CHAPTER 3

Writing Kernel-Mode Drivers

In the next chapters, we will frequently have to access system resources that are available in kernel-mode only. Large portions of the sample code are designed as kernel-mode driver routines. Therefore, some basic knowledge about the development of this type of software is required. Because I cannot assume that all readers already have this expertise, I will insert here a short introduction to kernel-mode programming that focuses on the usage of a driver wizard found on the accompanying CD.

This chapter also discusses the basics of the Windows 2000 Service Control Manager that allow loading, controlling, and unloading drivers at runtime, resulting in wonderfully short change-build-test turnaround cycles. The title of this chapter might be a bit misleading—the word *driver* is usually associated with low-level software that controls some piece of hardware. In fact, many kernel-mode programmers do just that all day long. However, the layered driver model of Windows 2000 allows much more than this. Kernel-mode drivers can do arbitrary complex tasks and might even act like high-level user-mode DLLs, except that they are running on a higher CPU privilege level and use a different programming interface. In this book, the driver paradigm will not be applied to any hardware. Instead, we will use this powerful programming technique to spy on Windows 2000 internals, using kernel-mode drivers as a shuttle to fly from the small world of user-mode to the outer space of the Windows 2000 kernel.

CREATING A DRIVER SKELETON

Even developers who have been writing Win32 applications or libraries for a long time tend to feel like absolute beginners as soon as they have to write their first kernel-mode driver. The reason for this is that kernel-mode code runs in a completely

different operating system environment. A Win32 programmer works exclusively with a set of system components that belong to a subsystem of Windows 2000, named Win32. Other programmers might prefer to write POSIX or OS/2 applications, which are also supported by Windows 2000 by means of additional subsystems. Thanks to its subsystem concept, Windows 2000 acts like a chameleon—it can emulate various operating systems by exposing their application interfaces in the form of subsystems. Contrary to this, kernel-mode modules are located somewhere below this layer, using a more basic operating system interface. Because there are no more subsystems on this system level, kernel-mode code can "see" the real Windows 2000 operating system. The interface they are talking to is the "final frontier." Of course, it is not absolutely correct that the kernel-mode zone is free of subsystems. In Chapter 2, we saw that the `win32k.sys` module is a kernel-mode branch of the Win32 GUI and Window Manager, installed there for performance reasons. However, only a small part of the API functions exposed by `win32k.sys` reappear in `gdi32.dll` and `user32.dll` as Win32 API functions, so Win32K is more than just a Win32 foot on kernel-mode soil. It could be regarded as a high-performance display engine kernel as well.

The Windows 2000 Device Driver Kit

Because kernel-mode programming works on a different system interface, the usual header and import library files used in Win32 programming aren't of use here. For Win32 development, Microsoft provides the Platform Software Development Kit (SDK). For kernel-mode drivers, the Windows 2000 Device Driver Kit (DDK) is required. Along with documentation, the DDK provides special header files and import libraries needed to interface the Windows 2000 kernel modules. After installing the DDK, your next step should be to open Microsoft Visual C to add the DDK file paths to the directory lists of the compiler and linker. From the main menu, select **Tools** and **Options...,** then click on the **Directories** tab. From the **Show directories for:** drop-down list, select **Include files** and add the appropriate DDK path to the list, as shown in Figure 3-1. By default, the DDK is installed into a base directory named `\NTDDK,` and the included files are located in the `\NTDDK\inc` subdirectory. After entering the path, use the **up arrow** to move it to the position of your choice—preferably on Top Two right after the Platform SDK. Always keep the original Microsoft Visual Studio files at the end of the list, because many of them are superseded by more recent SDK and DDK files.

After adding the base directory of the DDK header files, do the same for the import libraries. The DDK comes with two sets of files, one for free (release) builds, and another one for checked (debug) builds. The corresponding subdirectories are `\NTDDK\libfre\i386` and `\NTDDK\libchk\i386,` respectively. Figure 3-2

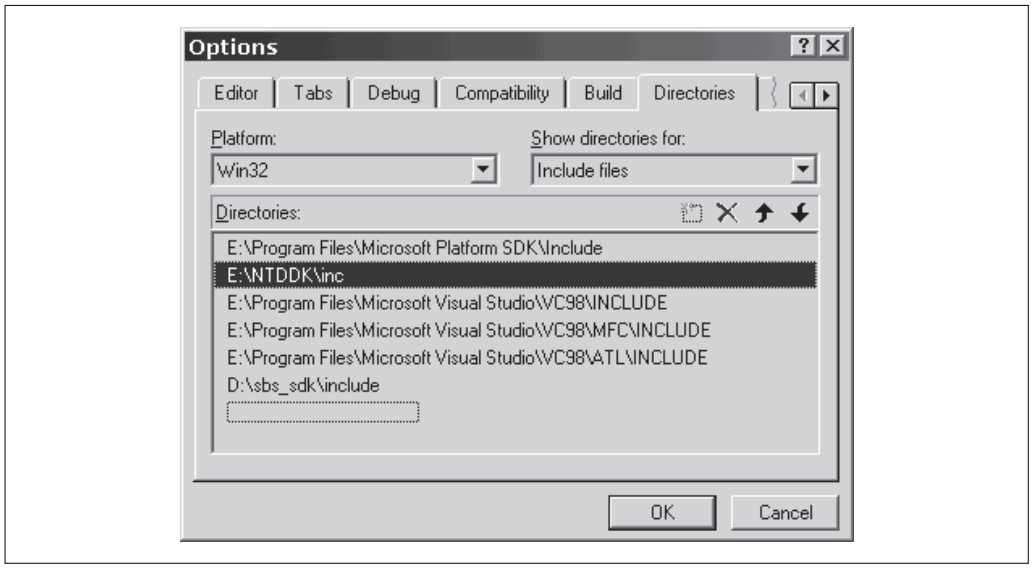

FIGURE 3-1. *Adding the DDK Header File Path*

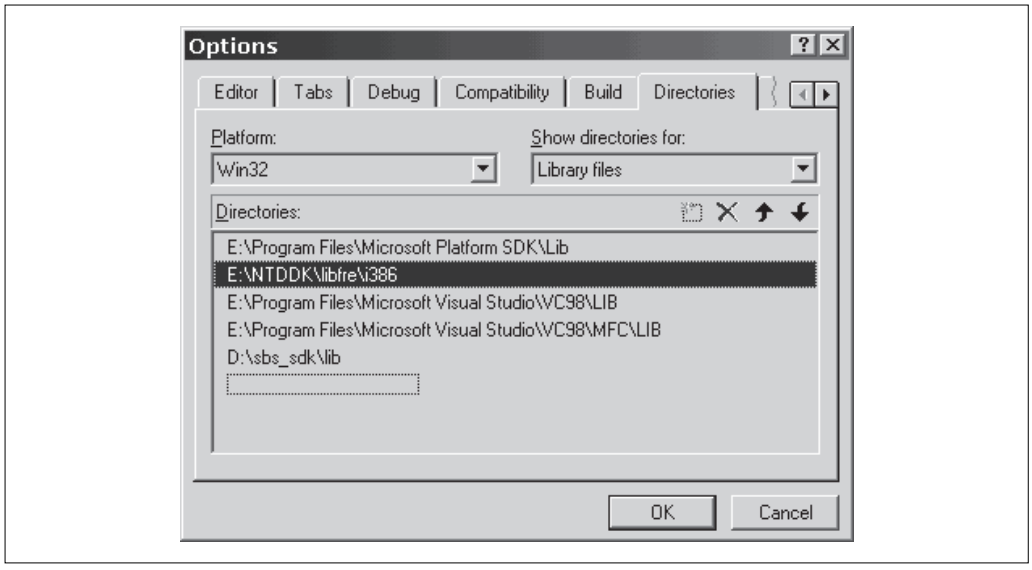

FIGURE 3-2. *Adding the DDK Import Library Path*

shows an example. To enter the path, select **Library files** from the **Show directories for:** list first. After you are done, move this entry to an appropriate position using the **up arrow.**

