE
  IMAGE_SECTION_HEADER PAGEVRFY
  IMAGE_SECTION_HEADER PAGEKD
  IMAGE_SECTION_HEADER PAGEHYDR
  IMAGE_SECTION_HEADER PAGESPEC
  IMAGE_SECTION_HEADER .edata
  IMAGE_SECTION_HEADER PAGE
  IMAGE_SECTION_HEADER PAGEVRFY
  IMAGE_SECTION_HEADER PAGEKD
  IMAGE_SECTION_HEADER PAGELK
  IMAGE_SECTION_HEADER INIT
  IMAGE_SECTION_HEADER .rsrc
  IMAGE_SECTION_HEADER .reloc
  SECTION .text
    IMPORT Address Table
    IMAGE_DEBUG_DIRECTORY
  SECTION POOLCODE
  SECTION POOLMI
  SECTION MISYSPTE
  SECTION .data
  SECTION PAGELK
  SECTION PAGE
  SECTION PAGEVRFY
  SECTION PAGEKD
  SECTION PAGEHYDR
  SECTION PAGESPEC
  SECTION .edata
    IMAGE_EXPORT_DIRECTORY
    EXPORT Address Table
    EXPORT Ordinal Table
    EXPORT Name Pointer Table
    EXPORT Names
  SECTION PAGE
  SECTION PAGEVRFY
  SECTION PAGEKD
  SECTION PAGELK
```

```
pFile      Data      Description                   Value
000000E0   010B      Magic
000000E2   05        Major Linker Version
000000E3   0C        Minor Linker Version
000000E4   00142DC0  Size of Code
000000E8   0004D680  Size of Initialized Data
000000EC   00000000  Size of Uninitialized Data
000000F0   0000D120  Address of Entry Point
000000F4   000004C0  Base of Code
000000F8   00067380  Base of Data
000000FC   00400000  Image Base
00000100   00000040  Section Alignment
00000104   00000040  File Alignment
00000108   0005      Major O/S Version
0000010A   0000      Minor O/S Version
0000010C   0005      Major Image Version
0000010E   0000      Minor Image Version
00000110   0005      Major Subsystem Version
00000112   0000      Minor Subsystem Version
00000114   00000000  Win32 Version Value
00000118   00190900  Size of Image
0000011C   000004C0  Size of Headers
00000120   0019B984  Checksum
00000124   0001      Subsystem
00000126   0000      DLL Characteristics
00000128   00040000  Size of Stack Reserve
0000012C   00001000  Size of Stack Commit
00000130   00100000  Size of Heap Reserve
00000134   00001000  Size of Heap Commit
00000138   00000000  Loader Flags
0000013C   00000010  Number of Directories
00000140   00138E80  RVA                           EXPORT Directory
00000144   00009107  Size
00000148   001688C8  RVA                           IMPORT Directory
0000014C   00000050  Size
00000150   00169040  RVA                           RESOURCE Directory
00000154   0001AD90  Size
00000158   00000000  RVA                           EXCEPTION Directory
0000015C   00000000  Size
00000160   00000000  RVA                           SECURITY Directory
00000164   00000000  Size
00000168   00183E00  RVA                           BASE RELOCATION Table
0000016C   0000CAC8  Size
00000170   00066BA5  RVA                           DEBUG Directory
00000174   0000001C  Size
00000178   00000000  RVA                           COPYRIGHT String
0000017C   00000000  Size
00000180   00000000  RVA                           GLOBAL POINTER Virtual Address
00000184   00000000  Size
00000188   00000000  RVA                           TLS Directory
0000018C   00000000  Size
00000190   00000000  RVA                           LOAD CONFIGURATION Directory
```

FIGURE 1-9. *PEview Dissecting the PE File Structure of* `ntoskrnl.exe`

As mentioned in the Preface, Wayne writes his Win32 software in assembly language (ASM). Yes, this is not only possible but also quite easy if you have the necessary tools. In fact, ASM programming is much easier on the Win32 platform than it was in the old DOS and Windows 3.x days, because you can take full advantage of the CPU's 32-bit instruction set. Wayne actively supports Win32 ASM by providing extensive sample code on his Web site. I have been a die-hard ASM programmer myself, but I retired from it after discovering that the Microsoft Visual C optimizer does a much better job than a human ASM coder, because it can use all sorts of tricks that an ASM programmer should never use—the code would be unreadable and almost impossible to maintain. The results of my ASM efforts are publicly available in the form of a FreeWare package for the Microsoft Macro Assembler (MASM). It is called *Win32 Assembly Language Kit (WALK32)* and can be downloaded from my Web site. Just go to http://www.orgon.com/pub/asm/ and get all files that contain the letters "walk" in the file name. However, be aware that I have abandoned WALK32, and will not support or update it anymore.

WINDOWS 2000 DEBUGGING INTERFACES

The Kernel Debugger is a powerful tool for everyone interested in exploring the internals of the system. However, its user interface is somewhat poor, and sometimes you might wish to have even more powerful commands. Fortunately, Windows 2000 offers two fully documented debugging interfaces that enable you to add debugging functionality to your applications. These interfaces are far from luxurious, but they have the blessing of official documentation by Microsoft. In this section, I will take you on a short tour of these debugging interfaces, showing what they can do for you and how you can get the most out of them.

psapi.dll, imagehlp.dll, and dbghelp.dll

For a long time, Windows NT had been criticized for its lack of support for the Windows 95 `ToolHelp32` interface. Some of the critics were possibly not aware that Windows NT 4.0 came with an alternative debugging interface of its own, buried inside a system component named `psapi.dll,` distributed with the Win32 SDK. This DLL, together with `imagehlp.dll` and `dbghelp.dll,` comprise the officially documented debugging interfaces of Windows NT and 2000. The five letters PSAPI are the acronym of Process Status Application Programming Interface, and this interface comprises a set of 14 functions providing system information about device drivers, processes, memory usage and modules of a process, working sets, and memory-mapped files. `psapi.dll` supports both ANSI and Unicode strings.

The other pair of debugging DLLs, `imagehlp.dll` and `dgbhelp.dll,` cover a different range of tasks. Both export a similar set of functions, with the major difference that `imagehlp.dll` offers more functions, whereas `dbghelp.dll` is a redistributable component. This means that Microsoft allows you to put `dbghelp.dll` into the setup package of your applications if it relies on that DLL. If you choose to use `imagehlp.dll` instead, you must take the one that is currently installed on the target system. Both DLLs provide a rich set of functions for parsing and manipulating PE files. However, their most outstanding feature probably is their ability to extract symbols from the symbol files you have installed for use with the Kernel Debugger. To guide your decision as to which DLL you should choose, I have compiled all functions exported by `imagehlp.dll` and `dgbhelp.dll` in Table 1-1, where the middle and right-hand columns show which functions are not supported by which component. An entry of N/A means "not available."

TABLE 1-1. *Comparison of* `imagehlp.dll` *and* `dbghelp.dll`

NAME	**imagehlp.dll**	**dbghelp.dll**
BindImage		N/A
BindImageEx		N/A
CheckSumMappedFile		N/A
EnumerateLoadedModules		
EnumerateLoadedModules64		
ExtensionApiVersion	N/A	
FindDebugInfoFile		
FindDebugInfoFileEx		
FindExecutableImage		
FindExecutableImageEx		
FindFileInSearchPath		
GetImageConfigInformation		N/A
GetImageUnusedHeaderBytes		N/A
GetTimestampForLoadedLibrary		
ImageAddCertificate		N/A
ImageDirectoryEntryToData		
ImageDirectoryEntryToDataEx		
ImageEnumerateCertificates		N/A
ImageGetCertificateData		N/A
ImageGetCertificateHeader		N/A
ImageGetDigestStream		N/A
ImagehlpApiVersion		
ImagehlpApiVersionEx		

(continued)

TABLE 1-1. *(continued)*

NAME	imagehlp.dll	dbghelp.dll
ImageLoad		N/A
ImageNtHeader		
ImageRemoveCertificate		N/A
ImageRvaToSection		
ImageRvaToVa		
ImageUnload		N/A
MakeSureDirectoryPathExists		
MapAndLoad		N/A
MapDebugInformation		
MapFileAndCheckSumA		N/A
MapFileAndCheckSumW		N/A
ReBaseImage		N/A
ReBaseImage64		N/A
RemovePrivateCvSymbolic		N/A
RemovePrivateCvSymbolicEx		N/A
RemoveRelocations		N/A
SearchTreeForFile		
SetImageConfigInformation		N/A
SplitSymbols		N/A
StackWalk		
StackWalk64		
sym	N/A	
SymCleanup		
SymEnumerateModules		
SymEnumerateModules64		
SymEnumerateSymbols		
SymEnumerateSymbols64		
SymEnumerateSymbolsW		
SymFunctionTableAccess		
SymFunctionTableAccess64		
SymGetLineFromAddr		
SymGetLineFromAddr64		
SymGetLineFromName		
SymGetLineFromName64		

TABLE 1-1. *(continued)*

NAME	`imagehlp.dll`	`dbghelp.dll`
SymGetLineNext		
SymGetLineNext64		
SymGetLinePrev		
SymGetLinePrev64		
SymGetModuleBase		
SymGetModuleBase64		
SymGetModuleInfo		
SymGetModuleInfo64		
SymGetModuleInfoEx		
SymGetModuleInfoEx64		
SymGetModuleInfoW		
SymGetModuleInfoW64		
SymGetOptions		
SymGetSearchPath		
SymGetSymbolInfo		
SymGetSymbolInfo64		
SymGetSymFromAddr		
SymGetSymFromAddr64		
SymGetSymFromName		
SymGetSymFromName64		
SymGetSymNext		
SymGetSymNext64		
SymGetSymPrev		
SymGetSymPrev64		
SymInitialize		
SymLoadModule		
SymLoadModule64		
SymMatchFileName		
SymEnumerateSymbolsW64		
SymRegisterCallback		
SymRegisterCallback64		
SymRegisterFunctionEntryCallback		
SymRegisterFunctionEntryCallback64		
SymSetOptions		

(continued)

TABLE 1-1. *(continued)*

NAME	`imagehlp.dll`	`dbghelp.dll`
SymSetSearchPath		
SymUnDName		
SymUnDName64		
SymUnloadModule		
SymUnloadModule64		
TouchFileTimes		N/A
UnDecorateSymbolName		
UnMapAndLoad		N/A
UnmapDebugInformation		
UpdateDebugInfoFile		N/A
UpdateDebugInfoFileEx		N/A
WinDbgExtensionDllInit	N/A	

In the sample source code following in this section, I will demonstrate how `psapi.dll` and `imagehlp.dll` are used for the following programming tasks:

- Enumeration of all kernel components and drivers
- Enumeration of all processes currently managed by the system
- Enumeration of all modules loaded inside a process' virtual address space
- Enumeration of all symbols of a given component, if available

The `psapi.dll` interface is not particularly well designed. It provides a minimum of functionality, although it would have been easy to add a bit more convenience. Also, this DLL queries quite a bit of information from the kernel and then throws away most of it, leaving only tiny bits and pieces.

Because the `psapi.dll` and `imagehlp.dll` functions are not part of the standard Win32 API, their header files and import libraries are not automatically included in your Visual C/C++ projects. Therefore, the four directives in Listing 1-2 should show up somewhere in your source files. The first pair pulls in the required header files, and the latter pair establishes the dynamic links to the API functions exported by both DLLs.

```
#include <imagehlp.h>
#include <psapi.h>

#pragma comment (linker, "/defaultlib:imagehlp.lib")
#pragma comment (linker, "/defaultlib:psapi.lib")
```

LISTING 1-2. *Adding* `psapi.dll` *and* `imagehlp.dll` *to a Visual C/C++ Project*

SAMPLE CODE ON THE CD

On the CD accompanying this book, two sample projects are included that are built on `psapi.dll` and `imagehlp.dll`. One of them is `w2k_sym.exe`—a Windows 2000 symbol browser that extracts symbol names from an arbitrary symbol file, provided you have installed it (see Setting Up a Debugging Environment). The symbol table can be sorted by name, address, and data size, and a wildcard filter can be applied as well. As an additional bonus, `w2k_sym.exe` also lists active system module/driver names, running processes, and modules loaded inside any process. The other sample project is the debugging support library `w2k_dbg.dll`, which contains several convenient wrappers around `psapi.dll` and `imagehlp.dll` functions. `w2k_sym.exe` relies entirely on this DLL. The source code of these projects is located in the CD directories `\src\w2k_dbg` and `\src\w2k_sym`, respectively.

Table 1-2 lists the functions that are used by `w2k_dbg.dll`. The column A/W indicates for all functions involving strings whether ANSI (A) or 16-bit wide Unicode characters (W) are supported. As noted earlier, `psapi.dll` supports both ANSI and Unicode. Unfortunately, `imagehlp.dll` and `dbghelp.dll` aren't that clever and require 8-bit ANSI strings for several functions. This is somewhat annoying because a Windows 2000 debugging application usually will not run on Windows 9x and therefore could use Unicode characters without reservation. With `imagehlp.dll` included in your project, you will either have to use ANSI or occasionally convert Unicode strings back and forth. Because I definitely hate to work with 8-bit strings on a system capable of handling 16-bit characters, I have opted for the latter approach. All functions exported by `w2k_dbg.dll` that involve strings expect Unicode characters, so you don't need to be concerned about character size issues if you are reusing this DLL in your own Windows 2000 projects.

On the other hand, `imagehlp.dll` and `dbghelp.dll` have an interesting feature that `psapi.dll` lacks: They are already fit for Win64—the 64-bit Windows every developer is frightened of, because nobody really knows how difficult it will be to port Win32 applications to Win64. These DLLs export Win64 API functions, and that's OK—maybe we will be able to use them someday.

TABLE 1-2. *Debugging Functions Used by* w2k_dbg.dll

NAME	A/W	LIBRARY
EnumDeviceDrivers		psapi.dll
EnumProcesses		psapi.dll
EnumProcessModules		psapi.dll
GetDeviceDriverFileName	A/W	psapi.dll
GetModuleFileNameEx	A/W	psapi.dll
GetModuleInformation		psapi.dll
ImageLoad	A	imagehlp.dll
ImageUnload		imagehlp.dll
SymCleanup		imagehlp.dll
SymEnumerateSymbols	A/W	imagehlp.dll
SymInitialize	A	imagehlp.dll
SymLoadModule	A	imagehlp.dll
SymUnloadModule		imagehlp.dll

I don't go into `psapi.dll` and `imagehlp.dll` in depth. This book focuses on undocumented interfaces, and the interfaces of both DLLs are satisfactorily documented in the Platform SDK. However, I don't want to bypass them completely because they are closely related to the Windows 2000 Native API, discussed in Chapter 2. Moreover, `psapi.dll` is a good example of why an undocumented interface might be preferable to a documented one. Its interface is not only spartan and clumsy—it might even return inconsistent data in certain situations. If I had to write and sell a professional debugging tool, I would not build it on this DLL. The Windows 2000 kernel offers powerful, versatile, and much better-suited debugging API functions. However, they are almost completely undocumented. Fortunately, many system utilities provided by Microsoft make extensive use of this API, so it has undergone only slight changes across Windows NT versions. Yes, you have to revise and carefully test your software on every new NT release if you are using this API, but its benefits more than outweigh this drawback.

Most of the following code samples are taken from the source code of `w2k_dbg.dll`, found in the CD accompanying this book in the file `\src\w2k_dbg\w2k_dbg.c`. This library encapsulates several steps that you would have to take separately in convenient opaque functions that return rich information sets. The data is returned in properly sized, linked lists, with optional indexes imposed on them for sorting and other such functions. Table 1-3 lists the API functions exported by this DLL. It is a long list, and discussing each function is beyond the scope of this chapter, so you are encouraged to consult the source code of the companion application `w2k_sym.exe` for details about the typical usage (see `\src\w2k_sym\w2k_sym.c` on the CD).

TABLE 1-3. `w2k_dbg.dll` *API Function Set*

FUNCTION NAME	DESCRIPTION
dbgBaseDriver	Return the base address and size of a driver, given its path
dbgBaseModule	Return the base address and size of a DLL module
dbgCrc32Block	Compute the CRC32 of a memory block
dbgCrc32Byte	Bytewise computation of a CRC32
dbgCrc32Start	CRC32 preconditioning
dbgCrc32Stop	CRC32 postconditioning
dbgDriverAdd	Add a driver entry to a list of drivers
dbgDriverAddresses	Return an array of driver addresses (`EnumDeviceDrivers()` wrapper)
dbgDriverIndex	Create an indexed (and optionally sorted) driver list
dbgDriverList	Create a flat driver list
dbgFileClose	Close a disk file
dbgFileLoad	Load the contents of a disk file to a memory block
dbgFileNew	Create a new disk file
dbgFileOpen	Open an existing disk file
dbgFileRoot	Get the offset of the root token in a file path
dbgFileSave	Save a memory block to a disk file
dbgFileUnload	Free a memory block created by `dbgFileLoad()`
dbgIndexCompare	Compare two entries referenced by an index (used by `dbgIndexSort()`)
dbgIndexCreate	Create a pointer index on an object list
dbgIndexCreateEx	Create a sorted pointer index on an object list
dbgIndexDestroy	Free the memory used by an index and its associated list
dbgIndexDestroyEx	Free the memory used by a two-dimensional index and its associated lists
dbgIndexList	Create a flat copy of a list from its index
dbgIndexListEx	Create a flat copy of a two-dimensional list from its index
dbgIndexReverse	Reverse the order of the list entries referenced by an index
dbgIndexSave	Save the memory image of an indexed list to a disk file
dbgIndexSaveEx	Save the memory image of a two-dimensional indexed list to a disk file
dbgIndexSort	Sort the list entries referenced by an index by address, size, ID, or name
dbgListCreate	Create an empty list
dbgListCreateEx	Create an empty list with reserved space
dbgListDestroy	Free the memory used by a list
dbgListFinish	Terminate a sequentially built list and trim any unused memory
dbgListIndex	Create a pointer index on an object list

(continued)

TABLE 1-3. *(continued)*

FUNCTION NAME	DESCRIPTION
dbgListLoad	Create a list from a disk file image
dbgListNext	Update the list header after adding an entry
dbgListResize	Reserve memory for additional list entries
dbgListSave	Save the memory image of a list to a disk file
dbgMemoryAlign	Round up a byte count to the next 64-bit boundary
dbgMemoryAlignEx	Round up a string character count to the next 64-bit boundary
dbgMemoryBase	Query the internal base address of a heap memory block
dbgMemoryBaseEx	Query the internal base address of an individually tagged heap memory block
dbgMemoryCreate	Allocate a memory block from the heap
dbgMemoryCreateEx	Allocate an individually tagged memory block from the heap
dbgMemoryDestroy	Return a memory block to the heap
dbgMemoryDestroyEx	Return an individually tagged memory block to the heap
dbgMemoryReset	Reset the memory usage statistics
dbgMemoryResize	Change the allocated size of a heap memory block
dbgMemoryResizeEx	Change the allocated size of an individually tagged heap memory block
dbgMemoryStatus	Query the memory usage statistics
dbgMemoryTrack	Update the memory usage statistics
dbgModuleIndex	Create an indexed (and optionally sorted) process module sub-list
dbgModuleList	Create a flat process module sub-list
dbgPathDriver	Build a default driver path specification
dbgPathFile	Get the offset of the file name token in a file path
dbgPrivilegeDebug	Request the debug privilege for the calling process
dbgPrivilegeSet	Request the specified privilege for the calling process
dbgProcessAdd	Add a process entry to a list of processes
dbgProcessGuess	Guess the default display name of an anonymous system process
dbgProcessIds	Return an array of process IDs (`EnumProcesses()` wrapper)
dbgProcessIndex	Create an indexed (and optionally sorted) process list
dbgProcessIndexEx	Create a two-dimensional indexed (and optionally sorted) process/module list
dbgProcessList	Create a flat process list
dbgProcessModules	Return a list of process module handles (`EnumProcessModules()` wrapper)
dbgSizeDivide	Divide a byte count by a power of two, optionally rounding up or down

TABLE 1-3. *(continued)*

FUNCTION NAME	DESCRIPTION
dbgSizeKB	Convert bytes to KB, optionally rounding up or down
dbgSizeMB	Convert bytes to MB, optionally rounding up or down
dbgStringAnsi	Convert a Unicode string to ANSI
dbgStringDay	Get the name of a day given a day-of-week number
dbgStringMatch	Apply a wildcard filter to a string
dbgSymbolCallback	Add a symbol entry to a list of symbols (called by `SymEnumerateSymbols()`)
dbgSymbolIndex	Create an indexed (and optionally sorted) symbol list
dbgSymbolList	Create a flat symbol list
dbgSymbolLoad	Load a module's symbol table
dbgSymbolLookup	Look up a symbol name and optional offset given a memory address
dbgSymbolUnload	Unload a module's symbol table

ENUMERATING SYSTEM MODULES AND DRIVERS

`psapi.dll` can be instructed to return a list of active kernel modules currently residing in memory. This is a fairly simple task. The `psapi.dll` function `EnumDeviceDrivers()` receives an array of `PVOID` slots, which it fills with the image base addresses of the active kernel-mode drivers, including the basic kernel modules `ntdll.dll`, `ntoskrnl.exe`, `win32k.sys`, `hal.dll`, and `bootvid.dll`. The reported values are the virtual memory addresses where the contents of the respective executable files have been mapped. If you examine the first few bytes at these addresses with the Kernel Debugger or some other debugging tool, you will clearly recognize the good old DOS stub program, starting with Mark Zbikowski's famous initials "MZ," and containing the message text, "This program cannot be run in DOS mode" or something similar. Listing 1-3 shows a sample invocation of `EnumDeviceDrivers()`, including this function's prototype at the top for your convenience.

`EnumDeviceDrivers()` expects three arguments: an array pointer, an input size value, and a pointer to an output size variable of type `DWORD`. The second argument specifies the size of the supplied image address array in bytes (!), and the third argument receives the number of bytes copied to the array. Therefore, you have to divide the resulting size by `sizeof (PVOID)` to obtain the number of addresses copied to the array. Unfortunately, this function doesn't help you to find out how large the output array should be, although it actually knows how many drivers are running. It just tells you how many bytes were returned, and, if the buffer is too small, it conceals the number of bytes that didn't fit in. Therefore, you have to employ a dull trial-and-error loop to determine the correct size, as demonstrated in Listing 1-3, assuming that the