The programming environment of the DDK differs somewhat from the Win32 model. The following list points out some of the most obvious differences:

- For Win32 programs, the main header file that has to be included is `windows.h`. In kernel-mode driver code, this file is not applicable. It is replaced by `ntddk.h`.
- The main entry point function is called `DriverEntry()`, not `WinMain()` or `main()`. Its prototype is shown in Listing 3-1.
- Be aware that some of the common Win32 data types, such as `BYTE`, `WORD`, and `DWORD`, are not available. The DDK prefers `UCHAR`, `USHORT`, `ULONG`, and the like. However, it is easy to define your favorite types, as done exemplary in Listing 3-2.

Three important differences between the Windows NT 4.0 and Windows 2000 versions of the DDK should be noted as well:

- Although the base directory of the Windows NT 4.0 DDK is called `\DDK` by default, the Windows 2000 DDK now uses the default name `\NTDDK`.
- In the Windows NT 4.0 DDK, the main header file `ntddk.h` resides in the base directory. In the Windows 2000 DDK, this file has moved to the subdirectory `ddk` of the base directory.
- The paths of the import library files have changed as well: `lib\i386\free` has become `libfre\i386`, and `lib\i386\checked` has been replaced by `libchk\i386`.

I am not sure whether this reshuffling and renaming was really necessary, but we do have to live with it now.

```
NTSTATUS DriverEntry (PDRIVER_OBJECT  pDriverObject,
                      PUNICODE_STRING pusRegistryPath);
```

LISTING 3-1. *Prototype of the* `DriverEntry()` *Function*

```
typedef UCHAR  BYTE,  *PBYTE;
typedef USHORT WORD,  *PWORD;
typedef ULONG  DWORD, *PDWORD;
```

LISTING 3-2. *Defining Common Win32 Data Types*

A CUSTOMIZABLE DRIVER WIZARD

The main problem with kernel-mode drivers is that Visual C/C++ doesn't provide a wizard for projects of this kind. None of the various project types offered by the **File/New** dialog is suited for drivers. Fortunately, the Microsoft Developer Network (MSDN) Library contains a series of great articles about Windows NT kernel-mode driver development, written between 1994 and 1995 by Ruediger R. Asche. Two of them give detailed instructions on how to add a custom driver wizard to Visual C/C++, with sample code and application notes (Asche 1995a, 1995b). These articles have been of immense help to me, and although the output of the original wizard didn't fit all my needs, it was an ideal starting point. The kernel-mode driver wizard I will present now is based on output files generated by Ruediger Asche's original wizard.

My driver wizard is included with full source code on the companion CD of this book in the directory tree `\src\w2k_wiz`. By reading the source files, you will find that its real title is "SBS Windows 2000 Code Wizard." In fact, this is a general-purpose Windows 2000 program skeleton generator that can produce several program types, including Win32 DLLs and applications. However, the configuration files on the CD are tailored to kernel-mode driver development. Essentially, my wizard is a file converter that reads in a set of files, converts them by applying some simple rules, and writes the results back to another set of files. The input files are templates, and the output files are C project files. By modifying the templates, the driver wizard can be turned into a DLL wizard, and so on. Up to seven templates can be supplied (if one is missing, a noncritical error is reported):

- Files with the extension `.tw` are workspace templates and will be saved as Microsoft Developer Studio Workspace Files with extension `.dsw`. You probably know this file type from the **File/Open Workspace...** menu command of Visual C/C++, which requests a `.dsw` file to be specified.
- Files with the extension `.tp` are project templates and will be saved as Microsoft Developer Studio Project Files with extension `.dsp`. Project files are referenced by the associated workspace files and contain all build settings of the project for all configurations (e.g., **Release** and **Debug**).

- Files with the extensions `.tc`, `.th`, `.tr`, and `.td` are C source files and will become files of type `.c`, `.h`, `.rc`, and `.def`. I am sure that everyone knows the purpose of these files.
- Files with the extension `.ti` are icon files and are saved unchanged with extension `.ico`. This template is just a dummy icon included with the wizard to prevent the resource compiler from reporting an error. You should edit or replace it by your own creation after running the wizard.

This seven-piece set of files is the minimum requirement of a new project. The `.def` file is a somewhat old-fashioned way of exporting API functions from a DLL, but I like it more than the `__declspec(dllexport)` method. Because drivers usually don't export functions, I have omitted the `.td` template, which results in a benign error reported by the wizard. I also could have omitted the resource script and the icon, but experience shows that both are nice to have. Moreover, the default `.rc` file output by the wizard contains a full-featured personalized version resource, constructed from your individual configuration settings. The applied conversion rules are simple, consisting of a short list of string substitutions. While scanning a template file, the converter looks for escape sequences consisting of character pairs in which the first one is a percent sign. If it detects one, it decides which action to take by evaluating the second character. Table 3-1 lists the recognized escape sequences.

TABLE 3-1. *The Wizard's String Substitution Rules*

INPUT	OUTPUT
%n	Project name (original notation)
%N	Project name (uppercase notation)
%s	Fully qualified path of the *w2k_wiz.ini* file
%d	Current day (always two digits)
%m	Current month (always two digits)
%y	Current year (always four digits)
%t	Default project description, as defined in *w2k_wiz.ini*
%c	Author's company name, as defined in *w2k_wiz.ini*
%a	Author's name, as defined in *w2k_wiz.ini*
%e	Author's email address, as defined in *w2k_wiz.ini*
%p	Default ProgID prefix, as defined in *w2k_wiz.ini*
%i	DDK header file path, as defined in *w2k_wiz.ini*
%l	DDK import library path (release configuration), as defined in *w2k_wiz.ini*
%L	DDK import library path (debug configuration), as defined in *w2k_wiz.ini*
%%	% (escapement for a single percent character)
%<other>	Copied unchanged to the output file

Table 3-1 contains several references to the configuration file `w2k_wiz.ini`. Its default contents are shown in Example 3-1. Before using the wizard, you should copy `w2k_wiz.exe, w2k_wiz.ini,` and all `w2k_wiz.t*` template files from the CD's `\src\w2k_wiz\release` directory to your hard disk and edit the configuration file, replacing the values in angular brackets with your personal settings. You should also set the `Include, Free,` and `Checked` values to match your DDK setup configuration. If you are using Visual C/C++ Version 6.0, the `Root` entry can remain unchanged. If not, set its value to the registry key where the base directory of your projects is stored. If this value ends with a backslash, it is interpreted as the default value of the specified registry key. Otherwise, the token following the last backslash should denote a named value of the key specified by the remaining character sequence. In Example 3-1, the key is `HKEY_CURRENT_USER\Software\Microsoft\DevStudio\6.0\Directories,` and its value `WorkspaceDir` stores the basic workspace directory.