```
BOOL WINAPI EnumDeviceDrivers (PVOID *lpImageBase,
                               DWORD  cb,
                               PDWORD lpcbNeeded);

PPVOID WINAPI dbgDriverAddresses (PDWORD pdCount)
    {
    DWORD  dSize;
    DWORD  dCount = 0;
    PPVOID ppList = NULL;

    dSize = SIZE_MINIMUM * sizeof (PVOID);

    while ((ppList = dbgMemoryCreate (dSize)) != NULL)
        {
        if (EnumDeviceDrivers (ppList, dSize, &dCount) &&
            (dCount < dSize))
            {
            dCount /= sizeof (PVOID);
            break;
            }
        dCount = 0;
        ppList = dbgMemoryDestroy (ppList);
        if ((dSize <<= 1) > (SIZE_MAXIMUM * sizeof (PVOID))) break;
        }
    if (pdCount != NULL) *pdCount = dCount;
    return ppList;
    }
```

LISTING 1-3. *Enumerating System Module Addresses*

data are incomplete whenever the returned size is equal to the size of the array. The code starts out with a reasonable minimum size of 256 entries, represented by the constant `SIZE_MINIMUM`. This is usually enough, but, if not, the buffer size is doubled on every new trial until all pointers are retrieved or the maximum size of 65,536 entries (`SIZE_MAXIMUM`) would be exceeded. The memory buffer is allocated and freed by the helper functions `dbgMemoryCreate()` and `dbgMemoryDestroy()`, which are just fancy wrappers around the standard Win32 functions `LocalAlloc()` and `LocalFree()`, and therefore aren't reprinted here.

Listing 1-4 shows a possible implementation of `EnumDeviceDrivers()`. Note that this is *not* the original source code from `psapi.dll`. It is a random sequence of characters that happens to yield equivalent binary code if fed to a C compiler. To keep things clear and simple, I have omitted some distracting details found in the original code, such as Structured Exception Handling (SEH) clauses, for example. At

```
BOOL WINAPI EnumDeviceDrivers (PVOID *lpImageBase,
                               DWORD  cb,
                               DWORD *lpcbNeeded)
    {
    SYSTEM_MODULE_INFORMATION_N(1) smi;
    PSYSTEM_MODULE_INFORMATION     psmi;
    DWORD                          dSize, i;
    NTSTATUS                       ns;
    BOOL                           fOk = FALSE;

    ns = NtQuerySystemInformation (SystemModuleInformation,
                                   &smi, sizeof (smi), NULL);

    if ((ns == STATUS_SUCCESS) ||
        (ns == STATUS_INFO_LENGTH_MISMATCH))
        {
        dSize = sizeof (SYSTEM_MODULE_INFORMATION) +
                (smi.dCount * sizeof (SYSTEM_MODULE));

        if ((psmi = LocalAlloc (LMEM_FIXED, dSize)) != NULL)
            {
            ns = NtQuerySystemInformation (SystemModuleInformation,
                                           psmi, dSize, NULL);

            if (ns == STATUS_SUCCESS)
                {
                for (i = 0; (i < psmi->dCount) &&
                            (i < cb / sizeof (DWORD)); i++)
                    {
                    lpImageBase [i] = psmi->aModules [i].pImageBase;
                    }
                *lpcbNeeded = i * sizeof (DWORD);
                fOk         = TRUE;
                }
            LocalFree (psmi);

            if (!fOk) SetLastError (RtlNtStatusToDosError (ns));
            }
        }
    else
        {
        SetLastError (RtlNtStatusToDosError (ns));
        }
    return fOk;
    }
```

LISTING 1-4. *Sample Implementation of* EnumDeviceDrivers()

the heart of Listing 1-4, you can see the `NtQuerySystemInformation()` call that does the hard work. This is one of my favorite Windows 2000 functions, because it gives access to various kinds of important data structures, such as driver, process, thread, handle, and LPC port lists, plus many more. The internals of this powerful function and its friend `NtSetSystemInformation()` have been documented for the first time in my article "Inside Windows NT System Data," published in the November 1999 issue of *Dr. Dobb's Journal* (Schreiber 1999). Another comprehensive description of these functions can be looked up in Gary Nebbett's indispensable *Windows NT/2000 Native API Reference* (Nebbett 2000).

Don't worry too much about the various implementation details of the `EnumDeviceDrivers()` function in Listing 1-4. I have added this code snippet just to illustrate an interesting aspect of this function that runs like a red thread through `psapi.dll`. After obtaining the complete list of drivers in the second `NtQuerySystemInformation()` call by specifying the information class `SystemModuleInformation`, the code loops through the driver module array and copies all `pImageBase` members to the caller's pointer array named `lpImageBase[]`. This might seem OK, as long as you aren't aware of the other data contained in the module array supplied by `NtQuerySystemInformation()`. This data structure is undocumented, but I can tell you right now that it also specifies the sizes of the modules in memory, their paths and names, load counts, and some flags. Even the offset of the file name token inside the path is readily available! `EnumDeviceDrivers()` is mercilessly throwing away all of this valuable information, retaining nothing but the bare image base addresses.

This drama gets even weirder if you try to obtain more information about the modules referenced by the returned pointers. Guess what `psapi.dll` does if you are calling its API function `GetDeviceDriverFileName()` to obtain the image file path corresponding to an image base address. It runs through a code sequence similar to the one in Listing 1-4, again requesting the complete driver list, and again looping through its entries in search of the given address. If it finds a matching entry, it copies the path stored there to the caller's buffer. That's very efficient, isn't it? Why didn't `EnumDeviceDrivers()` copy the paths while it was scanning the driver list for the first time? It wouldn't have been very difficult to implement the function in this way. Besides the efficiency consideration, this design has another potential problem: What if the module in question has been unloaded right before the invocation of `GetDeviceDriverFileName()`? This entry would be missing from the second driver list, and `GetDeviceDriverFileName()` would fail. I don't understand why Microsoft has released a DLL that cripples the data returned by a powerful API function until it is almost useless.

ENUMERATING ACTIVE PROCESSES

Another typical task for `psapi.dll` is the enumeration of processes currently running in the system. To this end, the `EnumProcesses()` function is provided. It works quite similar to `EnumDeviceDrivers()`, but returns process IDs instead of virtual addresses. Again, there is no indication of the required buffer size if the output buffer is too small, so the usual trial-and-error loop must be used, as demonstrated in Listing 1-5. Actually, this code is nearly identical to Listing 1-3, except for slightly different symbol and type names.

A process ID is a global numeric tag that uniquely identifies a process within the entire system. Process and thread IDs are drawn from the same pool of numbers, starting at zero with the so-called Idle process. None of the running processes and threads have the same IDs at the same time. However, after a process terminates, it is possible that another process reuses some of the IDs previously assigned to the ceased

```
BOOL WINAPI EnumProcesses (DWORD *lpidProcess,
                           DWORD  cb,
                           DWORD *lpcbNeeded);

PDWORD WINAPI dbgProcessIds (PDWORD pdCount)
    {
    DWORD  dSize;
    DWORD  dCount = 0;
    PDWORD pdList = NULL;

    dSize = SIZE_MINIMUM * sizeof (DWORD);

    while ((pdList = dbgMemoryCreate (dSize)) != NULL)
        {
        if (EnumProcesses (pdList, dSize, &dCount) &&
            (dCount < dSize))
            {
            dCount /= sizeof (DWORD);
            break;
            }
        dCount = 0;
        pdList = dbgMemoryDestroy (pdList);
        if ((dSize <<= 1) > (SIZE_MAXIMUM * sizeof (DWORD))) break;
        }
    if (pdCount != NULL) *pdCount = dCount;
    return pdList;
    }
```

LISTING 1-5. *Enumerating Process IDs*

process and its threads. Therefore, a process ID obtained at time X might refer to a completely different process at time Y. It also might be undefined at the time it is used, or it might be assigned to a thread. Thus, a plain list of process IDs as returned by `EnumProcesses()` does not represent a faithful snapshot of the process activity in the system. This design flaw is even less pardonable if the implementation of this function is considered. Listing 1-6 is another `psapi.dll` function clone, outlining the basic actions taken by `EnumProcesses()`. Like `EnumDeviceDrivers()`, it relies on `NtQuerySystemInformation()`, but specifies the information class `SystemProcessInformation` instead of `SystemModuleInformation`. Please note the loop in the middle of Listing 1-6, where the `lpidProcess[]` array is filled with data from a `SYSTEM_PROCESS_INFORMATION` structure. It is not surprising that this structure is undocumented.

After having seen how wasteful `EnumDeviceDrivers()` is with the data it receives from `NtQuerySystemInformation()`, odds are that `EnumProcesses()` is of a similar kind. In fact, it is even worse! The available process information is much more exhaustive than the driver module information, because along with process data it also includes details about every thread in the system. While I am writing this text, my system runs 37 processes, and calling `NtQuerySystemInformation()` yields a data block of no less than 24,488 bytes! All that is left after `EnumProcesses()` has finished processing the data are 148 bytes, required for the 37 process IDs.

```
BOOL WINAPI EnumProcesses (PDWORD lpidProcess,
                           DWORD  cb,
                           PDWORD lpcbNeeded)
    {
    PSYSTEM_PROCESS_INFORMATION pspi, pspiNext;
    DWORD                       dSize, i;
    NTSTATUS                    ns;
    BOOL                        fOk = FALSE;

    for (dSize  = 0x8000;
         ((pspi = LocalAlloc (LMEM_FIXED, dSize)) != NULL);
         dSize += 0x8000)
        {
        ns = NtQuerySystemInformation (SystemProcessInformation,
                                       pspi, dSize, NULL);

        if (ns == STATUS_SUCCESS)
            {
            pspiNext = pspi;

            for (i = 0; i < cb / sizeof (DWORD); i++)
                {
```

```
                lpidProcess [i] = pspiNext->dUniqueProcessId;

                pspiNext = (PSYSTEM_PROCESS_INFORMATION)
                           ((PBYTE) pspiNext + pspiNext->dNext);
                }
            *lpcbNeeded = i * sizeof (DWORD);
            fOk         = TRUE;
            }
        LocalFree (pspi);

        if (fOk || (ns != STATUS_INFO_LENGTH_MISMATCH))
            {
            if (!fOk) SetLastError (RtlNtStatusToDosError (ns));
            break;
            }
        }
    return fOk;
    }
```

LISTING 1-6. *Sample Implementation of* EnumProcesses()

Although `EnumDeviceDrivers()` makes me somewhat sad, `EnumProcesses()` really breaks my heart. If you need justification for using undocumented API functions, these two functions are the best arguments. Why use less efficient functions such as these if the real thing is just one step away? Why not call `NtQuerySystemInformation()` yourself and get all that interesting system information for free? Many system administration utilities supplied by Microsoft rely on `NtQuerySystemInformation()` rather than `psapi.dll` functions, so why settle for less?

ENUMERATING PROCESS MODULES

Once you have found a process ID of interest in the process list returned by `EnumProcesses()`, you might want to know which modules are currently loaded into its virtual address space. `psapi.dll` provides yet another API function for this purpose, called `EnumProcessModules()`. Unlike `EnumDeviceDrivers()` and `EnumProcesses()`, this function requires four arguments (see top of Listing 1-7). Whereas these two functions return global system lists, `EnumProcessModules()` retrieves a process-specific list, so the process must be uniquely identified by an additional argument. However, instead of a process ID, this function requires a process `HANDLE`. To obtain a process handle given an ID, the `OpenProcess()` function must be called.

```
BOOL WINAPI EnumProcessModules (HANDLE   hProcess,
                                HMODULE *lphModule,
                                DWORD    cb,
                                DWORD   *lpcbNeeded);

PHMODULE WINAPI dbgProcessModules (HANDLE hProcess,
                                   PDWORD pdCount)
    {
    DWORD    dSize;
    DWORD    dCount = 0;
    PHMODULE phList = NULL;

    if (hProcess != NULL)
        {
        dSize = SIZE_MINIMUM * sizeof (HMODULE);

        while ((phList = dbgMemoryCreate (dSize)) != NULL)
            {
            if (EnumProcessModules (hProcess, phList, dSize,
                                    &dCount))
                {
                if (dCount <= dSize)
                    {
                    dCount /= sizeof (HMODULE);
                    break;
                    }
                }
            else
                {
                dCount = 0;
                }
            phList = dbgMemoryDestroy (phList);
            if (!(dSize = dCount)) break;
            }
        }
    if (pdCount != NULL) *pdCount = dCount;
    return phList;
    }
```

LISTING 1-7. *Enumerating Process Modules*

`EnumProcessModules()` returns references to the modules of a process by specifying their module handles. On Windows 2000, an `HMODULE` is simply the image base address of a module. In the Platform SDK header file `windef.h,` it is defined as an alias for `HINSTANCE,` which in turn is a `HANDLE` type. Microsoft has probably chosen this type assignment to point out that a module handle is an opaque quantity, and no assumptions should be made about its value. However, an `HMODULE` is not a handle in the strict sense. Usually, handles are indexes into a table managed by the system, where properties of objects are looked up. Each handle returned by the system increments an object-specific handle count, and an object instance cannot be removed from memory until all handles have been returned to the system. The Win32 API provides the `CloseHandle()` function for the latter purpose. Its equivalent in the context of the Native API is called `NtClose()`. The important thing about `HMODULE`s is that these "handles" need *not* be closed.

Another confusing thing is the fact that module handles are not generally guaranteed to remain valid. The remarks on the `GetModuleHandle()` function in the Platform SDK documentation state clearly that special care must be taken in multithreaded applications, because one thread might invalidate an `HMODULE` used by another thread by unloading the module to which this handle refers. The same is true in a multitasking environment in which an application (e.g., a debugger) wants to use a module handle of another application. This makes `HMODULE`s appear fairly useless, doesn't it? However, there are two situations in which an `HMODULE` remains valid long enough:

1. A `HMODULE` returned by `LoadLibrary()` or `LoadLibraryEx()` remains valid until the process calls `FreeLibrary()`, because these functions involve a module reference count. This prevents the module from being unloaded unexpectedly even in a multithreaded application design.
2. An `HMODULE` from a different process remains valid if it refers to a module that is permanently loaded. For example, all Windows 2000 kernel components (not including kernel-mode device drivers) are mapped to the same fixed addresses in each process and remain there for the lifetime of the process.

Unfortunately, neither of these situations applies to the module handles returned by the `psapi.dll` function `EnumProcessModules()`, at least not generally. The `HMODULE` values copied to the caller's buffer reflect the image base addresses that were in effect at the time the process snapshot was taken. A second later, the process might have called `FreeLibrary()` for one of the modules, removing it from memory and invalidating its handle. It is even possible that the process calls `LoadLibrary()` for a different DLL immediately afterward, and the new module is mapped to the address that has just been freed. If this looks familiar, you are right. This is the same problem encountered with the `EnumDeviceDrivers()` pointer array and the `EnumProcesses()` ID array. However, this problem is not inevitable. The undocumented API functions called by `psapi.dll` to collect the data work around these data integrity issues by returning a complete snapshot of the requested objects, including all properties of interest. It is not necessary to call other functions at a later time to obtain additional information. In my opinion, the design of `psapi.dll` is poor because of its ignorance of data integrity, which is why I would not use this DLL as a basis for a professional debugging application.

The `EnumProcessModules()` function is a better citizen than `EnumDeviceDrivers()` and `EnumProcesses()`, because it indicates exactly how many bytes are missing if the output data doesn't fit into the caller's array. Note that Listing 1-7 doesn't contain a loop where the buffer size is increased until it is large enough. However, a trial-and error loop is still required because the required size reported by `EnumProcessModules()` might be invalid at the next call if the process in question has loaded another module in the meantime. Therefore, the code in Listing 1-7 keeps on enumerating modules until `EnumProcessModules()` reports that the required buffer size is less than or equal to the available size or an error occurs.

I won't describe an equivalent implementation of `EnumProcessModules()`, because this function is slightly more complex than `EnumDeviceDrivers()` and `EnumProcesses()` and involves several undocumented data structures. Basically, it calls `NtQueryInformationProcess()` (it is undocumented, of course) to get the address of the target Process Environment Block (PEB), where it retrieves a pointer to a module information list. Because neither the PEB nor this list are "visible" in the caller's address space, `EnumProcessModules()` calls the Win32 API function `ReadProcessMemory()` (this one is documented) to take a peek at the target address space. By the way, the layout of the PEB structure is discussed later in Chapter 7, and also appears in the structure definition section of Appendix C.

ADJUSTING PROCESS PRIVILEGES

Recall the earlier discussion about the process handle required by `EnumProcess Modules()`. Usually, you will begin with a process ID—probably one of those returned by `EnumProcesses()`. The Win32 API provides the `OpenProcess()` function to get a handle to a process if its ID is known. This function expects an access flag mask as its first argument. Assuming that the process ID is stored in the `DWORD` variable `dId`, and you are calling `OpenProcess (PROCESS_ALL_ACCESS, FALSE, dId)` to obtain a handle with maximum access rights, you will get an error code for several processes with low ID numbers. This is not a bug—it is a security feature! These processes are system services that keep the system alive. A normal user process is not allowed to execute all possible operations on system services. For example, it is not a good idea to allow all processes to kill any other process in the system. If an application accidentally terminates a system service, the entire system crashes. Therefore, certain access rights can only be used by a process that has the appropriate privileges.

You can always bump up the privilege level of an application by claiming that it is a debugger. For obvious reasons, a debugger must have a large number of access rights to do its job. Changing the privileges of a process is essentially a straightforward sequence of three steps:

1. First, the so-called access token of the process must be opened, using the Win32 `advapi32.dll` function `OpenProcessToken()`.
2. If this call succeeds, the next step is to prepare a `TOKEN_PRIVILEGES` structure that contains information about the requested privilege. This task is facilitated by another `advapi32.dll` function named `LookupPrivilegeValue()`. The privilege is specified by name. The Platform SDK file `winnt.h` defines 27 privilege names and assigns symbols to them. For example, the debugging privilege has the symbol `SE_DEBUG_NAME`, which evaluates to the string `"SeDebugPrivilege"`.
3. If this call succeeds as well, `AdjustTokenPrivileges()` can be called with the token handle of the process and the initialized `TOKEN_PRIVILEGES` structure. Again, this function is exported by `advapi32.dll`.

Remember to close the token handle afterward if `OpenProcessToken()` succeeds. `w2k_dbg.dll` contains the API function `dbgPrivilegeSet()` that combines these steps, as shown in Listing 1-8. At the bottom of this listing, another `w2k_dbg.dll` function is included. `dbgPrivilegeDebug()` is a simple but convenient `dbgPrivilegeSet()` wrapper that specifically requests the debugging privilege. By the way, this trick is also employed by the wonderful `kill.exe` utility contained in Microsoft's Windows NT Server Resource Kit. `kill.exe` needs the debugging privilege to be able to kick starved services from memory. This is an indispensable tool for NT server administrators who want to restart a dead system service that doesn't respond to service control calls anymore, circumventing a full reboot. Anyone who runs Microsoft Internet Information Server (IIS) on the Web or in an intranet or extranet probably has this nifty tool in the emergency toolbox and issues a `kill inetinfo.exe` command every now and then.

```
BOOL WINAPI dbgPrivilegeSet (PWORD pwName)
    {
    HANDLE           hToken;
    TOKEN_PRIVILEGES tp;
    BOOL             fOk = FALSE;

    if ((pwName != NULL)
        &&
        OpenProcessToken (GetCurrentProcess (),
                          TOKEN_ADJUST_PRIVILEGES,
                          &hToken))
        {
        if (LookupPrivilegeValue (NULL, pwName,
                                  &tp.Privileges->Luid))
            {
            tp.Privileges->Attributes = SE_PRIVILEGE_ENABLED;
            tp.PrivilegeCount          = 1;
```

(continued)

```
            fOk = AdjustTokenPrivileges (hToken, FALSE, &tp,
                                         0, NULL, NULL)
                  &&
                  (GetLastError () == ERROR_SUCCESS);
            }
        CloseHandle (hToken);
        }
    return fOk;
    }

// -----------------------------------------------------------------

BOOL WINAPI dbgPrivilegeDebug (void)
    {
    return dbgPrivilegeSet (SE_DEBUG_NAME);
    }
```

LISTING 1-8. *Requesting a Privilege for a Process*

ENUMERATING SYMBOLS

After having bashed `psapi.dll` without mercy, it's time for a few more positive words. `psapi.dll` might be a flop, but on the other hand, `imagehlp.dll` is a true pearl! I came across this fine piece of software while searching for more information about the internal structure of Windows 2000 symbol files. Finally, a 3-year-old article of the world's best Windows surgeon Matt Pietrek (Pietrek 1997b) convinced me—at least for now—that it is absolutely unnecessary to know the layout of symbol files, because `imagehlp.dll` readily dissects them for me. This magic is done by its API function `SymEnumerateSymbols()`, whose prototype is shown in the upper half of Listing 1-9. Meanwhile, I have learned a lot about the most essential internals of the Windows NT 4.0 and Windows 2000 symbol files, so I no longer depend on `imagehlp.dll`. I will cover this information in the next section of this chapter.

The `hProcess` argument is usually a handle to the calling process, so it can be set to the result of `GetCurrentProcess()`. Note that `GetCurrentProcess()` doesn't return a real process handle. Instead, it returns a constant value of `0xFFFFFFFF` called a *pseudo handle*, which is accepted by all API functions that expect a process handle. `0xFFFFFFFE` is another pseudo handle that is interpreted as a handle to the current thread and is analogously returned by the API function `GetCurrentThread()`.

`BaseOfDll` is defined as a `DWORD`, although it is actually sort of a `HMODULE` or `HINSTANCE`. I guess Microsoft has chosen this data type to express that this value need not be a valid `HMODULE`, although it frequently is. `SymEnumerateSymbols()` calculates the base addresses of all enumerated symbols relative to this value. It is absolutely OK to query the symbols of a DLL that isn't currently loaded into any process address space, so `BaseOfDll` can be chosen arbitrarily.

```
BOOL IMAGEAPISymEnumerateSymbols
        (HANDLE      hProcess,
        DWORD        BaseOfDll,
        PSYM_ENUMSYMBOLS_CALLBACK Callback,
        PVOID        UserContext);

typedef BOOL (CALLBACK *PSYM_ENUMSYMBOLS_CALLBACK)
        (PTSTR       SymbolName,
        DWORD        SymbolAddress,
        DWORD        SymbolSize,
        PVOID        UserContext);
```

LISTING 1-9. SymEnumerateSymbols() *and its Callback Function*

The `Callback` argument is a pointer to a user-defined callback function that is invoked for every symbol. The lower half of Listing 1-9 provides information about its arguments. The callback function receives a zero-terminated symbol name string, the base address of the symbol with respect to the `BaseOfDll` argument of `SymEnumerateSymbols()` and the estimated size of the item tagged by the symbol. `SymbolName` is defined as a `PTSTR,` which means that its actual type depends on whether the ANSI or Unicode version of `SymEnumerateSymbols()` has been called. The Platform SDK documentation explicitly states that `SymbolSize` is a "best-guess value," and can be zero. I have found that `SymbolAddress` might be zero as well, and that `SymbolSize` can assume the two's complement of `SymbolAddress,` that is, adding both values yields zero. It is a good idea to filter out these special cases if you are only interested in symbols that refer to real code or data.

`UserContext` is an arbitrary pointer that can be used by the caller to keep track of the enumeration sequence. For example, it might point to a memory block where the symbol information has accumulated. This pointer is identical to the `UserContext` argument passed to the `Callback` function. The callback function can cancel the enumeration any time by returning the value `FALSE.` This action is typically taken when an unrecoverable error occurs or the caller has received the information for which it was waiting.

Listing 1-10 demonstrates a typical application of `SymEnumerateSymbols(),` again taken from the source code of `w2k_dbg.dll.` To enumerate the symbols of a specified module, the following steps have to be taken:

1. Before anything else, `SymInitialize()` must be called to initialize the symbol handler. Listing 1-11 shows the prototypes of this and other functions discussed here. The `hProcess` argument can be a handle to any active process in the system. Debuggers that maintain symbolic information for several processes use this parameter to identify the target process. Applications that simply wish to enumerate symbols offline may

```
PDBG_LIST WINAPI dbgSymbolList (PWORD pwPath,
                                PVOID pBase)
    {
    PLOADED_IMAGE pli;
    HANDLE        hProcess = GetCurrentProcess ();
    PDBG_LIST     pdl      = NULL;

    if ((pwPath != NULL) &&
        SymInitialize (hProcess, NULL, FALSE))
        {
        if ((pli = dbgSymbolLoad (pwPath, pBase, hProcess)) != NULL)
            {
            if ((pdl = dbgListCreate ()) != NULL)
                {
                SymEnumerateSymbols (hProcess, (DWORD_PTR) pBase,
                                     dbgSymbolCallback, &pdl);
                }
            dbgSymbolUnload (pli, pBase, hProcess);
            }
        SymCleanup (hProcess);
        }
    return dbgListFinish (pdl);
    }
```

LISTING 1-10. *Creating a Symbol List*

pass in the value of `GetCurrentProcess()`. The resources allocated by `SymInitialize()` must be freed later by calling `SymCleanup()`.

2. To obtain accurate information about the module for which symbols will be enumerated, it is advisable to call `ImageLoad()` now. Note that this function is specific to `imagehlp.dll`—it is not exported by the redistributable component `dbghelp.dll`. `ImageLoad()` returns a pointer to a `LOADED_IMAGE` structure containing very detailed information about the loaded module (see Listing 1-11). This structure must be deallocated later using `ImageUnload()`.
3. The last step before `SymEnumerateSymbols()` can be called is to load the symbol table of the target module by invoking `SymLoadModule()`. If `ImageLoad()` has been called before, the `hFile` and `SizeOfImage` members of the returned `LOADED_IMAGE` structure can be passed in as the respective arguments. Otherwise, you have to set `hFile` to NULL and `SizeOfImage` to zero. In this case, `SymLoadModule()` attempts to obtain the image size from the symbol file, which is not guaranteed to be accurate. The symbol table must be unloaded later by calling `SymUnloadModule()`.

```
BOOL IMAGEAPI SymInitialize (HANDLE hProcess,
                             PSTR   UserSearchPath,
                             BOOL   fInvadeProcess);

BOOL IMAGEAPI SymCleanup (HANDLE hProcess);

DWORD IMAGEAPI SymLoadModule (HANDLE hProcess,
                              HANDLE hFile,
                              PSTR   ImageName,
                              PSTR   ModuleName,
                              DWORD  BaseOfDll,
                              DWORD  SizeOfDll);

BOOL IMAGEAPI SymUnloadModule (HANDLE hProcess,
                               DWORD  BaseOfDll);

PLOADED_IMAGE IMAGEAPI ImageLoad (PSTR DllName,
                                  PSTR DllPath);

BOOL IMAGEAPI ImageUnload (PLOADED_IMAGE LoadedImage);

typedef struct _LOADED_IMAGE
    {
    PSTR                  ModuleName;
    HANDLE                hFile;
    PUCHAR                MappedAddress;
    PIMAGE_NT_HEADERS     FileHeader;
    PIMAGE_SECTION_HEADER LastRvaSection;
    ULONG                 NumberOfSections;
    PIMAGE_SECTION_HEADER Sections;
    ULONG                 Characteristics;
    BOOLEAN               fSystemImage;
    BOOLEAN               fDOSImage;
    LIST_ENTRY            Links;
    ULONG                 SizeOfImage;
    }
    LOADED_IMAGE, *PLOADED_IMAGE;
```

LISTING 1-11. *Various* `imagehlp.dll` *API Prototypes*

In Listing 1-10, the `SymInitialize()`, `SymEnumerateSymbols()`, and `SymCleanup()` calls are clearly discernible. Please ignore the `dbgListCreate()` and `dbgListFinish()` calls—they refer to `w2k_dbg.dll` API functions that help build object lists in memory. The other `imagehlp.dll` function references mentioned above are hidden inside the `w2k_dbg.dll` API functions `dbgSymbolLoad()` and `dbgSymbolUnload()`, shown in Listing 1-12. Note that `dbgSymbolLoad()` uses `dbgStringAnsi()` to convert the module path string from Unicode to ANSI, because `imagehlp.dll` doesn't export a Unicode variant of `ImageLoad()`.

```
PLOADED_IMAGE WINAPI dbgSymbolLoad (PWORD  pwPath,
                                    PVOID  pBase,
                                    HANDLE hProcess)
    {
    WORD          awPath [MAX_PATH];
    PBYTE         pbPath;
    DWORD         dPath;
    PLOADED_IMAGE pli = NULL;

    if ((pbPath = dbgStringAnsi (pwPath, NULL)) != NULL)
        {
        if (((pli = ImageLoad (pbPath, NULL)) == NULL)         &&
            (dPath = dbgPathDriver (pwPath, awPath, MAX_PATH)) &&
            (dPath < MAX_PATH))
            {
            dbgMemoryDestroy (pbPath);

            if ((pbPath = dbgStringAnsi (awPath, NULL)) != NULL)
                {
                pli = ImageLoad (pbPath, NULL);
                }
            }
        if ((pli != NULL)
            &&
            (!SymLoadModule (hProcess, pli->hFile, pbPath, NULL,
                             (DWORD_PTR) pBase, pli->SizeOfImage)))
            {
            ImageUnload (pli);
            pli = NULL;
            }
        dbgMemoryDestroy (pbPath);
        }
    return pli;
    }

// ----------------------------------------------------------------

PLOADED_IMAGE WINAPI dbgSymbolUnload (PLOADED_IMAGE pli,
                                      PVOID         pBase,
                                      HANDLE        hProcess)
    {
    if (pli != NULL)
        {
        SymUnloadModule (hProcess, (DWORD_PTR) pBase);
        ImageUnload     (pli);
        }
    return NULL;
    }

// ----------------------------------------------------------------
```

```
PDBG_LIST WINAPI dbgSymbolList (PWORD pwPath,
                                PVOID pBase)
    {
    PLOADED_IMAGE pli;
    HANDLE        hProcess = GetCurrentProcess ();
    PDBG_LIST     pdl      = NULL;

    if ((pwPath != NULL) &&
        SymInitialize (hProcess, NULL, FALSE))
        {
        if ((pli = dbgSymbolLoad (pwPath, pBase, hProcess)) != NULL)
            {
            if ((pdl = dbgListCreate ()) != NULL)
                {
                SymEnumerateSymbols (hProcess, (DWORD_PTR) pBase,
                                     dbgSymbolCallback, &pdl);
                }
            dbgSymbolUnload (pli, pBase, hProcess);
            }
        SymCleanup (hProcess);
        }
    return dbgListFinish (pdl);
    }
```

LISTING 1-12. *Loading and Unloading Symbol Information*

ImageLoad() does a very good job locating the specified module, even if only its name is given, without any path information. However, it fails on kernel-mode drivers residing in the \winnt\system32\drivers directory, because it is usually not part of the system's search path list. In this case, dbgSymbolLoad() asks the dbgPathDriver() function for help and retries the LoadImage() call. dbgPathDriver() simply prefixes the specified path with the string "driver\" if the path consists of a bare file name only. If either of the ImageLoad() calls returns a valid LOADED_IMAGE pointer, dbgSymbolLoad() fulfills its mission by loading the module's symbol table via SymLoadModule() and returns the LOADED_IMAGE structure if successful. Its counterpart dbgSymbolUnload() is almost trivial—it unloads the symbol table and then destroys the LOADED_IMAGE structure.

In Listing 1-10, SymEnumerateSymbols() is instructed to use the w2k_dbg.dll function dbgSymbolCallback() for the callbacks. I am not including its source code here because it isn't relevant to imagehlp.dll. It just uses the symbol information it receives (see the definition of PSYM_ENUMSYMBOLS_CALLBACK in Listing 1-9) and adds it to a memory block passed in as its UserContext pointer. Although the list, index, and sorting functions featured by w2k_dbg.dll are interesting in their own right, they are beyond the scope of this book. Please consult the source files of w2k_dbg.dll and w2k_sym.exe on the CD if you need more information.

A Windows 2000 Symbol Browser

`w2k_sym.exe` is a sample client application of `w2k_dbg.dll` running in Win32 console mode. If you invoke it without arguments, it identifies itself as the Windows 2000 Symbol Browser and displays the help screen shown in Example 1-5. The program recognizes several command line switches that determine the actions it should take. The four basic options are `/p` (list processes), `/m` (list process modules), `/d` (list drivers and system modules), or the path of a module for which symbol information is requested. The default behavior can be altered by adding various display mode, sorting, and filtering switches. For example, if you want to see a list of all `ntoskrnl.exe`

```
// w2k_sym.exe
// SBS Windows 2000 Symbol Browser V1.00
// 08-27-2000 Sven B. Schreiber
// sbs@orgon.com

Usage: w2k_sym { <mode> [ /f | /F <filter> ] <operation> }

<mode> is a series of options for the next <operation>:

       /a : sort by address
       /s : sort by size
       /i : sort by ID (process/module lists only)
       /n : sort by name
       /c : sort by name (case-sensitive)
       /r : reverse order
       /l : load  checkpoint file (see below)
       /w : write checkpoint file (see below)
       /e : display end address instead of size
       /v : verbose mode

/f <filter> applies a case-insensitive search pattern.
/F <filter> works analogous, but case-sensitive.
In <filter>, the wildcards * and ? are allowed.