Invocation of the wizard is: Just type `w2k_wiz MyDriver,` and it will generate a project folder named `MyDriver` in the current directory, containing the files `MyDriver.dsw, MyDriver.dsp, MyDriver.c, MyDriver.h, MyDriver.rc,` and `MyDriver.ico`. If you specify the project name with a preceding path, the project folder will be created at the specified location. Another legal command option is the asterisk, such as in `w2k_wiz *MyDriver`. In this case, the wizard will not create the project folder in the current directory, but queries the registry for the default base directory maintained by Visual C/C++, using the `Root` entry in `w2k_wiz.ini`. This is probably the most convenient command variant and is the one I usually use.

```
; w2k_wiz.ini
; 08-27-2000 Sven B. Schreiber
; sbs@orgon.com

[Settings]
Text    = <SBS Windows 2000 Code Wizard Project>
Company = <MyCompany>
Author  = <MyName>
Email   = <my@email>
Prefix  = <MyPrefix>
Include = E:\NTDDK\inc
Free    = E:\NTDDK\libfre\i386
Checked = E:\NTDDK\libchk\i386
Root    = HKEY_CURRENT_USER\Software\Microsoft\DevStudio\6.0\Directories\WorkspaceDir
```

EXAMPLE 3-1. *Personal Settings Supported by the Wizard*

The wizard always looks for its configuration and template files in the directory of the executable. Therefore, you can keep several copies of the wizard with different settings on your disk, provided that they reside in individual directories or have different base names. The files on the CD are preset for simple kernel-mode driver projects. You can customize all files to fit your needs, keeping separate copies for drivers, Win32 applications, DLLs, or whatever type of Windows 2000 code you write.

Running the Driver Wizard

Now it is time to try the driver wizard. The example below resulted from the command `w2k_wiz *TestDrv` entered at a Windows 2000 console prompt. This should create a project named `TestDrv` in the default workspace folder of Visual C/C++. Example 3-2 shows the status messages displayed by the program on the screen while it is converting files.

```
D:\>w2k_wiz *TestDrv

// w2k_wiz.exe
// SBS Windows 2000 Code Wizard V1.00
// 08-27-2000 Sven B. Schreiber
// sbs@orgon.com

Project D:\Program Files\DevStudio\MyProjects\TestDrv\

Loading D:\etc32\w2k_wiz.tc ... OK
Writing D:\Program Files\DevStudio\MyProjects\TestDrv\TestDrv.c ... OK

Loading D:\etc32\w2k_wiz.td ... ERROR

Loading D:\etc32\w2k_wiz.th ... OK
Writing D:\Program Files\DevStudio\MyProjects\TestDrv\TestDrv.h ... OK

Loading D:\etc32\w2k_wiz.ti ... OK
Writing D:\Program Files\DevStudio\MyProjects\TestDrv\TestDrv.ico ... OK

Loading D:\etc32\w2k_wiz.tp ... OK
Writing D:\Program Files\DevStudio\MyProjects\TestDrv\TestDrv.dsp ... OK

Loading D:\etc32\w2k_wiz.tr ... OK
Writing D:\Program Files\DevStudio\MyProjects\TestDrv\TestDrv.rc ... OK

Loading D:\etc32\w2k_wiz.tw ... OK
Writing D:\Program Files\DevStudio\MyProjects\TestDrv\TestDrv.dsw ... OK
```

Example 3-2. *Running the Windows 2000 Code Wizard*

Obviously, all operations were completed without error except for the `.td` to `.def` conversion, which is a benign error condition. The driver skeleton produced by the wizard doesn't require a `.def` file, so there is no need for a `.td` template. Now it should be possible to open the new workspace in Visual C/C++, using the **File/Open Workspace...** menu command. Indeed, there is a new folder named `TestDrv`, and it contains a workspace file named `TestDrv.dsw` that can be opened without problem. Next, you should select the active configuration for your builds. The `.dsp` file generated by the driver wizard defines the following two configurations:

1. Win2K kernel-mode driver (debug)
2. Win2K kernel-mode driver (release)

By default, the debug configuration is selected, but you can switch configurations at any time by choosing **Build/Set Active Configuration...** from the Visual C/C++ menu. Next, you should copy the file `\src\common\include\DrvInfo.h` from the CD to one of your header file directories, and open the `TestDrv.c`, `TestDrv.h`, and `TestDrv.rc` files for editing. When opening `TestDrv.rc`, be sure to open it as a text file (Figure 3-3), because it uses complex macros from `DrvInfo.h` that cause the resource editor to die with an exception. This nasty problem was introduced with Visual C/C++ 5.0, as far as I remember, and has not yet been fixed. Contrary to the editor, the resource compiler doesn't have problems with complex resource macros.

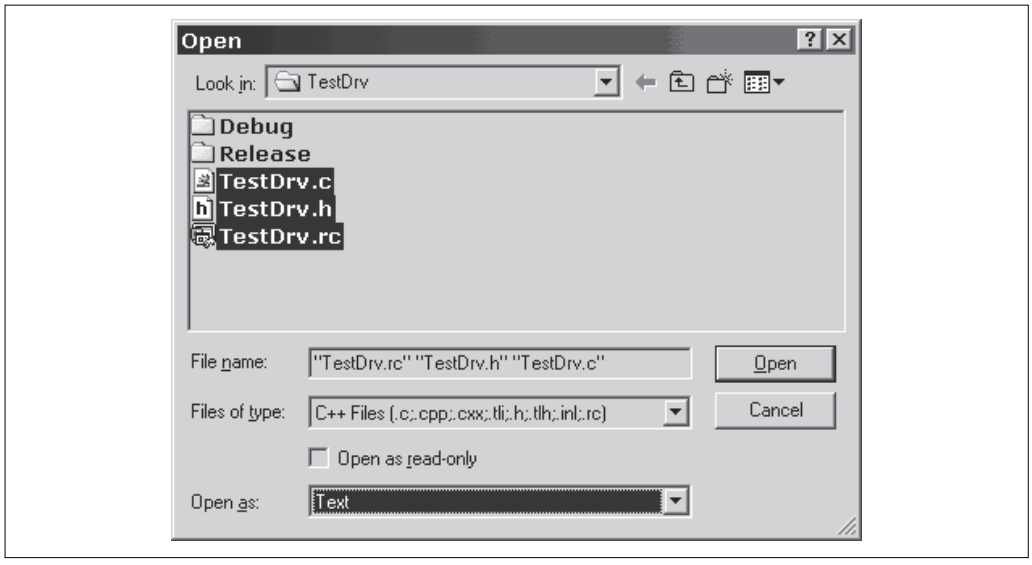

FIGURE 3-3. *Opening the Driver Source Files in Text Mode*

Now everything should be set up for the first build. In Example 3-3, I have attempted to build a release version of the new driver by selecting **Build/Rebuild All** from the Visual C/C++ menu, and it seems that everything works fine. By the way, the ellipses ending the first two lines of the build command output indicate that I have truncated them.