<operation> is one of the following:

       /p : display processes      - checkpoint: processes.dbgl
       /m : display modules        - checkpoint: modules.dbgl
       /d : display drivers        - checkpoint: drivers.dbgl
    <file> : display <file> symbols - checkpoint: symbols.dbgl

<file> is a file name, a relative path, or a fully qualified path.
Checkpoint files are loaded from and written to the current directory.
A checkpoint is an on-disk image of a DBG_LIST structure (see w2k_dbg.h).
```

Example 1-5. *The Command Help of* `w2k_sym.exe` symbols sorted by name, issue the command `w2k_sym /n/v ntoskrnl.exe`. The `/n` switch selects sort-by-name mode, and `/v` tells the program to be verbose, displaying the complete symbol list—otherwise, only summary information would be visible.

As an additional option, `w2k_sym.exe` allows reading and writing checkpoint files. A checkpoint is simply a one-to-one copy of an object list written to a disk file. You can use checkpoints to save the state of your system for later comparison. A checkpoint file contains a CRC32 field that is used to validate the contents of the file when it is loaded. `w2k_sym.exe` maintains four checkpoints in the current directory, corresponding to the four basic program options mentioned earlier, that is, process, module, driver, and symbol lists.

MICROSOFT SYMBOL FILE INTERNALS

It is great that Microsoft provides a standard interface to access the Windows 2000 symbol files, no matter what internal format they are using. Sometimes, however, you may wish to have direct access to their internals, just to gain more control of the data. This section shows you how the data in symbol files of type `.dbg` and `.pdb` are structured, and presents a DLL with a sample client application that allows you to look up and browse symbolic information buried inside them. Yes, this is going to be another symbol browser application, but don't worry—I won't bore you with a simple rehash of familiar code. The alternative symbol browser is quite different from the one discussed in the previous section.

SYMBOL DECORATION

Microsoft symbol files store the names of symbols in their so-called decorated form, which means that the symbol name might be prefixed and postfixed by additional character sequences that carry information about the type and usage of the symbol. Table 1-4 lists the most common forms of decorations. Symbols generated by C code usually have a leading underscore or `@` character, depending on the calling convention. An `@` character indicates a `__fastcall` function, and an underscore indicates `__stdcall` and `__cdecl` functions. Because the `__fastcall` and `__stdcall` conventions leave the task of cleaning up the argument stack to the called function, the symbols assigned to functions of this type also include the number of argument bytes put on the stack by the caller. This information is appended to the symbol name in decimal notation, separated by an `@` character. In this scenario, global variables are treated like `__cdecl` functions—that is, their symbols start with an underscore and have no trailing argument stack information.

TABLE 1-4. *Symbol Decoration Categories*

EXAMPLE	DESCRIPTION
symbol	Undecorated symbol (might have been declared in an ASM module)
_*symbol*	__*cdecl* function or global variable
_*symbol*@N	__*stdcall* function with *N* argument bytes
@*symbol*@N	__*fastcall* function with *N* argument bytes
__imp__*symbol*	import thunk of a __*cdecl* function or variable
__imp__*symbol*@N	import thunk of a __*stdcall* function with *N* argument bytes
__imp_@*symbol*@N	import thunk of a __*fastcall* function with *N* argument bytes
?*symbol*	C++ symbol with embedded argument type information
___@@_PchSym_*symbol*	PCH symbol

Some symbol names have a prefix of `__imp__` or `__imp_@`. These symbols are assigned to import thunks, which are pointers to functions or variables in other modules. Import thunks facilitate dynamic linking to symbols exported by other components at runtime, regardless of the actual load address of the target module. When a module is loaded, the loader mechanism fixes up the thunk pointers to refer to the actual entry point addresses. The benefit of import thunks is that the fixup for each imported function or variable has to be done only once per symbol—all references to this external symbol are routed through its thunk. It should be noted that import thunks are not a requirement. It is up to the compiler to decide whether it wants to minimize fixups by adding thunks or minimize memory usage by saving the space required for the thunks. As Table 1-4 shows, the same prefix/postfix rules apply to local and imported symbols, except that import thunks have an additional `__imp_` prefix (with two leading underscores!).

The undecoration problems of `imagehlp.dll` can easily be demonstrated with the help of the `w2k_sym.exe` sample application from the previous section, because it ultimately relies on the `imagehlp.dll` API via the `w2k_dbg.dll` library. If you issue the command `w2k_sym /v/n/f __* ntoskrnl.exe`, instructing `w2k_sym.exe` to display a sorted list of names starting with two underscore characters, you will see something that should look like the list in Example 1-6. What's strange is the pile of `__` symbols at the top of the table. Entering a command such as `ln 8047F798` in the Kernel Debugger yields the result `ntoskrnl!__`, which isn't any better. The original decorated name of the symbol at address `0x8047F798` is actually `___@@_PchSym_@00@UmgUkirezgvUmglhUlyUfkUlyqUrDIGUlykOlyq@ob`, so it seems that `imagehlp.dll` simply has stripped all characters except for two of the three leading underscores.

```
      #  ADDRESS      SIZE NAME
   ----------------------------------------------------------------
   6870: 8047F798       4 __
   6871: 80480B8C      14 __
   6872: 8047E724       4 __
   6873: 80471FE0       4 __
   6874: 804733B8      28 __
   6875: 804721D0      20 __
   6876: 804759A4       4 __
   6877: 80480004      1C __
   6878: 8047DA8C      14 __
   6879: 8047238C       4 __
   6880: 8047E6D4       4 __
   6881: 804755D4       4 __
   6882: 80471700       4 __decimal_point
   6883: 80471704       4 __decimal_point_length
   6884: 80471FC0       8 __fastflag
  ...
```

Example 1-6. *Results of the Command* `w2k_sym /v/n/f __* ntoskrnl.exe`

An even better example is the command `w2k_sym /v/n/f _imp_* ntoskrnl.exe` that displays all symbols starting with the character sequence `_imp_`. The resulting list, excerpted in Example 1-7, comprises the import thunks of `ntoskrnl.exe`. Again, the list starts with a long sequence of ambiguous names, and again the Kernel Debugger isn't helpful, because it reports the same names for these addresses. If I tell you now that the original name of the symbol at address `0x804005A4` is `__imp_@ExReleaseFastMutex@4`, what do you think? Obviously, one leading underscore has gotten lost, and the entire tail string starting at the first `@` character is missing. It seems that the undecoration algorithm inside `imagehlp.dll` has a problem with `@` characters. The reason for this strange behavior is that `@` is not only the prefix of `__fastcall` function names but also the separator for the argument stack size trailer of `__fastcall` and `__stdcall` functions. Obviously, the applied undecoration algorithm is satisfied to find a leading underscore and an `@` character, erroneously assuming that the remaining trailer specifies the number of bytes on the caller's argument stack. Therefore, the lengthy PCH symbols are stripped down to two underscores, and the `__fastcall` import thunks are reduced to `_imp_`. In both cases, the first leading underscore is removed and the first `@` plus all characters following it are discarded as well.

```
    #  ADDRESS      SIZE NAME
  ------------------------------------------------------------------
  6761: 804005A4       4 _imp_
  6762: 80400584       4 _imp_
  6763: 80400594       4 _imp_
  6764: 80400524       4 _imp_
  6765: 8040059C       4 _imp_
  6766: 80400534       4 _imp_
  6767: 80400590       4 _imp_
  6768: 804004EC       4 _imp_
  6769: 80400554       4 _imp_
  6770: 80400598       4 _imp_
  6771: 80400520       4 _imp__HalAllocateAdapterChannel
  6772: 804004C0       4 _imp__HalAllocateCommonBuffer
  6773: 804004E8       4 _imp__HalAllProcessorsStarted
...
```

EXAMPLE 1-7. *Results of the Command* `w2k_sym /v/n/f _imp_* ntoskrnl.exe`

The above examples are two potential reasons why you might lose patience and say: “Hey, I’m going to do it my own way!” The problem is that the internals of the Microsoft symbol file format are only scarcely documented, and some parts of the symbolic information—most notably the structure of Program Database (PDB) files—are completely undocumented. The Microsoft Knowledge Base even contains an article that clearly states:

“The Program Database File Format also known as PDB file format is not documented. This information is Microsoft proprietary.” (Microsoft 2000d.)

This sounds as if any attempts to roll your own symbol information parser must fail. However, you can bet that I’d never dare to add a section to this book that would end with the words “... but unfortunately, I can’t tell you more because the internals of PDB files are unknown to me.” Of course, I will tell you how PDB files are structured. But first, we will have to examine to the internals of `.dbg` files, because this is where the entire story starts.

THE INTERNAL STRUCTURE OF .dbg FILES

The symbolic information of the Windows NT 4.0 components is packed into files whose names end with a `.dbg` extension. The file names and the subdirectories hosting these files can be immediately derived from the component file name. Assuming that the symbol root directory of a system is `d:\winnt\symbols,` the full path of the symbol file of the component `filename.ext` is `d:\winnt\symbols\ext\filename.dbg`. For example, the kernel symbols can be found in the file `d:\winnt\symbols\exe\ntoskrnl.dbg`. Windows 2000 comes with `.dbg` files, too. However, the symbolic information has been moved to separate `.pdb` files. Therefore, each Windows 2000 component has an associated `ext\filename.dbg` and an additional `ext\filename.pdb` file in the symbol root directory. Aside from this difference, the contents of the Windows NT 4.0 and 2000 `.dbg` files are quite similar.

Fortunately, the internals of `.dbg` files are at least partially documented. The Win32 Platform SDK header file `winnt.h` provides important constant and type definitions of the core parts, and the Microsoft Developer Network (MSDN) Library contains some very helpful articles about this file format. Certainly the most enlightening article is Matt Pietrek's March 1999 edition of his "Under the Hood" column in *Microsoft Systems Journal (MSJ),* renamed *MSDN Magazine* (Pietrek 1999). Basically, a `.dbg` file consists of a header and a data section. Both sections have variable size and are further subdivided. The header part comprises four major subsections:

1. An `IMAGE_SEPARATE_DEBUG_HEADER` structure, starting with the two-letter signature "DI" (top section of Listing 1-13).
2. An array of `IMAGE_SECTION_HEADER` structures, one for each section in the component's PE file (middle section of Listing 1-13). The number of entries is specified by the `NumberOfSections` member of the `IMAGE_SEPARATE_DEBUG_HEADER`.
3. A sequence of zero-terminated 8-bit ANSI strings, comprising all exported symbols in undecorated form. The size of this subsection is specified by the `ExportedNamesSize` member of the `IMAGE_SEPARATE_DEBUG_HEADER`. If the module doesn't export any symbols, the `ExportedNamesSize` is zero, and the subsection is not present.
4. An array of `IMAGE_DEBUG_DIRECTORY` structures, describing the locations and formats of the subsequent data in the file (bottom section of Listing 1-13). The size of this subsection is specified by the `DebugDirectorySize` member of the `IMAGE_SEPARATE_DEBUG_HEADER`.

```
#define IMAGE_SEPARATE_DEBUG_SIGNATURE 0x4944 // "DI"
typedef struct _IMAGE_SEPARATE_DEBUG_HEADER
    {
    WORD  Signature;
    WORD  Flags;
    WORD  Machine;
    WORD  Characteristics;
    DWORD TimeDateStamp;
    DWORD CheckSum;
    DWORD ImageBase;
    DWORD SizeOfImage;
    DWORD NumberOfSections;
    DWORD ExportedNamesSize;
    DWORD DebugDirectorySize;
    DWORD SectionAlignment;
    DWORD Reserved[2];
    }
    IMAGE_SEPARATE_DEBUG_HEADER, *PIMAGE_SEPARATE_DEBUG_HEADER;

// -----------------------------------------------------------------

#define IMAGE_SIZEOF_SHORT_NAME 8

typedef struct _IMAGE_SECTION_HEADER
    {
    BYTE  Name[IMAGE_SIZEOF_SHORT_NAME];
    union
        {
        DWORD PhysicalAddress;
        DWORD VirtualSize;
        } Misc;
    DWORD VirtualAddress;
    DWORD SizeOfRawData;
    DWORD PointerToRawData;
    DWORD PointerToRelocations;
    DWORD PointerToLinenumbers;
    WORD  NumberOfRelocations;
    WORD  NumberOfLinenumbers;
    DWORD Characteristics;
    }
    IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;

// -----------------------------------------------------------------

#define IMAGE_DEBUG_TYPE_UNKNOWN        0
#define IMAGE_DEBUG_TYPE_COFF           1
#define IMAGE_DEBUG_TYPE_CODEVIEW       2
#define IMAGE_DEBUG_TYPE_FPO            3
#define IMAGE_DEBUG_TYPE_MISC           4
#define IMAGE_DEBUG_TYPE_EXCEPTION      5
#define IMAGE_DEBUG_TYPE_FIXUP          6
```

```
#define IMAGE_DEBUG_TYPE_OMAP_TO_SRC    7
#define IMAGE_DEBUG_TYPE_OMAP_FROM_SRC  8
#define IMAGE_DEBUG_TYPE_BORLAND        9
#define IMAGE_DEBUG_TYPE_RESERVED10    10
#define IMAGE_DEBUG_TYPE_CLSID         11

typedef struct _IMAGE_DEBUG_DIRECTORY
    {
    DWORD Characteristics;
    DWORD TimeDateStamp;
    WORD  MajorVersion;
    WORD  MinorVersion;
    DWORD Type;
    DWORD SizeOfData;
    DWORD AddressOfRawData;
    DWORD PointerToRawData;
    }
    IMAGE_DEBUG_DIRECTORY, *PIMAGE_DEBUG_DIRECTORY;
```

LISTING 1-13. *Header Structures of a* `.dbg` *File*

Because of the variable size of the header subsections, their absolute positions within the `.dbg` file must be computed from the size of the preceding subsections, respectively. A `.dbg` file parser usually applies the following algorithm:

- The `IMAGE_SEPARATE_DEBUG_HEADER` is always located at the beginning of the file.
- The first `IMAGE_SECTION_HEADER` immediately follows the `IMAGE_SEPARATE_DEBUG_HEADER`, so it is always found at file offset `0x30`.
- The offset of the first exported name is determined by multiplying the size of the `IMAGE_SECTION_HEADER` structure by the number of sections and adding it to the offset of the first section header. Thus, the first string is located at offset `0x30 + (NumberOfSections * 0x28)`.
- The location of the first `IMAGE_DEBUG_DIRECTORY` entry is determined by adding the `ExportedNamesSize` to the offset of the exported-names subsection.
- The offsets of the remaining data items in the `.dbg` file are determined by the `IMAGE_DEBUG_DIRECTORY` entries. The offsets and sizes of the associated data blocks are specified by the `PointerToRawData` and `SizeOfData` members, respectively.

The IMAGE_DEBUG_TYPE_* definitions in Listing 1-13 reflect the various data formats a .dbg file can comprise. However, the Windows NT 4.0 symbol files typically contain only four of them: IMAGE_DEBUG_TYPE_COFF, IMAGE_DEBUG_TYPE_CODEVIEW, IMAGE_DEBUG_TYPE_FPO, and IMAGE_DEBUG_TYPE_MISC. The Windows 2000 .dbg files usually add IMAGE_DEBUG_TYPE_OMAP_TO_SRC, IMAGE_DEBUG_TYPE_OMAP_FROM _SRC, and an undocumented type with ID 0x1000 to this list. If you are interested only in resolving or browsing symbols, the only required directory entries are IMAGE_ DEBUG_TYPE_CODEVIEW, IMAGE_DEBUG_TYPE_OMAP_TO_SRC, and IMAGE_DEBUG_ TYPE_OMAP_FROM_SRC.

The companion CD of this book contains a sample DLL named w2k_img.dll that parses .dbg and .pdb files and exports several interesting functions for developers of debugging tools. The source code of this DLL is found in the \src\w2k_img tree of the CD. One important property of w2k_img.dll is that it is designed to run on *all* Win32 platforms. This not only includes Windows 2000 and Windows NT 4.0 but also Windows 95 and 98. Like all good citizens in the Win32 world, this DLL comes with separate entry points for ANSI and Unicode strings. By default, a client application uses the ANSI functions. If the application includes the line #define UNICODE in its source code, the Unicode entry points are selected transparently. Client applications that run on Win32 platforms should use ANSI exclusively. Applications specific to Windows 2000/NT can switch to Unicode for better performance.

The sample CD also contains an example application called SBS Windows 2000 CodeView Decompiler, whose Microsoft Visual C/C++ project files are found in the \src\w2k_cv tree. It is a very simple application that dissects .dbg and .pdb files and dumps the contents of their sections to a console window. You can use it while reading this section to see live examples of the data structures discussed here. w2k_cv.exe makes heavy use of several w2k_img.dll API functions.

Listing 1-14 shows one of the basic data structures defined in w2k_img.h. The IMG_DBG structure is essentially a concatenation of the first two .dbg file header sections, that is, the fixed-size basic header and the array of PE section headers. The actual size of the structure, given the number of sections, is computed by the macro IMG_DBG__(). Its result specifies the file offset of the exported-names subsection.

Several w2k_img.dll API functions expect a pointer to an initialized IMG_DBG structure. The imgDbgLoad() function (not reprinted here) allocates and returns a properly initialized IMG_DBG structure containing the data of the specified .dbg file. imgDbgLoad() performs very strict sanity checks on the data to verify that the file is valid and complete. The returned IMG_DBG structure can be passed to several parsing functions that return the linear addresses of the most frequently used .dbg file components. For example, the imgDbgExports() function in Listing 1-15 computes the linear address of the sequence of exported names following the IMAGE_SECTION_HEADER array. It also counts the number of available names by scanning the string sequence up to the end of the subsection and optionally writes this value to the variable pointed to by the pdCount argument.