The linker creates an executable file named `TestDrv.sys` in the `Debug` or `Release` subdirectory of the project folder, depending on the chosen build configuration. The release version of the test driver is 5.5 KB in size, and the debug version is 8 KB. You can use the Multi-Format Visual Disassembler (MFVDasm) or the PE and COFF File Viewer (PEview) on the companion CD to verify that the resulting `TestDrv.sys` file contains valid code and data.

```
Deleting intermediate files and output files for project 'TestDrv - Win2K ...
-------- Configuration: TestDrv - Win2K kernel-mode driver (release) ...
Compiling resources...
Compiling...
TestDrv.c
Linking...

TestDrv.sys - 0 error(s), 0 warning(s)
```

EXAMPLE 3-3. *Building the Release Version of the Test Driver*

INSIDE THE DRIVER SKELETON

Listing 3-3 shows the `TestDrv.c` file emitted by the wizard. The associated header file `TestDrv.h` is shown in Listing 3-4. In Listing 3-3, please note the `<MyName>` and `<MyCompany>` tags in the heading and in the fourth line of the disclaimer. If the `Author` and `Company` entries in `w2k_wiz.ini` are set appropriately, your own name and company strings will go here. Also note that the current date appears in the heading, as well as in the revision history. (Listing 3-3 was generated on August 27, 2000, so the date is correct.) More values from the wizard's configuration file are found in the `PROGRAM IDENTIFICATION` section of Listing 3-4.

```
// TestDrv.c
// 08-27-2000 <MyName>
// Copyright © 2000 <MyCompany>

#define  _TESTDRV_SYS_
#include <ddk\ntddk.h>
#include "TestDrv.h"

// ================================================================
// DISCLAIMER
// ================================================================

/*

This software is provided "as is" and any express or implied
warranties, including, but not limited to, the implied warranties of
merchantability and fitness for a particular purpose are disclaimed.
In no event shall the author <MyName> be liable for any direct,
indirect, incidental, special, exemplary, or consequential
damages (including, but not limited to, procurement of substitute
goods or services; loss of use, data, or profits; or business
interruption) however caused and on any theory of liability, whether
in contract, strict liability, or tort (including negligence
or otherwise) arising in any way out of the use of this software,
even if advised of the possibility of such damage.

*/

// ================================================================
// REVISION HISTORY
// ================================================================

/*

// 08-27-2000 V1.00 Original version.

*/

// ================================================================
// GLOBAL DATA
// ================================================================

PRESET_UNICODE_STRING (usDeviceName,       CSTRING (DRV_DEVICE));
PRESET_UNICODE_STRING (usSymbolicLinkName, CSTRING (DRV_LINK  ));

PDEVICE_OBJECT  gpDeviceObject  = NULL;
PDEVICE_CONTEXT gpDeviceContext = NULL;

// ================================================================
// DISCARDABLE FUNCTIONS
// ================================================================
```

(continued)

```
NTSTATUS DriverInitialize (PDRIVER_OBJECT  pDriverObject,
                           PUNICODE_STRING pusRegistryPath);

NTSTATUS DriverEntry      (PDRIVER_OBJECT  pDriverObject,
                           PUNICODE_STRING pusRegistryPath);

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

#ifdef ALLOC_PRAGMA

#pragma alloc_text (INIT, DriverInitialize)
#pragma alloc_text (INIT, DriverEntry)

#endif

// ==================================================================
// DEVICE REQUEST HANDLER
// ==================================================================

NTSTATUS DeviceDispatcher (PDEVICE_CONTEXT pDeviceContext,
                           PIRP            pIrp)
    {
    PIO_STACK_LOCATION pisl;
    DWORD              dInfo = 0;
    NTSTATUS           ns    = STATUS_NOT_IMPLEMENTED;

    pisl = IoGetCurrentIrpStackLocation (pIrp);

    switch (pisl->MajorFunction)
        {
        case IRP_MJ_CREATE:
        case IRP_MJ_CLEANUP:
        case IRP_MJ_CLOSE:
            {
            ns = STATUS_SUCCESS;
            break;
            }
        }
    pIrp->IoStatus.Status      = ns;
    pIrp->IoStatus.Information = dInfo;

 IoCompleteRequest (pIrp, IO_NO_INCREMENT);
 return ns;
    }

// ==================================================================
// DRIVER REQUEST HANDLER
// ==================================================================

NTSTATUS DriverDispatcher (PDEVICE_OBJECT pDeviceObject,
                           PIRP           pIrp)
    {
```

```
    return (pDeviceObject == gpDeviceObject
            ? DeviceDispatcher (gpDeviceContext, pIrp)
            : STATUS_INVALID_PARAMETER_1);
    }

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

void DriverUnload (PDRIVER_OBJECT pDriverObject)
    {
    IoDeleteSymbolicLink (&usSymbolicLinkName);
    IoDeleteDevice       (gpDeviceObject);
    return;
    }

// ==================================================================
// DRIVER INITIALIZATION
// ==================================================================

NTSTATUS DriverInitialize (PDRIVER_OBJECT  pDriverObject,
                           PUNICODE_STRING pusRegistryPath)
    {
    PDEVICE_OBJECT pDeviceObject = NULL;
    NTSTATUS       ns = STATUS_DEVICE_CONFIGURATION_ERROR;

    if ((ns = IoCreateDevice (pDriverObject, DEVICE_CONTEXT_,
                              &usDeviceName, FILE_DEVICE_CUSTOM,
                              0, FALSE, &pDeviceObject))
        == STATUS_SUCCESS)
        {
        if ((ns = IoCreateSymbolicLink (&usSymbolicLinkName,
                                        &usDeviceName))
            == STATUS_SUCCESS)
            {
            gpDeviceObject  = pDeviceObject;
            gpDeviceContext = pDeviceObject->DeviceExtension;

            gpDeviceContext->pDriverObject = pDriverObject;
            gpDeviceContext->pDeviceObject = pDeviceObject;
            }
        else
            {
            IoDeleteDevice (pDeviceObject);
            }
        }
    return ns;
    }

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

NTSTATUS DriverEntry (PDRIVER_OBJECT  pDriverObject,
                      PUNICODE_STRING pusRegistryPath)
    {
```

(continued)

```
    PDRIVER_DISPATCH *ppdd;
    NTSTATUS          ns = STATUS_DEVICE_CONFIGURATION_ERROR;

    if ((ns = DriverInitialize (pDriverObject, pusRegistryPath))
        == STATUS_SUCCESS)
        {
        ppdd = pDriverObject->MajorFunction;

        ppdd [IRP_MJ_CREATE                   ] =
        ppdd [IRP_MJ_CREATE_NAMED_PIPE        ] =
        ppdd [IRP_MJ_CLOSE                    ] =
        ppdd [IRP_MJ_READ                     ] =
        ppdd [IRP_MJ_WRITE                    ] =
        ppdd [IRP_MJ_QUERY_INFORMATION        ] =
        ppdd [IRP_MJ_SET_INFORMATION          ] =
        ppdd [IRP_MJ_QUERY_EA                 ] =
        ppdd [IRP_MJ_SET_EA                   ] =
        ppdd [IRP_MJ_FLUSH_BUFFERS            ] =
        ppdd [IRP_MJ_QUERY_VOLUME_INFORMATION] =
        ppdd [IRP_MJ_SET_VOLUME_INFORMATION   ] =
        ppdd [IRP_MJ_DIRECTORY_CONTROL        ] =
        ppdd [IRP_MJ_FILE_SYSTEM_CONTROL      ] =
        ppdd [IRP_MJ_DEVICE_CONTROL           ] =
        ppdd [IRP_MJ_INTERNAL_DEVICE_CONTROL  ] =
        ppdd [IRP_MJ_SHUTDOWN                 ] =
        ppdd [IRP_MJ_LOCK_CONTROL             ] =
        ppdd [IRP_MJ_CLEANUP                  ] =
        ppdd [IRP_MJ_CREATE_MAILSLOT          ] =
        ppdd [IRP_MJ_QUERY_SECURITY           ] =
        ppdd [IRP_MJ_SET_SECURITY             ] =
        ppdd [IRP_MJ_POWER                    ] =
        ppdd [IRP_MJ_SYSTEM_CONTROL           ] =
        ppdd [IRP_MJ_DEVICE_CHANGE            ] =
        ppdd [IRP_MJ_QUERY_QUOTA              ] =
        ppdd [IRP_MJ_SET_QUOTA                ] =
        ppdd [IRP_MJ_PNP                      ] = DriverDispatcher;
        pDriverObject->DriverUnload             = DriverUnload;
        }
    return ns;
    }

// ====================================================================
// END OF PROGRAM
// ====================================================================
```