```
typedef struct _IMG_DBG
    {
    IMAGE_SEPARATE_DEBUG_HEADER Header;
    IMAGE_SECTION_HEADER        aSections [];
    }
    IMG_DBG, *PIMG_DBG, **PPIMG_DBG;

#define IMG_DBG_ sizeof (IMG_DBG)
#define IMG_DBG__(_n) (IMG_DBG_ + ((_n) * IMAGE_SECTION_HEADER_))

#define IMG_DBG_DATA(_p,_d) \
        ((PVOID) ((PBYTE) (_p) + (_d)->PointerToRawData))
```

LISTING 1-14. *The* `IMG_DBG` *Structure and Related Macros*

```
PBYTE WINAPI imgDbgExports (PIMG_DBG pid,
                            PDWORD   pdCount)
    {
    DWORD i, j;
    DWORD dCount    = 0;
    PBYTE pbExports = NULL;

    if (pid != NULL)
        {
        pbExports = (PBYTE) pid->aSections
                    + (pid->Header.NumberOfSections
                       * IMAGE_SECTION_HEADER_);

        for (i = 0; i < pid->Header.ExportedNamesSize; i = j)
            {
            if (!pbExports [j = i]) break;

            while ((j < pid->Header.ExportedNamesSize) &&
                   pbExports [j++]);

            if ((j > i) && (!pbExports [j-1])) dCount++;
            }
        }
    if (pdCount != NULL) *pdCount = dCount;
    return pbExports;
    }
```

LISTING 1-15. *The* `imgDbgExports()` *API Function*

Listing 1-16 defines two more API functions that locate debug directory entries by their `IMAGE_DEBUG_TYPE_*` IDs. `imgDbgDirectories()` returns the base address of the `IMAGE_DEBUG_DIRECTORY` array, whereas `imgDbgDirectory()` returns a pointer to the first directory entry with the specified type ID or returns `NULL` if no such entry exists.

```
PIMAGE_DEBUG_DIRECTORY WINAPI imgDbgDirectories (PIMG_DBG pid,
                                                 PDWORD   pdCount)
    {
    DWORD                  dCount = 0;
    PIMAGE_DEBUG_DIRECTORY pidd   = NULL;

    if (pid != NULL)
        {
        pidd   = (PIMAGE_DEBUG_DIRECTORY)
                 ((PBYTE) pid
                  + IMG_DBG__ (pid->Header.NumberOfSections)
                  + pid->Header.ExportedNamesSize);

        dCount = pid->Header.DebugDirectorySize
                 / IMAGE_DEBUG_DIRECTORY_;
        }
    if (pdCount != NULL) *pdCount = dCount;
    return pidd;
    }

// -----------------------------------------------------------------

PIMAGE_DEBUG_DIRECTORY WINAPI imgDbgDirectory (PIMG_DBG pid,
                                               DWORD    dType)
    {
    DWORD                  dCount, i;
    PIMAGE_DEBUG_DIRECTORY pidd = NULL;

    if ((pidd = imgDbgDirectories (pid, &dCount)) != NULL)
        {
        for (i = 0; i < dCount; i++, pidd++)
            {
            if (pidd->Type == dType) break;
            }
        if (i == dCount) pidd = NULL;
        }
    return pidd;
    }
```

LISTING 1-16. *The* `imgDbgDirectories()` *and* `imgDbgDirectory()` *API Functions*

The `imgDbgDirectory()` function can be used to look up the CodeView data in the `.dbg` file. This is done by the `imgDbgCv()` function in Listing 1-17. It calls `imgDbgDirectory()` with the `IMAGE_DEBUG_TYPE_CODEVIEW` type ID, and invokes the `IMG_DBG_DATA()` macro shown in Listing 1-14 to convert the data offset supplied by the `IMAGE_DEBUG_DIRECTORY` entry to an absolute linear address. This macro simply adds the offset to the base address of the `IMG_DBG` structure and typecasts it to a `PVOID` pointer. `imgDbgCv()` copies the size of the CodeView subsection to `*pdSize` if the `pdSize` argument is not `NULL`. The internals of the CodeView data are discussed below.

The API functions for the other data subsections look quite similar. Listing 1-18 shows the `imgDbgOmapToSrc()` and `imgDbgOmapFromSrc()` functions along with the `OMAP_TO_SRC` and `OMAP_FROM_SRC` structures on which they operate. Later, we will need these structures to compute the linear addresses of a symbol from its CodeView data. Because the OMAP data are an array of fixed-length structures, both API functions don't return the plain subsection size, but compute the number of entries in the array by simply dividing the overall size by the size of an entry. The result is copied to `*pdCount` if the `pdCount` argument is not `NULL`.

```
PCV_DATA WINAPI imgDbgCv (PIMG_DBG pid,
                          PDWORD   pdSize)
    {
    PIMAGE_DEBUG_DIRECTORY pidd;
    DWORD                  dSize = 0;
    PCV_DATA               pcd   = NULL;

    if ((pidd = imgDbgDirectory (pid, IMAGE_DEBUG_TYPE_CODEVIEW))
        != NULL)
        {
        pcd   = IMG_DBG_DATA (pid, pidd);
        dSize = pidd->SizeOfData;
        }
    if (pdSize != NULL) *pdSize = dSize;
    return pcd;
    }
```

LISTING 1-17. *The* `imgDbgCv()` *API Function*

```
typedef struct _OMAP_TO_SRC
    {
    DWORD dTarget;
    DWORD dSource;
    }
    OMAP_TO_SRC, *POMAP_TO_SRC, **PPOMAP_TO_SRC;

#define OMAP_TO_SRC_ sizeof (OMAP_TO_SRC)
// -----------------------------------------------------------------
```

(continued)

```
// ----------------------------------------------------------------

typedef struct _OMAP_FROM_SRC
    {
    DWORD dSource;
    DWORD dTarget;
    }
    OMAP_FROM_SRC, *POMAP_FROM_SRC, **PPOMAP_FROM_SRC;

#define OMAP_FROM_SRC_ sizeof (OMAP_FROM_SRC)

// ----------------------------------------------------------------

POMAP_TO_SRC WINAPI imgDbgOmapToSrc (PIMG_DBG pid,
                                     PDWORD   pdCount)
    {
    PIMAGE_DEBUG_DIRECTORY pidd;
    DWORD                  dCount = 0;
    POMAP_TO_SRC           pots   = NULL;

    if ((pidd = imgDbgDirectory (pid,
                                 IMAGE_DEBUG_TYPE_OMAP_TO_SRC))
        != NULL)
        {
        pots   = IMG_DBG_DATA (pid, pidd);
        dCount = pidd->SizeOfData / OMAP_TO_SRC_;
        }
    if (pdCount != NULL) *pdCount = dCount;
    return pots;
    }

// ----------------------------------------------------------------

POMAP_FROM_SRC WINAPI imgDbgOmapFromSrc (PIMG_DBG pid,
                                         PDWORD   pdCount)
    {
    PIMAGE_DEBUG_DIRECTORY pidd;
    DWORD                  dCount = 0;
    POMAP_FROM_SRC         pofs   = NULL;

    if ((pidd = imgDbgDirectory (pid,
                                 IMAGE_DEBUG_TYPE_OMAP_FROM_SRC))
        != NULL)
        {
        pofs   = IMG_DBG_DATA (pid, pidd);
        dCount = pidd->SizeOfData / OMAP_FROM_SRC_;
        }
    if (pdCount != NULL) *pdCount = dCount;
    return pofs;
    }
```

LISTING 1-18. *The* `imgDbgOmapToSrc()` *and* imgDbgOmapFromSrc() *API Function*

CodeView Subsections

CodeView is Microsoft's own debugging information format. It has undergone various metamorphoses through the years of the evolution of the Microsoft C/C++ compiler and linker. The internals of some CodeView versions differ radically from each other. However, all CodeView versions share a 32-bit signature at the beginning of the data that uniquely identifies the data format. The Windows NT 4.0 symbol files use the `NB09` format, which has been introduced by CodeView 4.10. The Windows 2000 files contain `NB10` CodeView data, which is merely a referral to a separate `.pdb` file, as I will demonstrate later.

`NB09` CodeView data is subdivided into a directory and subordinate entries. As Matt Pietrek points out in his *MSJ* article about `.dbg` files, most of the basic CodeView structures are defined in a set of sample header files coming with the Platform SDK. If you have installed the SDK samples, you will find a group of highly interesting files in the directory `\Program Files\Microsoft Platform SDK\Samples\SdkTools\Image\Include`. The files you need for CodeView parsing are named `cvexefmt.h` and `cvinfo.h`. Unfortunately, these files haven't been updated for a long time, as their file date 09-07-1994 indicates. It is striking that all structure names defined in `cvexefmt.h` start with the letters OMF, which is the acronym for Object Module Format. OMF is the standard file format used by 16-bit DOS and Windows `.obj` and `.lib` files. Starting with the Win32 versions of Microsoft's development tools, this format has been superseded by the Common Object File Format (COFF, see Gircys 1988 for details).

Although the original OMF format is obsolete today, it must be acknowledged that it was a clever file format. One of its objectives is to waste as little memory and disk space as possible. Another important property is that this format can be successfully parsed by applications even if they do not fully understand all parts of the file. The basic OMF data structure is the tagged record, starting with a tag byte identifying the type of data contained in the record, and a 16-bit length word specifying the number of subsequent bytes. This design makes it possible for an OMF reader to skip from record to record, picking out the record types in which it is interested. Microsoft has adopted this paradigm for its CodeView format, which explains the OMF prefix of the CodeView structure names in `cvexefmt.h`. Although the CodeView records have very few things in common with the original OMF records, the basic property that the format can be read without understanding all contents still remains.

Listing 1-19 comprises the definitions of various basic CodeView structures, taken from `w2k_img.h`. Some of them loosely correspond to structures found in `cvexefmt.h` and `cvinfo.h`, but are tweaked to the requirements of the `w2k_img.dll` API functions. The `CV_HEADER` structure is present in all CodeView data, regardless of the format version. The `Signature` is a 32-bit format version ID, like `CV_SIGNATURE_NB09` or `CV_SIGNATURE_NB10`. The `lOffset` member specifies the offset of the CodeView directory relative to the header address. In `NB09`-formatted Windows NT

4.0 symbol files, its value seems to be always equal to eight, indicating that the directory follows immediately after the header. The Windows 2000 symbol files contain `NB10` data with `lOffset` set to zero. This data format will be discussed in detail later in this chapter.

```
#define CV_SIGNATURE_NB   'BN'
#define CV_SIGNATURE_NB09 '90BN'
#define CV_SIGNATURE_NB10 '01BN'

// -----------------------------------------------------------------

typedef union _CV_SIGNATURE
    {
    WORD  wMagic;     // 'BN'
    DWORD dVersion;   // 'xxBN'
    BYTE  abText [4]; // "NBxx"
    }
    CV_SIGNATURE, *PCV_SIGNATURE, **PPCV_SIGNATURE;

#define CV_SIGNATURE_ sizeof (CV_SIGNATURE)

// -----------------------------------------------------------------

typedef struct _CV_HEADER
    {
    CV_SIGNATURE Signature;
    LONG         lOffset;
    }
    CV_HEADER, *PCV_HEADER, **PPCV_HEADER;

#define CV_HEADER_ sizeof (CV_HEADER)

// -----------------------------------------------------------------

typedef struct _CV_DIRECTORY
    {
    WORD  wSize;      // in bytes, including this member
    WORD  wEntrySize; // in bytes
    DWORD dEntries;
    LONG  lOffset;
    DWORD dFlags;
    }
    CV_DIRECTORY, *PCV_DIRECTORY, **PPCV_DIRECTORY;

#define CV_DIRECTORY_ sizeof (CV_DIRECTORY)

// -----------------------------------------------------------------
```

```
#define sstModule     0x0120 // CV_MODULE
#define sstGlobalPub  0x012A // CV_PUBSYM
#define sstSegMap     0x012D // SV_SEGMAP

// -----------------------------------------------------------------

typedef struct _CV_ENTRY
    {
    WORD  wSubSectionType;   // sst*
    WORD  wModuleIndex;      // -1 if not applicable
    LONG  lSubSectionOffset; // relative to CV_HEADER
    DWORD dSubSectionSize;   // in bytes, not including padding
    }
    CV_ENTRY, *PCV_ENTRY, **PPCV_ENTRY;

#define CV_ENTRY_ sizeof (CV_ENTRY)

// -----------------------------------------------------------------

typedef struct _CV_NB09 // CodeView 4.10
    {
    CV_HEADER    Header;
    CV_DIRECTORY Directory;
    CV_ENTRY     Entries [];
    }
    CV_NB09, *PCV_NB09, **PPCV_NB09;

#define CV_NB09_ sizeof (CV_NB09)
```

LISTING 1-19. *CodeView Data Structures*

The CodeView `NB09` directory consists of a single `CV_DIRECTORY` structure followed by an array of `CV_ENTRY` items. This is reflected by the `CV_NB09` structure defined at the end of Listing 1-19. It comprises the CodeView header, directory, and entry array. The size of the `Entries[]` array is determined by the `dEntries` member of the `CV_DIRECTORY`. Each `CV_ENTRY` refers to a CodeView subsection of the type specified by the `wSubSectionType` member. `cvexefmt.h` defines no fewer than 21 subsection types. However, the Windows NT 4.0 symbol files make use of only 3 of them: `sstModule (0x0120)`, `sstGlobalPub (0x012A)`, and `sstSegMap (0x012D)`. You will usually see several `sstModule` subsections in a symbol file, but the `sstGlobalPub` and `sstSegMap` subsections are unique. As the name suggests, `sstGlobalPub` is where we will find the global public symbol information of the corresponding module.

The `w2k_img.dll` API function `imgCvEntry()` shown in Listing 1-20 allows easy look up of CodeView directory entries by type. Its `pc09` argument points to a `CV_NB09` structure, that is, to the `NB09` signature of the CodeView data block inside a `.dbg` file. The `dType` argument specifies one of the CodeView subsection type IDs `sst*`, and the `dIndex` value selects a specific subsection instance in cases of multiple subsections of the same type. Therefore, setting `dIndex` to a value other than zero makes sense only if `dType` indicates `sstModule`.

```
PCV_ENTRY WINAPI imgCvEntry (PCV_NB09 pc09,
                             DWORD    dType,
                             DWORD    dIndex)
    {
    DWORD     i, j;
    PCV_ENTRY pce = NULL;

    if ((pc09 != NULL) &&
        (pc09->Header.Signature.dVersion == CV_SIGNATURE_NB09))
        {
        for (i = j = 0; i < pc09->Directory.dEntries; i++)
            {
            if ((pc09->Entries [i].wSubSectionType == dType) &&
                (j++ == dIndex))
                {
                pce = pc09->Entries + i;
                break;
                }
            }
        }
    return pce;
    }

// -------------------------------------------------------------------

PCV_PUBSYM WINAPI imgCvSymbols (PCV_NB09 pc09,
                                PDWORD   pdCount,
                                PDWORD   pdSize)
    {
    PCV_ENTRY  pce;
    PCV_PUBSYM pcp1;
    DWORD      i;
    DWORD      dCount = 0;
    DWORD      dSize  = 0;
    PCV_PUBSYM pcp    = NULL;

    if ((pce = imgCvEntry (pc09, sstGlobalPub, 0)) != NULL)
        {
        pcp = CV_PUBSYM_DATA ((PBYTE) pc09
                              + pce->lSubSectionOffset);
        dSize = pce->dSubSectionSize;
        for (i   = 0; dSize - i >= CV_PUBSYM_;
             i += CV_PUBSYM_SIZE (pcp1))
            {
            pcp1 = (PCV_PUBSYM) ((PBYTE) pcp + i);
            if (dSize - i < CV_PUBSYM_SIZE (pcp1)) break;
            if (pcp1->Header.wRecordType == CV_PUB32) dCount++;
            }
        }
    if (pdCount != NULL) *pdCount = dCount;
    if (pdSize  != NULL) *pdSize  = dSize;
    return pcp;
    }
```

LISTING 1-20. *The* `imgCvEntry()` *and* `imgCvSymbols()` *API Functions*

CODEVIEW SYMBOLS

The lower half of Listing 1-20 shows the `imgCvSymbols()` function that returns a pointer to the first CodeView symbol record. The `sstGlobalPub` subsection consists of a fixed-length `CV_SYMHASH` header, followed by a sequence of variable-length `CV_PUBSYM` records. The definitions of both types are included in Listing 1-21. First, `imgCvSymbols()` calls `imgCvEntry()` to find the `CV_ENTRY` that has its `wSubSectionType` member set to `sstGlobalPub`. If available, it uses the `CV_PUBSYM_DATA()` macro included at the bottom of Listing 1-4 to skip over the leading `CV_SYMHASH` structure. Finally, `imgCvSymbols()` counts the number of symbols by walking through the list of `CV_PUBSYM` records, using the `CV_PUBSYM_SIZE()` macro in Listing 1-21 to compute the size of each record.

The `CV_PUBSYM` sequence bears some resemblance to the contents of an OMF object file. As already noted, an OMF data stream consists of variable-length records, each starting with a tag byte and a length word. `CV_PUBSYM` records are similar. They start with an `OMF_HEADER` that comprises `wRecordSize` and `wRecordType` members. This is just a variant of the OMF principle, different only in that the length word comes first and the tag byte has been extended to 16 bits. The last part of the `CV_PUBSYM` structure is the symbol name, specified in PASCAL format, as is usual in an OMF record. A PASCAL string consists of a leading length byte, followed by 0 to 255 8-bit characters. Contrary to C strings, no terminating zero byte is appended. The `CV_PUBSYM` record ends after the last `Name` character. However, the record is stuffed with filler bytes up to the next 32-bit boundary. This padding is accounted for by the `wRecordSize` value in the `OMF_HEADER`. Note that the `wRecordSize` specifies the size of the `CV_PUBSYM` record, *excluding* the `wRecordSize` member itself. That's why the `CV_PUBSYM_SIZE()` macro in Listing 1-21 adds `sizeof (WORD)` to the `wRecordSize` value to yield the total record size.

```
typedef struct _CV_SYMHASH
    {
    WORD  wSymbolHashIndex;
    WORD  wAddressHashIndex;
    DWORD dSymbolInfoSize;
    DWORD dSymbolHashSize;
    DWORD dAddressHashSize;
    }
    CV_SYMHASH, *PCV_SYMHASH, **PPCV_SYMHASH;

#define CV_SYMHASH_ sizeof (CV_SYMHASH)

// -----------------------------------------------------------------

typedef struct _OMF_HEADER
    {
    WORD wRecordSize; // in bytes, not including this member
    WORD wRecordType;
    }
    OMF_HEADER, *POMF_HEADER, **PPOMF_HEADER;

#define OMF_HEADER_ sizeof (OMF_HEADER)

// -----------------------------------------------------------------

typedef struct _OMF_NAME
    {
    BYTE bLength;     // in bytes, not including this member
    BYTE abName [];
    }
    OMF_NAME, *POMF_NAME, **PPOMF_NAME;

#define OMF_NAME_ sizeof (OMF_NAME)

// -----------------------------------------------------------------


#define S_PUB32  0x0203
#define S_ALIGN  0x0402

#define CV_PUB32 S_PUB32

// -----------------------------------------------------------------

typedef struct _CV_PUBSYM
    {
    OMF_HEADER Header;
    DWORD      dOffset;
    WORD       wSegment;    // 1-based section index     WORD       wTypeIndex;  // 0
    OMF_NAME   Name;        // zero-padded to next DWORD
    }
    CV_PUBSYM, *PCV_PUBSYM, **PPCV_PUBSYM;

#define CV_PUBSYM_ sizeof (CV_PUBSYM)

#define CV_PUBSYM_DATA(_p) \
        ((PCV_PUBSYM) ((PBYTE) (_p) + CV_SYMHASH_))

#define CV_PUBSYM_SIZE(_p) \
        ((DWORD) (_p)->Header.wRecordSize + sizeof (WORD))

#define CV_PUBSYM_NEXT(_p) \
        ((PCV_PUBSYM) ((PBYTE) (_p) + CV_PUBSYM_SIZE (_p)))
```

LISTING 1-21. *The* `CV_SYMHASH` *and* `CV_PUBSYM` *Structures*

If you are scanning the `CV_PUBSYM` stream, you typically will encounter two record types: `S_PUB32 (0x0203)` or `S_ALIGN (0x0402)`. The latter can be safely ignored because it is only padding. The `S_PUB32` records carry the real symbol information. Besides the symbol `Name`, the `wSegment` and `dOffset` members are of interest. `wSegment` specifies a one-based index that identifies the PE file section that contains the symbol. This value minus one can be used as an index into the `IMAGE_SECTION_HEADER` array at the beginning of the `.dbg` file. `dOffset` is the symbol's address relative to the beginning of its PE section. In this context, a symbol address is the entry point of the function or the base address of the global variable associated with the symbol. Normally, the `dOffset` value can simply be added to the `VirtualAddress` of the corresponding `IMAGE_SECTION_HEADER` to yield the address of the symbol relative to the module's base address. However, if the `.dbg` file includes `IMAGE_DEBUG_TYPE_OMAP_TO_SRC` and `IMAGE_DEBUG_TYPE_OMAP_FROM_SRC` subsections, the `dOffset` must pass through an additional conversion layer. The usage of OMAP tables will be discussed later, after introduction of the PDB file format.

The order of the symbols in a CodeView `sstGlobalPub` subsection appears somewhat random. I don't know what principle underlies it. However, I can say for sure that the symbols are *not* sorted by section number, offset, or name. Don't rely on assumptions about the order—if your applications need a specific sorting sequence, you have to sort the symbol records yourself. The `w2k_img.dll` sample library found on the companion CD provides three default symbol orders: by address, by name with case sensitivity, and by name ignoring the character case.

THE INTERNAL STRUCTURE OF .pdb FILES

After installing the Windows 2000 symbol files, the first striking observation is usually that each module now has *two* associated files: one with the `.dbg` extension, as usual, and an additional one with an extension of `.pdb`. Peeking into one of the `.pdb` files reveals the string "Microsoft C/C++ program database 2.00" at its very beginning. So PDB is obviously the acronym of Program Database. Searching for details about the internal PDB structure in the MSDN Library or on the Internet doesn't reveal anything useful, except for a Microsoft Knowledge Base article that classifies this format as Microsoft proprietary (Microsoft Corporation, 2000d). Even Windows guru Matt Pietrek admits:

> *"The format of PDB symbol tables isn't publicly documented. (Even I don't know the exact format, especially as it continues to evolve with each new release of Visual C++.)" (Pietrek 1997a)*

Well, it *might* evolve with each Visual C/C++ release, but for the current version of Windows 2000, I can tell you exactly how its PDB symbol files are structured. This is probably the first time the PDB format has been publicly documented. But first, let's examine how the `.dbg` and `.pdb` files are linked together.

One remarkable property of the Windows 2000 `.dbg` files is that they contain just a very tiny, almost negligible CodeView subsection. Example 1-8 shows the entire CodeView data included in the `ntoskrnl.dbg` file, generated by the `w2k_dump.exe` utility in the `\src\w2k_dump` directory tree of the sample CD. That's all—just those 32 bytes. As usual, the subsection starts with a `CV_HEADER` structure containing the CodeView version signature. This time, it is `NB10`. The MSDN Library (Microsoft 2000a) really doesn't tell us much about this special version:

> *"NB10 The signature for an executable with the debug information stored in a separate PDB file. Corresponds with the formats set forth in NB09 or NB11." (MSDN Library—April 2000 \ Specifications \ Technologies and Languages \ Visual C++ 5.0 Symbolic Debug Information Specification \ Debug Information Format).*

I don't know the internals of the `NB11` format, but the PDB format has almost nothing in common with the `NB09` format discussed above! The first sentence clearly states why the `NB10` data block is that small. All relevant information is moved to a separate file, so the main purpose of this CodeView section is to provide a link to the real data. As Example 1-8 suggests, the symbol information must be sought in the `ntoskrnl.pdb` file in the Windows 2000 symbol setup.

```
Address  | 00 01 02 03-04 05 06 07 : 08 09 0A 0B-0C 0D 0E 0F | 0123456789ABCDEF
———-|———————-:———————-|—————-
00006590 | 4E 42 31 30-00 00 00 00 : 20 7D 23 38-54 00 00 00 | NB10.... }#8T...
000065A0 | 6E 74 6F 73-6B 72 6E 6C : 2E 70 64 62-00 00 00 00 | ntoskrnl.pdb....
```

EXAMPLE 1-8. *Hex Dump of a PDB CodeView Subsection*

If you are wondering what purpose the remaining data in Example 1-8 serves, Listing 1-22 should satisfy your curiosity. The `CV_HEADER` is self-explanatory. The next two members at offset `0x8` and `0xC` are named `dSignature` and `dAge` and play an important role in the linkage of `.dbg` and `.pdb` files. `dSignature` is a 32-bit UNIX-style time stamp, specifying the build date and time of the debug information in seconds since 01-01-1970. The `w2k_img.dll` sample library provides the API functions `imgTimeUnpack()` and `imgTimePack()` to convert this Windows-untypical date/time format back and forth. The purpose of the `dAge` member isn't entirely clear to me. However, it appears that its value is initially set to one and incremented each time the PDB data is rewritten. The `dSignature` and `dAge` values together constitute a 64-bit ID that can be used by debuggers to verify that a given PDB file matches the `.dbg` file referring to it. The PDB file contains duplicates of both values in one of its data streams, so a debugger can refuse processing a `.dbg/.pdb` pair of files with unmatched `dSignature` and `dAge` information.

Whenever you are faced with an unknown data format, the first thing to do is to run some examples of it through a hex dump viewer. The `w2k_dump.exe` utility on this book's companion CD does a good job in this respect. Examining the hex dump of a Windows 2000 PDB file such as `ntoskrnl.pdb` or `ntfs.pdb` reveals some interesting properties:

- The file seems to be divided into blocks of fixed size—typically `0x400` bytes.
- Some blocks consist of long runs of 1-bits, occasionally interrupted by shorter sequences of 0-bits.
- The information in the file is not necessarily contiguous. Sometimes, the data end abruptly at a block boundary, but continue somewhere else in the file.
- Some data blocks appear repeatedly within the file.

```
typedef struct _CV_NB10 // PDB reference
    {
    CV_HEADER    Header;
    DWORD        dSignature;   // seconds since 01-01-1970
    DWORD        dAge;         // 1++
    BYTE         abPdbName []; // zero-terminated
    }
    CV_NB10, *PCV_NB10, **PPCV_NB10;

#define CV_NB10_ sizeof (CV_NB10)
```

LISTING 1-22. *The CodeView* NB10 *Subsection*

It took some time for me to finally realize that these are typical properties of a compound file. A compound file is a small file system packaged into a single file. The "file system" metaphor readily explains some of the above observations:

- A file system subdivides a disk into *sectors* of fixed size and groups the sectors into *files* of variable size. The sectors representing a file can be located anywhere on the disk and don't need to be contiguous—the file/sector assignments are defined in a *file directory.*
- A compound file subdivides a raw disk file into *pages* of fixed size and groups the pages into *streams* of variable size. The pages representing a file can be located anywhere in the raw disk file and don't need to be contiguous—the stream/page assignments are defined in a *stream directory.*

Obviously, almost any assertions about file systems can be mapped to compound files by simply replacing "sector" by "page," and "file" by "stream." The file system metaphor explains why a PDB file is organized in fixed-size blocks. It also explains why the blocks are not necessarily contiguous. What about the pages with the masses of 1-bits? Actually, this type of data is something very common in file systems. To keep track of used and unused sectors on the disk, many file systems maintain an allocation bit array that provides one bit for each sector (or sector cluster). If a sector is unused, its bit is set. Whenever the file system allocates space for a file, it searches for unused sectors by scanning the allocation bits. After adding a sector to a file, its allocation bit is set to zero. The same procedure is applied to the pages and streams of a compound file. The long runs of 1-bits represent unused pages, and the 0-bits are assigned to existing streams.

The only thing that remains is the observation that some data blocks reoccur within a PDB file. The same thing happens with sectors on a disk. When a file in a file system is rewritten a couple of times, each write operation may use different sectors to store the data. Thus, it can happen that the disk contains free sectors with older duplicates of the file information. This doesn't constitute a problem for the file system. If the sector is marked free in the allocation bit array, it is unimportant what data it contains. As soon as the sector is reclaimed for another file, the data will be overwritten anyway. Applying the file system metaphor once more to compound files, this means that the observed duplicate pages are usually left over from earlier versions of a stream that has been rewritten to different pages in the compound file. They can be safely ignored; all we have to care for are the pages that are referred to by the stream directory. The remaining unassigned pages should be regarded as garbage.

With the basic paradigm of PDB files being introduced now, we can step to the more interesting task of examining their basic building blocks. Listing 1-23 shows the layout of the PDB header. The `PDB_HEADER` starts with a lengthy signature that specifies the PDB version as a text string. The text is terminated with an end-of-file (EOF) character (ASCII code `0x1A`) and supplemented with the magic number `0x0000474A`, or `"JG\0\0"` if interpreted as a string. Maybe these are the initials of the designer of the PDB format. The embedded EOF character has the nice effect that an unknowledgeable user can issue a command such as `type ntoskrnl.pdb` in a console window without getting garbage on the screen. The only thing that will be displayed is the message `Microsoft C/C++ program database 2.00\r\n.` All Windows 2000 symbol files are shipped as PDB 2.00 files. Apparently, a PDB 1.00 format exists as well, but it seems to be structured quite differently.

```
#define PDB_SIGNATURE_200 \
        "Microsoft C/C++ program database 2.00\r\n\x1AJG\0"

#define PDB_SIGNATURE_TEXT 40

// -----------------------------------------------------------------

typedef struct _PDB_SIGNATURE
    {
    BYTE abSignature [PDB_SIGNATURE_TEXT+4]; // PDB_SIGNATURE_nnn
    }
    PDB_SIGNATURE, *PPDB_SIGNATURE, **PPPDB_SIGNATURE;

#define PDB_SIGNATURE_ sizeof (PDB_SIGNATURE)

// -----------------------------------------------------------------

#define PDB_STREAM_FREE -1

// -----------------------------------------------------------------
```

(continued)

```
typedef struct _PDB_STREAM
    {
    DWORD dStreamSize;   // in bytes, -1 = free stream
    PWORD pwStreamPages; // array of page numbers
    }
    PDB_STREAM, *PPDB_STREAM, **PPPDB_STREAM;

#define PDB_STREAM_ sizeof (PDB_STREAM)

// -----------------------------------------------------------------

#define PDB_PAGE_SIZE_1K   0x0400 // bytes per page
#define PDB_PAGE_SIZE_2K   0x0800
#define PDB_PAGE_SIZE_4K   0x1000

#define PDB_PAGE_SHIFT_1K  10     // log2 (PDB_PAGE_SIZE_*)
#define PDB_PAGE_SHIFT_2K  11
#define PDB_PAGE_SHIFT_4K  12

#define PDB_PAGE_COUNT_1K  0xFFFF // page number < PDB_PAGE_COUNT_*
#define PDB_PAGE_COUNT_2K  0xFFFF
#define PDB_PAGE_COUNT_4K  0x7FFF

// -----------------------------------------------------------------

typedef struct _PDB_HEADER
    {
    PDB_SIGNATURE Signature;      // PDB_SIGNATURE_200
    DWORD         dPageSize;      // 0x0400, 0x0800, 0x1000
    WORD          wStartPage;     // 0x0009, 0x0005, 0x0002
    WORD          wFilePages;     // file size / dPageSize
    PDB_STREAM    RootStream;     // stream directory
    WORD          awRootPages []; // pages containing PDB_ROOT
    }
    PDB_HEADER, *PPDB_HEADER, **PPPDB_HEADER;

#define PDB_HEADER_ sizeof (PDB_HEADER)
```

LISTING 1-23. *The PDB File Header*

Following the signature at offset `0x2C` is a `DWORD` named `dPageSize` that specifies the size of the compound file pages in bytes. Legal values are `0x0400` (1 KB), `0x0800` (2 KB), and `0x1000` (4 KB). The `wFilePages` member reflects the total number of pages used by the PDB file image. The result of multiplying this value by the page size should always exactly match the file size in bytes. `wStartPage` is a zero-based page number that points to the first data page. The byte offset of this page can be computed by multiplying the page number by the page size. Typical values are *9* for 1-KB pages (byte offset `0x2400`), *5* for 2-KB pages (byte offset `0x2800`), or *2* for 4-KB pages (byte offset

`0x2000`). The pages between the `PDB_HEADER` and the first data page are reserved for the allocation bit array of the compound file, always starting at the beginning of the second page. This means that the PDB file maintains `0x2000` bytes with `0x10000` allocation bits if the page size is 1 or 2 KB, and `0x1000` bytes with `0x8000` allocation bits if the page size is 4 KB. In turn, this implies that the maximum amount of data a PDB file can manage is 64 MB in 1-KB page mode, and 128 MB in 2-KB or 4-KB page mode.

The `RootStream` and `awRootPages[]` members concluding the `PDB_HEADER` describe the location of the stream directory within the PDB file. As already noted, the PDB file is conceptually a collection of variable-length streams that carry the actual data. The locations and compositions of the streams are managed in a single stream directory. Odd as it may seem, the stream directory itself is stored in a stream. I have called this special stream the "root stream." The root stream holding the stream directory can be located anywhere in the PDB file. Its location and size are supplied by the `RootStream` and `awRootPages[]` members of the `PDB_HEADER`. The `dStreamSize` member of the `PDB_STREAM` substructure specifies the number of pages occupied by the stream directory, and the entries in the `awRootPages[]` array point to the pages containing the data.

Let's illustrate this with a simple example. The hex dump excerpt in Example 1-9 shows the `PDB_HEADER` of the `ntoskrnl.pdb` file. The values referenced are underlined. Obviously, this PDB file uses a page size of `0x0400` bytes and comprises `0x02D1` pages, resulting in a file size of `0xB4400` (`738,304` in decimal notation). A quick check with the `dir` command shows that this value is correct. The root stream size is `0x5B0` bytes. With a page size of `0x400` bytes, this means that the `awRootPages[]` array contains two entries, found at the file offsets `0x3C` and `0x3E`. The values in these slots are page numbers that need to be multiplied by the page size to yield the corresponding byte offsets. In this case, the results are `0xB2000` and `0xB2800`.

The bottom line of this computation is that the stream directory of the `ntoskrnl.exe` PDB file is located in two file pages, extending from `0xB2000` to `0xB23FF` and `0xB2800` to `0xB29AF`, respectively. Parts of these ranges are shown in Example 1-10.