LISTING 3-3. *The Source Code of the Driver Skeleton*

The C code of the driver skeleton in Listings 3-3 and 3-4 contains some common boilerplate code that is shared by all kernel-mode drivers I have written so far. I have designed the wizard to be as customizable as possible. Feel free to change the

```
// TestDrv.h
// 08-27-2000 <MyName>
// Copyright © 2000 <MyCompany>

// ====================================================================
// PROGRAM IDENTIFICATION
// ====================================================================

#define DRV_BUILD           1
#define DRV_VERSION_HIGH    1
#define DRV_VERSION_LOW     0

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

#define DRV_DAY             27
#define DRV_MONTH           08
#define DRV_YEAR            2000

// --------------------------------------------------------------------
// Customize these settings by editing the configuration file
// D:\etc32\w2k_wiz.ini

#define DRV_MODULE          TestDrv
#define DRV_NAME            <SBS Windows 2000 Code Wizard Project>
#define DRV_COMPANY         <MyCompany>
#define DRV_AUTHOR          <MyName>
#define DRV_EMAIL           <my@email>
#define DRV_PREFIX          <MyPrefix>

// ====================================================================
// HEADER FILES
// ====================================================================

#include <drvinfo.h>        // defines more DRV_* items

//////////////////////////////////////////////////////////////////////
#ifndef _RC_PASS_
//////////////////////////////////////////////////////////////////////

// ====================================================================
// CONSTANTS
// ====================================================================

#define FILE_DEVICE_CUSTOM      0x8000

// ====================================================================
// STRUCTURES
// ====================================================================

typedef struct _DEVICE_CONTEXT
    {
```

(continued)

```
    PDRIVER_OBJECT pDriverObject;
    PDEVICE_OBJECT pDeviceObject;
    }
    DEVICE_CONTEXT, *PDEVICE_CONTEXT, **PPDEVICE_CONTEXT;

#define DEVICE_CONTEXT_ sizeof (DEVICE_CONTEXT)

//////////////////////////////////////////////////////////////////
#endif // #ifndef _RC_PASS_
//////////////////////////////////////////////////////////////////

// ==============================================================
// END OF FILE
// ==============================================================
```

LISTING 3-4. *The Header File of the Driver Skeleton*

wizard's templates. For those who want to keep the code for now, the following section is a short description of its internals.

The main entry point of the driver module is `DriverEntry()`. Like all Windows 2000 module entry points, this name is not a requirement. You can choose any symbol you like, but you must tell the linker the name of the entry point by adding the `/entry` switch to its command line. For this test driver, the wizard has already taken care of this task. Inside the `w2k_wiz.tp` template or the resulting `TestDrv.dsp` file, you will find two occurrences of the string `/entry:"DriverEntry@8"` in the linker command line, one for each build configuration. The `@8` suffix indicates that `DriverEntry()` receives eight argument bytes on the stack, which is in perfect congruence with its prototype definition in Listing 3-1: two pointer arguments, each of them 32 bits wide, yield 64 bits, or 8 bytes.

The first thing `DriverEntry()` does is call `DriverInitialize()`, which will create a device object and a symbolic link that you will probably need later to communicate with the device from user-mode applications. It is a bit difficult to find out which names are used in the `IoCreateDevice()` and `IoCreateSymbolicLink()` calls, because they are constructed by macros defined in the common header file `DrvInfo.h`, found in the `\src\common\include` directory of the CD. This file is a header file that compiles various sorts of program information from a couple of basic customizable strings. If you want to know more about this trick, go to the `PROGRAM IDENTIFICATION` section in `TestDrv.h` (see top of Listing 3-4) and trace the `DRV_*` definitions as they are grouped in various ways inside `DrvInfo.h`. For example, a full-fledged `VERSIONINFO` resource is constructed from several pieces. Among other things, the constants `DRV_DEVICE` and `DRV_LINK` are defined, which evaluate to

\Device\TestDrv and \DosDevices\TestDrv here, respectively. Note that many kernel API functions, such as IoCreateDevice() and IoCreateSymbolicLink(), don't accept strings as plain zero-terminated character sequences, but rather expect them to be packed into a special UNICODE_STRING structure, introduced in Chapter 2 and repeated in Listing 3-5. The macro PRESET_UNICODE_STRING, defined in DrvInfo.h and applied in the GLOBAL DATA section of TestDrv.c in Listing 3-3, creates a static UNICODE_STRING structure from a simple Unicode string literal. This is a convenient shorthand notation for the definition of UNICODE_STRINGs that remain unchanged throughout the lifetime of a program instance.

After successfully creating the device object and its symbolic link, DriverInitialize() stores pointers to the device object and the device context in static global variables. The device context is a private structure of the device that can have arbitrary size and shape. The driver skeleton attaches a simple DEVICE_CONTEXT structure, defined in TestDrv.h, to its device. This structure contains nothing but pointers to the device and driver objects. You can extend this structure if you need persistent device-specific storage for any private data of your driver. The device context will be supplied by the system with every I/O Request Packet (IRP) the driver receives.

After DriverInitialize() returns and reports success, DriverEntry() sets up an important array, passed in by the system as part of the driver object structure pDriverObject. This array contains slots for all IRPs the driver can expect, and DriverEntry() has to write callback function pointers to the slots of all request types it wishes to handle. The driver skeleton defers this decision and saves a single DriverDispatcher() pointer to all 28 available slots, listed in Table 3-2. Later on, DriverDispatcher() will decide which IRP types are of interest, returning STATUS_NOT_IMPLEMENTED for all unhandled IRPs. Note that there are subtle differences between the Windows NT 4.0 and Windows 2000 layouts of the IRP handler array. In Table 3-2, the differing slots are marked boldface.

```
typedef struct _UNICODE_STRING
    {
    WORD  Length;
    WORD  MaximumLength;
    PWORD Buffer;
    }
    UNICODE_STRING, *PUNICODE_STRING;
```