```
Address  | 00 01 02 03-04 05 06 07 : 08 09 0A 0B-0C 0D 0E 0F | 0123456789ABCDEF
─────────|──────────────:───────────:|─────────────────────────|
00000000 | 4D 69 63 72-6F 73 6F 66 : 74 20 43 2F-43 2B 2B 20 | Microsoft C/C++
00000010 | 70 72 6F 67-72 61 6D 20 : 64 61 74 61-62 61 73 65 | program database
00000020 | 20 32 2E 30-30 0D 0A 1A : 4A 47 00 00-00 04 00 00 |  2.00...JG......
00000030 | 09 00 D1 02-B0 05 00 00 : 5C 00 78 00-C8 02 CA 02 | ..Ñ.°...\.x.È.Ê.
```

EXAMPLE 1-9. *A Sample PDB Header*

```
Address  | 00 01 02 03-04 05 06 07 : 08 09 0A 0B-0C 0D 0E 0F | 0123456789ABCDEF
---------|-------------------------:-------------------------|
000B2000 | 08 00 00 00-B0 05 00 00 : 98 22 28 00-3A 00 00 00 | ....°...?"(.:...
000B2010 | 88 57 26 00-38 00 00 00 : 78 57 26 00-A9 02 04 00 | ?W&.8...xW&.©...
000B2020 | F8 BA E9 00-00 00 00 00 : 68 57 26 00-04 40 00 00 | øºé.....hW&..@..
000B2030 | C8 29 28 00-B4 9E 01 00 : 08 90 ED 00-3C DF 04 00 | È)(.´?...•í.<ß..
000B2040 | 08 BD E9 00-12 00 C9 02 : C7 02 13 00-C6 02 C6 01 | .½é...É.Ç...Æ.Æ.
000B2050 | C7 01 C8 01-C9 01 CA 01 : CB 01 CC 01-CD 01 CE 01 | Ç.È.É.Ê.Ë.Ì.Í.Î.
...
000B23A0 | BD 00 BE 00-BF 00 C0 00 : C1 00 C2 00-C3 00 C4 00 | ½.¾.¿.À.Á.Â.Ã.Ä.
000B23B0 | C5 00 C6 00-C7 00 C8 00 : C9 00 CA 00-CB 00 CC 00 | Å.Æ.Ç.È.É.Ê.Ë.Ì.
000B23C0 | CD 00 CE 00-CF 00 D0 00 : D1 00 D2 00-D3 00 D4 00 | Í.Î.Ï.Ð.Ñ.Ò.Ó.Ô.
000B23D0 | D5 00 D6 00-D7 00 D8 00 : D9 00 DA 00-DB 00 DC 00 | Õ.Ö.×.Ø.Ù.Ú.Û.Ü.
000B23E0 | DD 00 DE 00-DF 00 E0 00 : E1 00 E2 00-E3 00 E4 00 | _.þ.ß.à.á.â.ã.ä.
000B23F0 | E5 00 E6 00-E7 00 E8 00 : E9 00 EA 00-EB 00 EC 00 | å.æ.ç.è.é.ê.ë.ì.
---------|-------------------------:-------------------------|
000B2800 | ED 00 EE 00-EF 00 F0 00 : F1 00 F2 00-F3 00 F4 00 | í.î.ï.õ.ñ.õ.ó.ô.
000B2810 | F5 00 F6 00-F7 00 F8 00 : F9 00 FA 00-FB 00 FC 00 | õ.ö.÷.ø.ù.ú.û.ü.
000B2820 | FD 00 FE 00-FF 00 00 01 : 01 01 02 01-03 01 04 01 | ý.þ.ÿ...........
000B2830 | 05 01 06 01-07 01 08 01 : 09 01 0A 01-0B 01 0C 01 | ................
000B2840 | 0D 01 0E 01-0F 01 10 01 : 11 01 12 01-13 01 14 01 | ................
000B2850 | 15 01 16 01-17 01 18 01 : 19 01 1A 01-1B 01 1C 01 | ................
...
000B2950 | 95 01 96 01-97 01 98 01 : 99 01 9A 01-9B 01 9C 01 | ?.?.?.?.?.?.?.?.
000B2960 | 9D 01 9E 01-9F 01 A0 01 : A1 01 A2 01-A3 01 A4 01 | •.?.?. .¡.¢.£.¤ .
000B2970 | A5 01 A6 01-A7 01 A8 01 : A9 01 AA 01-AB 01 AC 01 | ¥.¦.§.¨.©.ª.«.¬.
000B2980 | AD 01 AE 01-AF 01 B0 01 : B1 01 B2 01-B3 01 B4 01 | =.®.¯.°.±.².³.´.
000B2990 | B5 01 B6 01-B7 01 B8 01 : B9 01 BA 01-BB 01 BC 01 | µ.¶.·.¸.¹.º.».¼.
000B29A0 | BD 01 BE 01-BF 01 C0 01 : C1 01 C2 01-C3 01 C4 01 | ½.¾.¿.À.Á.Â.Ã.Ä.
```

EXAMPLE 1-10. *Excerpts from a Sample PDB Stream Directory*

The stream directory is composed of two sections: a header part in the form of a `PDB_ROOT` structure, as defined in Listing 1-24, and a data part consisting of an array of 16-bit page numbers. The `wCount` member of the `PDB_ROOT` section specifies the number of streams stored in the PDB compound file. The `aStreams[]` array contains a `PDB_STREAM` entry (see Listing 1-23) for each stream, and the page number slots follow immediately after the last `aStreams[]` entry. In Example 1-10, the number of streams is eight, as the underlined value at offset `0xB2000` indicates. The subsequent eight `PDB_STREAM` structures define streams of size `0x5B0, 0x3A, 0x38, 0x402A9, 0x0, 0x4004, 0x19EB4,` and `0x4DF3C,` respectively. These values are underlined in Example 1-10, too. Expressed in 1-KB pages, the stream sizes are `0x2, 0x1, 0x1, 0x101, 0x0, 0x11, 0x68,` and `0x138,` yielding a total of `0x2B6` pages used by the streams. The first underlined value after the `PDB_STREAM` array is the first slot of the page number list. Counting two bytes per page number, and taking into account that the page directory is interrupted by one page that belongs somewhere else, the next offset after the page numbers should be `0xB2044 + 0x400 + (0x2B6 * 2) = 0xB29B0,` which fits perfectly into the picture.

```
#define PDB_STREAM_DIRECTORY 0
#define PDB_STREAM_PDB       1
#define PDB_STREAM_PUBSYM    7

// -----------------------------------------------------------------

typedef struct _PDB_ROOT
    {
    WORD       wCount;      // < PDB_STREAM_MAX
    WORD       wReserved;   // 0
    PDB_STREAM aStreams []; // stream #0 reserved for stream table
    }
    PDB_ROOT, *PPDB_ROOT, **PPPDB_ROOT;

#define PDB_ROOT_ sizeof (PDB_ROOT)
```

LISTING 1-24. *The PDB Stream Directory*

Finding the page number block associated with a given stream is somewhat tricky, because the page directory does not provide any cues except the stream size. If you are interested in stream 3, you have to compute the number of pages occupied by streams 1 and 2 to get the desired start index within the page number array. Once the stream's page number list is located, reading the stream data is simple. Just walk through the list and multiply each page number by the page size to yield the file offset, and read pages from the computed offsets until the end of the stream is reached. On first look, parsing a PDB file seemed rather tough. But it turns out that it is actually quite simple—probably much simpler than parsing a `.dbg` file. The compound-file nature of the PDB format with its clear-cut random access to stream pages reduces the task of reading a stream to a mere concatenation of fixed-sized pages. I'm amazed at this elegant data access mechanism!

An even greater benefit of the PDB format becomes apparent when updating an existing PDB file. Inserting data into a file with a sequential structure usually means reshuffling large portions of the contents. The PDB file's random-access structure borrowed from file systems allows addition and deletion of data with minimal effort, just as files can be modified with ease on a file system media. Only the stream directory has to be reshuffled when a stream grows or shrinks across a page boundary. This important property facilitates incremental updating of PDB files. Microsoft states the following in a Knowledge Base article titled "INFO: PDB and DBG Files—What They Are and How They Work":

"The .PDB extension stands for 'program database.' It holds the new format for storing debugging information that was introduced in Visual C++ version 1.0. In the future, the .PDB file will also hold other project state information.

One of the most important motivations for the change in format was to allow incremental linking of debug versions of programs, a change first introduced in Visual C++ version 2.0" (Microsoft Corporation 2000e).

Now that the internal format of PDB files is clear, the next problem is to identify the contents of their streams. After examining various PDB files, I have come to the conclusion that each stream number serves a predefined purpose. The first stream seems to always contain a stream directory, and the second one contains information about the PDB file that can be used to verify that the file matches an associated `.dbg` file. For example, the latter stream contains `dSignature` and `dAge` members that should have the same values as the corresponding members of an `NB10` CodeView section, as outlined in Listing 1-22. The eighth stream is most interesting in the context of this chapter because it hosts the CodeView symbol information we have been seeking. The meaning of the other streams is still unclear to me and is another area for future research.

I am not going to include PDB reader sample code here because this would exceed the scope of this chapter. Instead, I encourage you to peek into the `w2k_img.c` and `w2k_img.h` source files on the sample CD. Look for functions named `imgPdb*()` and data items called `PDB_*` for extensive code and data. By the way, the CD contains a ready-to-run PDB stream reader with full source code. You already know this program—it is the `w2k_dump.exe` utility that I have used to create some of the hex dump examples above. This simple console-mode utility provides a `+p` command line option that enables PDB stream decomposition. If the specified file is not a valid PDB file, the program falls back to sequential hex dump mode. The Visual C/C++ project files of `w2k_dump.exe` are found on the CD in the `\src\w2k_dump` directory tree.

PDB Symbols

After this long but hopefully interesting detour through the PDB format, it is time to return to our initial mission: the extraction of CodeView symbol information. Fortunately, this task is quite similar to the enumeration of public symbols in an `NB09` CodeView subsection. Once the stream containing the symbols is located, we are again faced with a sequence of OMF-like records of variable size. Unfortunately, the `NB09` and `NB10` record formats differ somewhat, but the deviations are only marginal. Listing 1-25 shows the layout of the `PDB_PUBSYM` structure. Compared with the corresponding `CV_PUBSYM` structure of the NB09 format, included in Listing 1-21, the `dOffset` and `wSegment` members have moved a bit toward the end. This and the fact that the tag value of PDB symbols is `0x1009` instead of `0x0203` are the most remarkable differences.

```
#define PDB_PUB32 0x1009

// -----------------------------------------------------------------

typedef struct _PDB_PUBSYM
    {
    OMF_HEADER Header;
    DWORD      dReserved;
    DWORD      dOffset;
    WORD       wSegment;    // 1-based section index
    OMF_NAME   Name;        // zero-padded to next DWORD
    }
    PDB_PUBSYM, *PPDB_PUBSYM, **PPPDB_PUBSYM;

#define PDB_PUBSYM_ sizeof (PDB_PUBSYM)

#define PDB_PUBSYM_SIZE(_p) \
        ((DWORD) (_p)->Header.wRecordSize + sizeof (WORD))

#define PDB_PUBSYM_NEXT(_p) \
        ((PPDB_PUBSYM) ((PBYTE) (_p) + PDB_PUBSYM_SIZE (_p)))
```

LISTING 1-25. *The* `PDB_PUBSYM` *Structure*

The `IMG_PUBSYM` union in Listing 1-26 is a convenient means to reference symbol records regardless of their type. This union can be interpreted in three ways:

1. `OMF_HEADER`: This point of view should be assumed unless the symbol type is known. The header provides just enough information to identify the symbol type or to skip to the next record.
2. `CV_PUBSYM`: This interpretation is valid only if the `wRecordType` of the `OMF_HEADER` is set to `CV_PUB32 (0x0203)`.
3. `PDB_PUBSYM`: This interpretation is valid only if the `wRecordType` of the `OMF_HEADER` is set to `PDB_PUB32 (0x1009)`.

The `IMG_PUBSYM_SIZE()` and `IMG_PUBSYM_NEXT()` macros found at the end of Listing 1-26 allow type-independent determination of the size of the current record and the address of the subsequent one, respectively.

SYMBOL ADDRESS COMPUTATION

The `wSegment` and `dOffset` members of the `CV_PUBSYM` and `PDB_PUBSYM` symbol records, together with the `IMAGE_SECTION_HEADER` array at the beginning of the `.dbg` file, supply necessary information for the computation of the address of a symbol relative to the beginning of the module's base address. If the `.dbg` file

```
typedef union _IMG_PUBSYM
    {
    OMF_HEADER Header;    // CV_PUB32 or PDB_PUB32
    CV_PUBSYM  CvPubSym;
    PDB_PUBSYM PdbPubSym;
    }
    IMG_PUBSYM, *PIMG_PUBSYM, **PPIMG_PUBSYM;

#define IMG_PUBSYM_ sizeof (IMG_PUBSYM)

#define IMG_PUBSYM_SIZE(_p) \
        ((DWORD) (_p)->Header.wRecordSize + sizeof (WORD))

#define IMG_PUBSYM_NEXT(_p) \
        ((PIMG_PUBSYM) ((PBYTE) (_p) + IMG_PUBSYM_SIZE (_p)))
```

LISTING 1-26. *The* `IMG_PUBSYUM` *Union*

doesn't contain any OMAP data in the form of `IMAGE_DEBUG_TYPE_OMAP_TO_SRC` and `IMAGE_DEBUG_TYPE_OMAP_FROM_SRC` subsections, the address computation algorithm is straightforward:

- Read the `wSegment` value of the symbol record and decrement it by one.
- Use the resulting index to look up the `IMAGE_SECTION_HEADER` of the target section where the symbol resides.
- Retrieve the `VirtualAddress` of this `IMAGE_SECTION_HEADER`.
- Add the `dOffset` value of the symbol record.

In case the load address of the module is known, the absolute linear address of the symbol can be determined by simply adding the computed relative address to the base address. The `ImageBase` member of the `IMAGE_SEPARATE_DEBUG_HEADER` at the very beginning of the `.dbg` file specifies the module's preferred load address. Unfortunately, this address isn't very helpful because many kernel modules are actually loaded to completely different addresses. For example, `ntoskrnl.dbg` reports a preferred load address of `0x00400000`, which is certainly wrong because this address is far outside the kernel memory range. Therefore, the `w2k_img.dll` provides the `imgModuleBase()` API function that attempts to locate kernel modules in memory. It uses the undocumented `NtQuerySystemInformation()` function exported by `ntdll.dll` to retrieve a list of modules currently found in memory. However, this function works on Windows 2000/NT only. For Windows 9x compatibility, `imgModuleBase()` loads `ntdll.dll` dynamically, so `w2k_img.dll` won't blow up immediately with a dynalink error while it is being loaded. Therefore, it always returns a `NULL` pointer on Windows 9x. This is the same value that you will get on Windows 2000 and Windows NT 4.0 if the specified module is not present in memory.

OMAP Address Conversion

Several Windows 2000 symbol files contain OMAP subsections, identified by `IMAGE_DEBUG_DIRECTORY` entries with `Type` IDs of `IMAGE_DEBUG_TYPE_OMAP_TO_SRC` and `IMAGE_DEBUG_TYPE_OMAP_FROM_SRC`. OMAP is yet another undocumented feature of the Microsoft development tools, so the reasons for its existence are still somewhat speculative. The OMAP data inside a `.dbg` file consist of two arrays of `OMAP_TO_SRC` and `OMAP_FROM_SRC` structures, as outlined in Listing 1-27, and this information is used in the computation of symbol addresses from the offset values stored in `CV_PUBSYM` or `PDB_PUBSYM` records.

In one of his fine *MSJ* "Under the Hood" articles about Microsoft debug information, Matt Pietrek writes his thoughts about OMAP:

> *Yet another form of debug information is relatively new and undocumented, except for a few obscure references in WINNT.H and the Win32 SDK help. This type of information is known as OMAP. Apparently, as part of Microsoft's internal build procedure, small fragments of code in EXEs and DLLs are moved around to put the most commonly used code at the beginning of the code section. This presumably keeps the process memory working set as small as possible. However, when shifting around the blocks of code, the corresponding debug information isn't updated. Instead, OMAP information is created. It lets symbol table code translate between the original address in a symbol table and the modified address where the variable or line of code really exists in memory. (Pietrek 1997a)*

```
typedef struct _OMAP_TO_SRC
    {
    DWORD dTarget;
    DWORD dSource;
    }
    OMAP_TO_SRC, *POMAP_TO_SRC, **PPOMAP_TO_SRC;

#define OMAP_TO_SRC_ sizeof (OMAP_TO_SRC)

// -----------------------------------------------------------------

typedef struct _OMAP_FROM_SRC
    {
    DWORD dSource;
    DWORD dTarget;
    }
    OMAP_FROM_SRC, *POMAP_FROM_SRC, **PPOMAP_FROM_SRC;

#define OMAP_FROM_SRC_ sizeof (OMAP_FROM_SRC)
```

Listing 1-27. *The* `OMAP_TO_SRC` *and* `OMAP_FROM_SRC` *Table Entries*

And more than 2 years later, *MSJ* columnist John Robbins elaborates on this assumption in the October 1997 "Bugslayer":

> *The undocumented OMAP information is interesting because it appears to have something to do with basic block relocations. (Fellow* MSJ *colleague Matt Pietrek briefly discussed this in the May 1997 "Under the Hood" column.) My guess is that Microsoft has some sort of internal tool that packs the binary so that the most common code is pushed up to the front and the rest is put in the rear so that the working set is much smaller. Consequently, this binary rearrangement makes the program faster because it will not have to page in as much of the program. (Robbins 1999)*

Although the working set argument is striking, the fact that the `ntoskrnl.exe` module makes heavy use of OMAP seems to be at odds with it. As I will show in Chapter 4, the entire `ntoskrnl.exe` module is mapped to a single 4-MB memory page that is always present in memory, so splitting the code into more frequently and more rarely used fractions shouldn't be of benefit with respect to paging. My assumption is that this split is supposed to aid the processor's instruction prefetch. Examination of the OMAP tables reveals that the addresses they contain typically point to the beginning of a function, to an instruction that immediately follows a jump or call, or to unused filler code. This suggests that the OMAP data is used to reshuffle the branches of `if/else` instructions. Obviously, the Windows 2000 kernel developers at Microsoft can somehow tell the compiler whether the `if` or `else` branch is executed more frequently, so the code fraction that is run less frequently can be moved out of the way. Normally, a compiler tends to keep the code of a function in a monolithic block, and doesn't split up `if/else` branches. In the Windows 2000 kernel modules, however, it can be easily observed that large functions with numerous `if/else` clauses are heavily fragmented. The fact that the OMAP code atoms correspond to conditional branches leads me to the assumption that OMAP has something to do with branch prediction. If less frequently executed branches are separated from the more frequently used ones, the CPU can perform more effective instruction prefetch.

The `OMAP_TO_SRC` table converts a real instruction offset to a source offset, for example, the real offset of the `ExInterlockedAddLargeInteger()` API function relative to the base address of `ntoskrnl.exe` is `0x0000231E`. To verify this, enter the command `u ExInterlockedAddLargeInteger` at the Kernel Debugger prompt—it will unassemble a couple of lines, starting at the linear address `0x8040231E`. Subtracting the `ntoskrnl.exe` load address `0x80400000` yields `0x0000231E`, as expected. If you scan the `OMAP_TO_SRC` table inside `ntoskrnl.dbg`, you will find an entry whose `dTarget` member is set to this offset, and the corresponding `dSource` offset is `0x0005E7E4`. The `ExInterlockedAddLargeInteger()` function is located in the `.text` section, and the offset of this section relative to the image base address is

0x000004C0 according to its IMAGE_SECTION_HEADER. Subtracting the section offset from the source offset yields a raw symbol offset of 0x0005E324, and this is exactly the dOffset value of the PDB_PUBSYM record that defines the ExInterlockedAdd-LargeInteger symbol. That's easy, isn't it? Well, not really.

The OMAP_TO_SRC entries are always sorted in ascending order with respect to the target address. This is a good idea, because it facilitates the lookup of addresses by binary searching. The OMAP_FROM_SRC table is essentially a replica of the OMAP_TO_SRC table, but with all source and target addresses swapped and resorted by source address. This dual-table approach allows easy address translation in both directions.

An OMAP problem that puzzled me for several days is that you cannot make immediate use of the VirtualAddress values stored in the IMAGE_SECTION_HEADER array of the .dbg file while converting from source to target addresses via the OMAP_FROM_SRC table. In all PE sections except for the first one, this will result in target addresses that are too high. The reason for this strange effect is that the VirtualAddress values are valid in the target address world only. On the source address side, different section addresses apply. The main problem is now to find out the source addresses of the PE sections. After scanning the .dbg and .pdb files repeatedly—but without success—for any tables that might perform this translation, I eventually ended up with a trick that works fine, although I'm not sure whether it is legal. To determine the source address of a section, I simply enumerate all OMAP_TO_SRC entries that belong to this section and compute the minimum of their source addresses. This procedure is based on the assumption that OMAP is just a permutation of code fractions, so minimizing the source addresses of a section means finding the snippet that has been bumped to the top of the section. This address should correspond to the source address of the section. I have applied this technique to numerous Windows 2000 symbol files, and thus far, it has not failed.

If it sounds appealing to implement a symbol file parser based on the above information, just do it! Or, you can use the w2k_img.dll on the sample CD as is or rip out code from it. This DLL contains everything you need to take .dbg and .pdb files apart and much more. The most powerful API function set it exports is the imgTable*() group. It comprises the three functions listed in Table 1-5, whose prototypes are shown in Listing 1-28. They are intended for use by debugger or disassembler writers. With the imgTableLookup() API function, an application can display symbols instead of raw addresses, and the imgTableResolve() function can be used as a basis for a symbol search option. Both functions are carefully optimized for speed, which is of great benefit to applications that browse large amounts of symbol information. The sample symbol browser presented below is based on w2k_img.dll and is able to dump a sorted list of all ntoskrnl.exe symbols with lots of additional information to a file in less than 2 seconds.

TABLE 1-5. *Symbol Table Management Functions Exported by* `w2k_img.dll`

NAME	DESCRIPTION
imgTableLoad()	Builds an `IMG_TABLE` symbol table from a `.dbg` or `.pdb` file
imgTableLookup()	Finds an `IMG_ENTRY` symbol table entry matching a symbol address
imgTableResolve()	Finds an `IMG_ENTRY` symbol table entry matching a symbol name