LISTING 3-5. *An Ubiquitous Windows 2000 Structure:* UNICODE_STRING

TABLE 3-2. *I/O Request Packet Slots Compared*

SLOT	WINDOWS NT 4.0	WINDOWS 2000
0x00	IRP_MJ_CREATE	IRP_MJ_CREATE
0x01	IRP_MJ_CREATE_NAMED_PIPE	IRP_MJ_CREATE_NAMED_PIPE
0x02	IRP_MJ_CLOSE	IRP_MJ_CLOSE
0x03	IRP_MJ_READ	IRP_MJ_READ
0x04	IRP_MJ_WRITE	IRP_MJ_WRITE
0x05	IRP_MJ_QUERY_INFORMATION	IRP_MJ_QUERY_INFORMATION
0x06	IRP_MJ_SET_INFORMATION	IRP_MJ_SET_INFORMATION
0x07	IRP_MJ_QUERY_EA	IRP_MJ_QUERY_EA
0x08	IRP_MJ_SET_EA	IRP_MJ_SET_EA
0x09	IRP_MJ_FLUSH_BUFFERS	IRP_MJ_FLUSH_BUFFERS
0x0A	IRP_MJ_QUERY_VOLUME_INFORMATION	IRP_MJ_QUERY_VOLUME_INFORMATION
0x0B	IRP_MJ_SET_VOLUME_INFORMATION	IRP_MJ_SET_VOLUME_INFORMATION
0x0C	IRP_MJ_DIRECTORY_CONTROL	IRP_MJ_DIRECTORY_CONTROL
0x0D	IRP_MJ_FILE_SYSTEM_CONTROL	IRP_MJ_FILE_SYSTEM_CONTROL
0x0E	IRP_MJ_DEVICE_CONTROL	IRP_MJ_DEVICE_CONTROL
0x0F	IRP_MJ_INTERNAL_DEVICE_CONTROL	IRP_MJ_INTERNAL_DEVICE_CONTROL
0x10	IRP_MJ_SHUTDOWN	IRP_MJ_SHUTDOWN
0x11	IRP_MJ_LOCK_CONTROL	IRP_MJ_LOCK_CONTROL
0x12	IRP_MJ_CLEANUP	IRP_MJ_CLEANUP
0x13	IRP_MJ_CREATE_MAILSLOT	IRP_MJ_CREATE_MAILSLOT
0x14	IRP_MJ_QUERY_SECURITY	IRP_MJ_QUERY_SECURITY
0x15	IRP_MJ_SET_SECURITY	IRP_MJ_SET_SECURITY
0x16	**IRP_MJ_QUERY_POWER**	**IRP_MJ_POWER**
0x17	**IRP_MJ_SET_POWER**	**IRP_MJ_SYSTEM_CONTROL**
0x18	IRP_MJ_DEVICE_CHANGE	IRP_MJ_DEVICE_CHANGE
0x19	IRP_MJ_QUERY_QUOTA	IRP_MJ_QUERY_QUOTA
0x1A	IRP_MJ_SET_QUOTA	IRP_MJ_SET_QUOTA
0x1B	**IRP_MJ_PNP_POWER**	**IRP_MJ_PNP**

As soon as the IRP array is complete, `DriverEntry()` writes a pointer to its `DriverUnload()` callback function to the driver object structure. This allows the driver to be unloaded at runtime. `DriverUnload()` simply destroys all objects created by `DriverInitialize()`, that is, the symbolic link and the device. After that, the driver can be safely removed from the system.

The `DriverDispatcher()` function is invoked whenever a module requests a response from the driver. Because a driver can host several devices, the dispatcher first checks which device should handle the request. The driver skeleton maintains just a single device, so the only thing needed is a sanity check to verify that the device object pointer is identical to the one received from `IoCreateDevice()` during initialization. If it is, `DriverDispatcher()` forwards the received IRP to the `DeviceDispatcher()` function, along with the device context prepared by `DriverInitialize()`. When you extend the skeleton to a multidevice driver, you may have to write distinct IRP dispatchers for each device. The `DeviceDispatcher()` in Listing 3-3 is a trivial implementation that recognizes only three very common requests: `IRP_MJ_CREATE`, `IRP_MJ_CLEANUP`, and `IRP_MJ_CLOSE`. These requests are handled by returning a `STATUS_SUCCESS` code. This is the minimum requirement to allow the device to be opened and closed without error. Other requests cause a `STATUS_NOT_IMPLEMENTED` to be reported.

You may wonder about the purpose of the `#pragma alloc_text` lines in the `DISCARDABLE FUNCTIONS` section of Listing 3-3. `#pragma` directives are a powerful means to send commands to the compiler and linker while they are building a module. The `alloc_text` command instructs them to write the code of the specified function to a nondefault section inside the executable file. By default, all code goes into the `.text` section. However, the directive `#pragma alloc_text (INIT, DriverEntry)` causes the `DriverEntry()` code to be saved to a new file section called `INIT`. The driver loader recognizes this special section and discards it after initialization. `DriverEntry()` and its helper function `DriverInitialize()` are called only once while the driver starts up; therefore, they can be safely removed from memory after having done their work.

The remaining ingredient of the driver skeleton is the resource script `TestDrv.rc`, shown in Listing 3-6. This file is trivial because it consists of references to macros from `DrvInfo.h` only. `DRV_RC_VERSION` creates a `VERSIONINFO` resource with various items compiled from data contributed by the wizard, and `DRV_RC_ICON` evaluates to a simple `ICON` resource statement that adds `TestDrv.ico` to the resource section of `TestDrv.sys`.

Device I/O Control

As mentioned in the introductory remarks of this chapter, we won't build hardware drivers in this book. Instead, we will use the powerful capabilities of kernel-mode drivers to investigate Windows 2000 secrets. The power of the drivers results from the fact that these modules run at the highest possible CPU privilege level. This means that a kernel-mode driver has access to all system resources, can read all memory, and is allowed to execute privileged CPU instructions, such as reading the

```
// TestDrv.rc
// 08-27-2000 <MyName>
// Copyright © 2000 <MyCompany>

#define  _RC_PASS_
#define  _TESTDRV_SYS_
#include "TestDrv.h"

// ====================================================================
// STANDARD RESOURCES
// ====================================================================

DRV_RC_VERSION
DRV_RC_ICON

// ====================================================================
// END OF FILE
// ====================================================================
```

LISTING 3-6. *The Resource Script of the Driver Skeleton*

current values of the CPU's control registers. User-mode applications will be aborted immediately if they try to read a single byte from kernel memory or try to execute an assembly language instruction such as `MOV EAX, CR3`. However, the downside of this power is that a driver can trash the entire system with a snap. Even the smallest error is answered by the system with a Blue Screen, so a kernel-mode programmer must be far more concerned about bugs than is a Win32 application or DLL developer. Remember the Windows 2000 killer device we used in Chapter 1 to get a crash dump of the system? All it did was touch the virtual memory address `0x00000000`—and boom! Be aware that you will boot your machine much more frequently when developing kernel-mode drivers.

The driver code I will present in the following chapters will employ a technique called Device I/O Control (IOCTL) to allow user-mode code some degree of "remote control." If an application needs access to some system resources that are unreachable from user-mode, a kernel-mode driver will do the job, and IOCTL will be the bridge between the two. Actually, IOCTL is neither new nor specific to Windows 2000. Even ancient operating systems such as DOS 2.11 had this capability—`Function 0x44` with its various subfunctions has been the IOCTL workhorse of DOS. Basically, IOCTL is a means to communicate with a device on a control channel, which is logically separated from its data channel. Imagine a hard disk device that transfers disk sector contents through its main data channel. If a client wants information about the media currently used by the device, it has to use a different

channel. For example, DOS function `0x44`, subfunction `0x0D`, sub-subfunction `0x66` is the DOS IOCTL call that reads the 32-bit serial number of a disk drive (see Brown and Kyle 1991, 1993).