```
PIMG_TABLE WINAPI imgTableLoad (PTBYTE ptPath,
                               PVOID  pBase);

PIMG_ENTRY WINAPI imgTableLookup (PIMG_TABLE pit,
                                  PVOID      pAddress,
                                  PDWORD     dOffset);

PIMG_ENTRY WINAPI imgTableResolve (PIMG_TABLE pit,
                                   PBYTE      pbSymbol);
```

LISTING 1-28. *Prototypes of the Symbol Table Management Functions*

Listing 1-29 is a compilation of the structures on which the `imgTable*()` functions operate. Apparently, they don't resemble the CodeView and PDB structures discussed above. In fact, the symbol table management functions inside `w2k_img.dll` completely rearrange the information found in the symbol files, allowing easier and faster processing. The most fundamental structure is the `IMG_TABLE`, which comprises the entire symbol information. It is composed of a fixed-size header, an array of `IMG_ENTRY` structures, and three `IMG_INDEX` arrays. Because the arrays are of variable size depending on the number of symbols, the `IMG_TABLE` also contains three pointers to the `IMG_INDEX` base addresses. As indicated by the comments in Listing 1-29, the indexes are sorted by address, by name considering character case, and by name ignoring character case. These indexes are not only convenient for applications that output symbol lists, but also for the `imgTableLookup()` and `imgTableResolve()` functions because they allow them to perform fast binary searches for addresses and names.

One particularly nice feature of the `IMG_ENTRY` structure is that it specifies the calling convention assigned to a symbol. This information is derived directly from the symbol decoration, based on the rules in Table 1-4. This nontrivial task is done by the `imgSymbolUndecorate()` function shown in Listing 1-30. First, it tries to identify one of the common prefixes, listed in the `apbPrefixes[]` array. In the next step, the code looks for a stack size trailer consisting of an `@` character and a decimal number. The calling convention is detected along the way by testing for special prefix/trailer combinations. `w2k_img.dll` undecorates symbols with high reliability. Actually, it correctly handles all `__fastcall` import thunks that `imagehlp.dll` is unable to manage. `imgSymbolUndecorate()`, however, does not attempt to undecorate C++ and PCH symbols. Maybe I will add this feature in a future version of `w2k_img.dll`.

```
#define IMG_CONVENTION_UNDEFINED    0
#define IMG_CONVENTION_STDCALL      1
#define IMG_CONVENTION_CDECL        2
#define IMG_CONVENTION_FASTCALL     3

// ----------------------------------------------------------------

typedef struct _IMG_ENTRY
    {
    DWORD dSection;          // 1-based section number
    PVOID pAddress;          // symbol address
    DWORD dConvention;       // calling convention IMG_CONVENTION_*
    DWORD dStack;            // number of argument stack bytes
    BOOL  fExported;         // TRUE if exported symbol
    BOOL  fSpecial;          // TRUE if special symbol
    BYTE  abSection   [IMAGE_SIZEOF_SHORT_NAME+4]; // section name
    BYTE  abSymbol    [256]; // undecorated symbol name
    BYTE  abDecorated [256]; // decorated symbol name
    }
    IMG_ENTRY, *PIMG_ENTRY, **PPIMG_ENTRY;

#define IMG_ENTRY_ sizeof (IMG_ENTRY)

// ----------------------------------------------------------------

typedef struct _IMG_INDEX
    {
    PIMG_ENTRY apEntries [1];
    }
    IMG_INDEX, *PIMG_INDEX, **PPIMG_INDEX;

#define IMG_INDEX_ sizeof (IMG_INDEX)
#define IMG_INDEX__(_n) ((_n) * IMG_INDEX_)

// ----------------------------------------------------------------

typedef struct _IMG_TABLE
    {
    DWORD      dSize;      // table size in bytes
    DWORD      dSections;  // number of sections
    DWORD      dSymbols;   // number of symbols
    DWORD      dTimeStamp; // module time stamp (sec since 1-1-1970)
    DWORD      dCheckSum;  // module checksum
    PVOID      pBase;      // module base address
    PIMG_INDEX piiAddress; // entries sorted by address
    PIMG_INDEX piiName;    // entries sorted by name
    PIMG_INDEX piiNameIC;  // entries sorted by name (ignore case)
    BOOL       fUnicode;   // character format
    union
        {
```

(continued)

```
        TBYTE atPath [MAX_PATH]; // .dbg file path
        BYTE  abPath [MAX_PATH]; // .dbg file path (ANSI)
        WORD  awPath [MAX_PATH]; // .dbg file path (Unicode)
        };
    IMG_ENTRY  aEntries []; // symbol info array
    }
    IMG_TABLE, *PIMG_TABLE, **PPIMG_TABLE;

#define IMG_TABLE_ sizeof (IMG_TABLE)

#define IMG_TABLE__(_n) \
        (IMG_TABLE_ + ((_n) * IMG_ENTRY_) + (3 * IMG_INDEX__ (_n)))
```

LISTING 1-29. *Symbol Table Management Structures*

```
DWORD WINAPI imgSymbolUndecorate (PBYTE  pbSymbol,
                                  PBYTE  pbBuffer,
                                  PDWORD pdConvention)
    {
    PBYTE apbPrefixes [] = {"__imp__", "__imp_@", "__imp_",
                            "_", "@", "\x7F",
                            NULL};

    BYTE  abBuffer [256] = "";
    DWORD i, j, k, l;
    DWORD dConvention = IMG_CONVENTION_UNDEFINED;
    DWORD dStack      = -1;

    if (pbSymbol != NULL)
        {
        // skip common prefixes
        for (i = j = 0; apbPrefixes [i] != NULL; i++)
            {
            for (j = 0; apbPrefixes [i] [j]; j++)
                {
                if (apbPrefixes [i] [j] != pbSymbol [j]) break;
                }
            if (!apbPrefixes [i] [j]) break;
            j = 0;
            }
        // test for multiple '@'
        for (k = j, l = 0; (l < 2) && pbSymbol [k]; k++)
            {
            if (pbSymbol [k] == '@') l++;
            }
```

```
// don’t undecorate if multiple ‘@’, or C++ symbol
if ((l == 2) || (pbSymbol [0] == ‘?’))
    {
    j = 0;        // keep prefix
    k = MAXDWORD; // keep length
    }
else
    {
    // search for next ‘@’
    for (k = j; pbSymbol [k] && (pbSymbol [k] != ‘@’); k++);

    // read number of argument stack bytes if ‘@’ found
    if (pbSymbol [k] == ‘@’)
        {
        dStack = 0;

        for (l = k + 1; (pbSymbol [l] >= ‘0’) &&
                        (pbSymbol [l] <= ‘9’); l++)
            {
            dStack *= 10;
            dStack += pbSymbol [l] - ‘0’;
            }
        // don’t undecorate if non-numeric or empty trailer
        if (pbSymbol [l] || (l == k + 1))
            {
            dStack = -1;  // no stack size info

            j = 0;        // keep prefix
            k = MAXDWORD; // keep length
            }
        }
    }
// determine calling convention if single-char prefix
if (j == 1)
    {
    switch (pbSymbol [0])
        {
        case ‘@’:
            {
            dConvention = IMG_CONVENTION_FASTCALL;
            break;
            }
        case ‘_’:
            {
```

(continued)

```
                    dConvention = (dStack != -1
                                   ? IMG_CONVENTION_STDCALL
                                   : IMG_CONVENTION_CDECL);
                    break;
                    }
                }
            }
        // copy selected name portion
        k = min (k - j, sizeof (abBuffer) - 1);
        lstrcpynA (abBuffer, pbSymbol + j, k + 1);
        }
    if (pbBuffer != NULL)
        {
        lstrcpyA (pbBuffer, abBuffer);
        }
    if (pdConvention != NULL) *pdConvention = dConvention;
    return dStack;
    }
```

LISTING 1-30. *The* `imgSymbolUndecorate()` *API Function*

Note that the `imgTableResolve()` function ignores all symbols with undefined calling convention. This restriction safely excludes all import thunk, C++, and PCH symbols. Unfortunately, it also excludes some of the "good" symbols that don't have standard decorations. I don't think, however, that this is a big problem, because these symbols are not among those most frequently used.

The basic framework of a `w2k_img.dll` client application is outlined in Listing 1-31. The application first loads the symbol table from the `.dbg` file specified by the `ptPath` argument, using the `imgTableLoad()` API function. If the file contains an `NB10` CodeView subsection, the associated PDB file is loaded seamlessly. If the returned pointer is valid, the symbol entries can be enumerated in four ways, described by the comments inside the `for()` loop. Basically, the client can use the original order of the symbols as they appear in the `.dbg` or `.pdb` file, or it can choose one of the predefined sort indexes. When the symbol processing is finished, the application has to destroy the symbol table by calling `imgMemory Destroy()`. That's all! The application doesn't need any intimate knowledge about the internals of symbol files. All information it needs is stored in the `IMG_TABLE` and `IMG_ENTRY` structures set up by `imgTableLoad()`.

```
VOID WINAPI SymbolProcessor (PTBYTE ptPath)
    {
    PIMG_TABLE pit;
    PIMG_ENTRY pie;
    PVOID      pBase;
    DWORD      i;

    pBase = imgModuleBase (ptPath); // get current module load address

    if ((pit = imgTableLoad (ptPath, pBase)) != NULL)
        {
        for (i = j = 0; i < pit->dSymbols; i++)
            {
            // Option #1: default symbol order
            // pie = pit->aEntries + i;

            // Option #2: symbols sorted by address
            // pie = pit->piiAddress->apEntries [i];

            // Option #3: symbols sorted by name (case sensitive)
            // pie = pit->piiName->apEntries [i];

            // Option #4: symbols sorted by name (ignore case)
            // pie = pit->piiNameIC->apEntries [i];

            // Now, pie points to the IMG_ENTRY of the next symbol!
            // Do something useful with it ...

            }
        imgMemoryDestroy (pit);
        }
    return;
    }
```

LISTING 1-31. *Using the Symbol Table Management Functions*

A typical client application of `w2k_img.dll` will be presented in the next subsection. Note that I will return to this powerful utility DLL in Chapter 6, where it serves a rather unusual purpose: It looks up addresses of internal `ntoskrnl.exe` symbols that are neither documented nor exported, and a companion DLL uses this information to call into or read from these addresses. This trick sounds odd, but it works fine and can solve some tough programming and debugging problems. Stay tuned!

ANOTHER WINDOWS 2000 SYMBOL BROWSER

The sample application that shall demonstrate the usage of `w2k_img.dll` is an alternative version of the symbol browser presented in the previous section. It is named `w2k_sym2.exe`, but despite the name similarity, it is not just a rehash of `w2k_sym.exe`. The sample applications have quite different features and command options—just compare their command help screens, shown in Examples 1-5 and 1-11. The source code of `w2k_sym2.exe` is found on the CD accompanying this book in the `\src\w2k_sym2` directory tree.

Example 1-12 shows some sample output, generated by the command `w2k_sym2 +nu beep.sys`. The `+n` option selects sorting by name without consideration of the character case, and the `+u` option forces inclusion of symbols with unknown calling convention. The symbols with `CDECL` or `STDCALL` in the `ARGUMENTS` column refer to addresses of functions or global variables. The remaining rows in Example 1-12 are mostly import thunks into `ntoskrnl.exe` or `hal.dll`.

```
// w2k_sym2.exe
// SBS Windows 2000 Symbol Browser V1.00
// 08-27-2000 Sven B. Schreiber
// sbs@orgon.com

Usage: w2k_sym2 { [+-anNiprdxusz] [:<sections>] [/<symbols>] <path> }

       +   enable subsequent options
       -   disable subsequent option
       a   sort by address
       n   sort by name
       N   sort by name (case sensitive)
       i   ignore case in filter strings
       p   force preferred load address
       r   display relative addresses
       d   display decorated symbols
       x   display exported symbols only
       u   include symbols with unknown calling convention
       s   include special symbols
       z   include zero-address symbols

<sections> and <symbols> are filter expressions,
optionally containing the wildcards * and ?.
```

EXAMPLE 1-11. *The Command Help of* `w2k_sym2.exe`

```
// w2k_sym2.exe
// SBS Windows 2000 Symbol Browser V1.00
// 08-27-2000 Sven B. Schreiber
// sbs@orgon.com

Module name:    beep.sys
Time stamp:     Wednesday, 10-20-1999, 22:18:59
Base address:   0xF09CF000
Check sum:      0x0000C54F
Symbol file:    E:\WINNT\Symbols\sys\beep.dbg
Symbol table:   23520 bytes
Symbol filter:  *
Sections:       *

   # INDEX ADDRESS  SECTION     ARGUMENTS   X NAME
-----------------------------------------------------------------

   1     0 F09CF70C  2 .rdata                _allmul
   2     1 F09CF6B2  1 .text       CDECL     _allmul
   3     2 F09CF7B4  3 INIT        CDECL     _IMPORT_DESCRIPTOR_HAL
   4     3 F09CF7A0  3 INIT        CDECL     _IMPORT_DESCRIPTOR_ntoskrnl
   5     4 F09CF7C8  3 INIT        CDECL     _NULL_IMPORT_DESCRIPTOR
   6     5 F09CF34C  1 .text     8 STDCALL    BeepCancel
   7     6 F09CF39E  1 .text     8 STDCALL    BeepCleanup
   8     7 F09CF50E  1 .text     8 STDCALL    BeepClose
   9     8 F09CF456  1 .text     8 STDCALL    BeepDeviceControl
  10     9 F09CF4C0  1 .text     8 STDCALL    BeepOpen
  11    10 F09CF572  1 .text     8 STDCALL    BeepStartIo
  12    11 F09CF660  1 .text    10 STDCALL    BeepTimeOut
  13    12 F09CF67E  1 .text     4 STDCALL    BeepUnload
  14    13 F09CF29A  1 .text     8 STDCALL    DriverEntry
  15    14 F09CF6C0  2 .rdata    4           ExAcquireFastMutex
  16    15 F09CF6C4  2 .rdata    4           ExReleaseFastMutex
  17    16 F09CF6D4  2 .rdata                HAL_NULL_THUNK_DATA
  18    17 F09CF6D0  2 .rdata    4           HalMakeBeep
  19    18 F09CF724  2 .rdata    4           InterlockedDecrement
  20    19 F09CF6E0  2 .rdata    8           InterlockedExchange
  21    20 F09CF708  2 .rdata    4           InterlockedIncrement
  22    21 F09CF6E8  2 .rdata    4           IoAcquireCancelSpinLock
  23    22 F09CF714  2 .rdata   1C           IoCreateDevice
  24    23 F09CF710  2 .rdata    4           IoDeleteDevice
  25    24 F09CF6F4  2 .rdata    8           IofCompleteRequest
  26    25 F09CF6F8  2 .rdata    4           IoReleaseCancelSpinLock
  27    26 F09CF700  2 .rdata    8           IoStartNextPacket
  28    27 F09CF6EC  2 .rdata   10           IoStartPacket
  29    28 F09CF728  2 .rdata    4           KeCancelTimer
  30    29 F09CF718  2 .rdata    C           KeInitializeDpc
```

(continued)

```
  31    30 F09CF720  2 .rdata    C           KeInitializeEvent
  32    31 F09CF71C  2 .rdata    4           KeInitializeTimer
  33    32 F09CF6E4  2 .rdata    4           KeRemoveDeviceQueue
  34    33 F09CF6DC  2 .rdata    8           KeRemoveEntryDeviceQueue
  35    34 F09CF704  2 .rdata   10           KeSetTimer
  36    35 F09CF6CC  2 .rdata    4           KfLowerIrql
  37    36 F09CF6C8  2 .rdata    4           KfRaiseIrql
  38    37 F09CF6F0  2 .rdata    4           MmLockPagableDataSection
  39    38 F09CF6FC  2 .rdata    4           MmUnlockPagableImageSection
  40    39 F09CF72C  2 .rdata                ntoskrnl_NULL_THUNK_DATA
  41    40 F09CF6D8  2 .rdata    8           RtlInitUnicodeString
---------------------------------------
  13 non-NULL symbols
   0 exported symbols
```

EXAMPLE 1-12. *Sample Output of* w2k_sym2.exe

Note that the `_allmul` symbol appears twice in the list. The first one is an import thunk for the `_allmul()` function exported by `ntoskrnl.exe;` the other one is a simple function call forwarder that jumps through this thunk. If you add the `+d` switch to the command to view the symbols with full decoration, you can see that the `_allmul` import thunk is really called `__imp___allmul,` whereas the original name of the forwarder is `__allmul.` Obviously, those decorations *do* serve some useful purpose, even though they are sometimes quite distracting.

This chapter has presented extensive information. Maybe you didn't expect that there is so much to say about Windows 2000 debuggers, debugging APIs, and symbol files. Most Windows programming books don't dedicate much space to this kind of information. However, I believe that this essential background knowledge will help you in writing your own debugging utilities.