Device I/O Control can be implemented in various ways, depending on the device to be controlled. In its general form, IOCTL has the following characteristics:

- A client controls a device through a special entry point. On DOS, this has been `INT 21h`, function `0x44`. On Windows 2000, it is the Win32 `DeviceIoControl()` function exported by `kernel32.dll`.
- The client provides a device identifier, a control code, an input data buffer, and an output data buffer upon calling the IOCTL entry point. On Windows 2000, the device identifier is a `HANDLE` to a successfully opened device.
- The control code tells the target device's IOCTL dispatcher which control function is requested by the client.
- The input buffer contains any additional data that the device might need to fulfill the request.
- If the request generates any data, it is returned in the client's output buffer.
- The overall result of the IOCTL operation is reported to the client by means of a status code.

It is obvious that this is a powerful general-purpose mechanism that can cover a wide range of control requests. For example, an application might want to have access to forbidden kernel memory. Because the application would throw an exception as soon as it touched the first byte, it could work around this problem by loading a kernel-mode driver to delegate this task. Both modules would have to agree on an IOCTL protocol to manage the data transfer. For example, the application might send the control code `0x80002000` to the driver if it wanted to read memory or `0x80002001` if it wanted to write to it. In a read request, the IOCTL input buffer would probably specify the base address and the number of bytes to read. The kernel-mode driver could pick up requests and distinguish read and write operations by evaluating the control code. In a read request, it would copy the requested memory range to the caller's output buffer and report success if the output buffer is large enough to hold the data. In a write request, the driver would copy data from the input buffer to a memory location that has been specified in the input buffer as well. In Chapter 4, I will provide sample code for such a memory spy.

By now, it should be obvious that IOCTL is a sort of backdoor that Win32 applications can use to perform almost any action that is usually allowed to privileged modules only. Of course, this involves writing such a privileged module in the first place, but once you have such a spy module running in the system, everything else is easy. Two aims of this book are to demonstrate in detail how to write such code and to provide a sample driver that is capable of doing lots of amazing things.

The Windows 2000 Killer Device

Before stepping to more advanced driver projects, let's take a look at a very simple driver. In Chapter 1, I introduced the Windows 2000 killer device `w2k_kill.sys`, which is designed to cause a benign system crash. This driver doesn't require most of the code in Listing 3-3 because it will tear down the system before it had an opportunity to receive the first I/O request packet. Listing 3-7 shows its apparently trivial implementation. The file `w2k_kill.h` is not reprinted here because it doesn't contain any code of interest.

The code in Listing 3-7 does not attempt to perform initialization inside its `DriverEntry()` function. The system will stop before `DriverEntry()` returns, so extra work is unnecessary.

LOADING AND UNLOADING DRIVERS

After writing a kernel-mode driver, you probably want to run it immediately. How is this done? Typically, drivers are loaded and started at system boot time, so do you have to reboot the system every time you have updated your driver? Fortunately, this is not necessary. Windows 2000 features a Win32 interface that allows loading and unloading drivers at runtime. This is done by the Service Control (SC) Manager, and the following section details its use.

The Service Control Manager

The name "Service Control Manager" is a bit misleading because it suggests that this component manages services only. Services are a class of powerful Windows 2000 modules well suited to run applications in the background, independent of the user interface shell. That is, a service is a Win32 process that can keep running in the system even if no user is logged in. Although service development is an exciting topic, it is beyond the scope of this book. For further reading on service development, refer to Paula Tomlinson's excellent tutorial in *Windows Developer's Journal (WDJ)* (Tomlinson 1996a), as well as her follow-up treatises on services in her *WDJ* column "Understanding NT" (Tomlinson 1996b and follow-up articles).

```
#define  _W2K_KILL_SYS_
#include <ddk\ntddk.h>
#include "w2k_kill.h"

// ====================================================================
// DISCARDABLE FUNCTIONS
// ====================================================================

NTSTATUS DriverEntry (PDRIVER_OBJECT  pDriverObject,
                      PUNICODE_STRING pusRegistryPath);

#ifdef ALLOC_PRAGMA
#pragma alloc_text (INIT, DriverEntry)
#endif

// ====================================================================
// DRIVER INITIALIZATION
// ====================================================================

NTSTATUS DriverEntry (PDRIVER_OBJECT  pDriverObject,
                      PUNICODE_STRING pusRegistryPath)
    {
    return *((NTSTATUS *) 0);
    }

// ====================================================================
// END OF PROGRAM
// ====================================================================
```

LISTING 3-7. *A Tiny System Crasher*

The SC Manager can handle both services and drivers. For reasons of simplicity, I will use the term "service" here to refer to all objects controlled by the SC Manager, including services in the strict sense of the word and kernel-mode drivers. The SC interface is made available to Win32 applications by the Win32 subsystem component `advapi32.dll`, which hosts an interesting collection of API functions. The names of the main API functions required to load, control, and unload services are listed in Table 3-3, along with short descriptions. Before you can load or access any services, you must obtain a handle to the SC Manager by calling `OpenSCManager()`. In the following discussion, this will be called a *manager handle.* This handle is required in all `CreateService()` and `OpenService()` calls. In turn, these functions return handles that will be called *service handles* here. This type of handle can be specified in all calls that refer to a specific service, such as `ControlService()`, `DeleteService()`, and `StartService()`. Both types of SC handles are released by the `CloseServiceHandle()` function.

TABLE 3-3. *Essential Service Control API Functions*

NAME	DESCRIPTION
CloseServiceHandle	Close handle obtained from `OpenSCManager()`, `CreateService()`, or `OpenService()`
ControlService	Stop, pause, continue, interrogate, or notify a loaded service/driver
CreateService	Load a service/driver
DeleteService	Unload a service/driver
OpenSCManager	Obtain a handle to the SC Manager
OpenService	Obtain a handle to a loaded service/driver
QueryServiceStatus	Query the properties and the current state of a service/driver
StartService	Start a loaded service/driver

Loading and running a service involves the following typical sequence of steps:

1. Call `OpenSCManager()` to obtain a manager handle.
2. Call `CreateService()` to add the service to the system.
3. Call `StartService()` to set the service to the running state.
4. Call `CloseServiceHandle()` to release the manager and service handles.

Be sure to rewind all previous successful actions if an error occurs somewhere in this sequence. For example, you should call `DeleteService()` if the SC Manager reports an error on `StartService()`. Otherwise, the service will remain loaded in an undesired state. Another stumbling stone of the SC Manager API is that the `CreateService()` function insists on receiving a fully qualified path to the executable file. If you specify a relative path, the function will fail—it will not be looking for the file in the current directory. Therefore, you should normalize all file specifications passed to `CreateService()` using the Win32 function `GetFullPathName()` unless they are guaranteed to be fully qualified.

HIGH-LEVEL DRIVER MANAGEMENT FUNCTIONS

To make interaction with the SC Manager easier, the CD accompanying this book contains high-level wrapper functions that hide most of its peculiarities. These functions are part of the large Windows 2000 Utility Library found on the CD in the directory tree `\src\w2k_lib`. All functions exported by `w2k_lib.dll` have a global name prefix of `w2k_`, and the service and driver management functions are discernible by the group name prefix `w2kService`. Listing 3-8 shows the implementation of the library functions that load, control, and unload services and drivers.

```
SC_HANDLE WINAPI w2kServiceConnect (void)
    {
    return OpenSCManager (NULL, NULL, SC_MANAGER_ALL_ACCESS);
    }

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

SC_HANDLE WINAPI w2kServiceDisconnect (SC_HANDLE hManager)
    {
    if (hManager != NULL) CloseServiceHandle (hManager);
    return NULL;
    }
// ---------------------------------------------------------------------

SC_HANDLE WINAPI w2kServiceManager (SC_HANDLE  hManager,
                                    PSC_HANDLE phManager,
                                    BOOL       fOpen)
    {
    SC_HANDLE hManager1 = NULL;

    if (phManager != NULL)
        {
        if (fOpen)
            {
            if (hManager == NULL)
                {
                *phManager = w2kServiceConnect ();
                }
            else
                {
                *phManager = hManager;
                }
            }
        else
            {
            if (hManager == NULL)
                {
                *phManager = w2kServiceDisconnect (*phManager);
                }
            }
        hManager1 = *phManager;
        }
    return hManager1;
    }

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

SC_HANDLE WINAPI w2kServiceOpen (SC_HANDLE hManager,
                                 PWORD     pwName)
    {
    SC_HANDLE hManager1;
```

(continued)

```
    SC_HANDLE hService = NULL;

    w2kServiceManager (hManager, &hManager1, TRUE);

    if ((hManager1 != NULL) && (pwName != NULL))
        {
        hService = OpenService (hManager1, pwName,
                                SERVICE_ALL_ACCESS);
        }
    w2kServiceManager (hManager, &hManager1, FALSE);
    return hService;
    }

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

BOOL WINAPI w2kServiceClose (SC_HANDLE hService)
    {
    return (hService != NULL) && CloseServiceHandle (hService);
    }

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

BOOL WINAPI w2kServiceAdd (SC_HANDLE hManager,
                           PWORD     pwName,
                           PWORD     pwInfo,
                           PWORD     pwPath)
    {
    SC_HANDLE hManager1, hService;
    PWORD     pwFile;
    WORD      awPath [MAX_PATH];
    DWORD     n;
    BOOL      fOk = FALSE;

    w2kServiceManager (hManager, &hManager1, TRUE);

    if ((hManager1 != NULL) && (pwName != NULL) &&
        (pwInfo    != NULL) && (pwPath != NULL) &&
        (n = GetFullPathName (pwPath, MAX_PATH, awPath, &pwFile)) &&
        (n < MAX_PATH))
        {
        if ((hService = CreateService (hManager1, pwName, pwInfo,
                                       SERVICE_ALL_ACCESS,
                                       SERVICE_KERNEL_DRIVER,
                                       SERVICE_DEMAND_START,
                                       SERVICE_ERROR_NORMAL,
                                       awPath, NULL, NULL,
                                       NULL, NULL, NULL))
            != NULL)
            {
            w2kServiceClose (hService);
            fOk = TRUE;
```

```
            }
        else
            {
            fOk = (GetLastError () ==
                   ERROR_SERVICE_EXISTS);
            }
        }
    w2kServiceManager (hManager, &hManager1, FALSE);
    return fOk;
    }

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

BOOL WINAPI w2kServiceRemove (SC_HANDLE hManager,
                              PWORD     pwName)
    {
    SC_HANDLE hService;
    BOOL      fOk = FALSE;

    if ((hService = w2kServiceOpen (hManager, pwName)) != NULL)
        {
        if (DeleteService (hService))
            {
            fOk = TRUE;
            }
        else
            {
            fOk = (GetLastError () ==
                   ERROR_SERVICE_MARKED_FOR_DELETE);
            }
        w2kServiceClose (hService);
        }
    return fOk;
    }

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

BOOL WINAPI w2kServiceStart (SC_HANDLE hManager,
                             PWORD     pwName)
    {
    SC_HANDLE hService;
    BOOL      fOk = FALSE;

    if ((hService = w2kServiceOpen (hManager, pwName)) != NULL)
        {
        if (StartService (hService, 1, &pwName))
            {
            fOk = TRUE;
            }
        else
            {
```

(continued)

```
            fOk = (GetLastError () ==
                   ERROR_SERVICE_ALREADY_RUNNING);
            }
        w2kServiceClose (hService);
        }
    return fOk;
    }

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

BOOL WINAPI w2kServiceControl (SC_HANDLE hManager,
                               PWORD     pwName,
                               DWORD     dControl)
    {
    SC_HANDLE      hService;
    SERVICE_STATUS ServiceStatus;
    BOOL           fOk = FALSE;

    if ((hService = w2kServiceOpen (hManager, pwName)) != NULL)
        {
        if (QueryServiceStatus (hService, &ServiceStatus))
            {
            switch (ServiceStatus.dwCurrentState)
                {
                case SERVICE_STOP_PENDING:
                case SERVICE_STOPPED:
                    {
                    fOk = (dControl == SERVICE_CONTROL_STOP);
                    break;
                    }
                case SERVICE_PAUSE_PENDING:
                case SERVICE_PAUSED:
                    {
                    fOk = (dControl == SERVICE_CONTROL_PAUSE);
                    break;
                    }
                case SERVICE_START_PENDING:
                case SERVICE_CONTINUE_PENDING:
                case SERVICE_RUNNING:
                    {
                    fOk = (dControl == SERVICE_CONTROL_CONTINUE);
                    break;
                    }
                }
            }
        fOk = fOk ||
              ControlService (hService, dControl, &ServiceStatus);

        w2kServiceClose (hService);
        }
```

```
    return fOk;
    }

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

BOOL WINAPI w2kServiceStop (SC_HANDLE hManager,
                            PWORD     pwName)
    {
    return w2kServiceControl (hManager, pwName,
                              SERVICE_CONTROL_STOP);
    }

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

BOOL WINAPI w2kServicePause (SC_HANDLE hManager,
                             PWORD     pwName)
    {
    return w2kServiceControl (hManager, pwName,
                              SERVICE_CONTROL_PAUSE);
    }

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

BOOL WINAPI w2kServiceContinue (SC_HANDLE hManager,
                                PWORD     pwName)
    {
    return w2kServiceControl (hManager, pwName,
                              SERVICE_CONTROL_CONTINUE);
    }

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

SC_HANDLE WINAPI w2kServiceLoad (PWORD pwName,
                                 PWORD pwInfo,
                                 PWORD pwPath,
                                 BOOL  fStart)
    {
    BOOL      fOk;
    SC_HANDLE hManager = NULL;

    if ((hManager = w2kServiceConnect ()) != NULL)
        {
        fOk = w2kServiceAdd (hManager, pwName, pwInfo, pwPath);

        if (fOk && fStart)
            {
            if (!(fOk = w2kServiceStart (hManager, pwName)))
                {
                w2kServiceRemove (hManager, pwName);
                }
            }
```

(continued)

```
        if (!fOk)
            {
            hManager = w2kServiceDisconnect (hManager);
            }
        }
    return hManager;
    }

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

SC_HANDLE WINAPI w2kServiceLoadEx (PWORD pwPath,
                                   BOOL  fStart)
    {
    PVS_VERSIONDATA pvvd;
    PWORD           pwPath1, pwInfo;
    WORD            awName [MAX_PATH];
    DWORD           dName, dExtension;
    SC_HANDLE       hManager = NULL;

    if (pwPath != NULL)
        {
        dName = w2kPathName (pwPath, &dExtension);

        lstrcpyn (awName, pwPath + dName,
                  min (MAX_PATH, dExtension - dName + 1));

        pwPath1 = w2kPathEvaluate (pwPath,  NULL);
        pvvd    = w2kVersionData  (pwPath1, -1);

        pwInfo  = ((pvvd != NULL) && pvvd->awFileDescription [0]
                   ? pvvd->awFileDescription
                   : awName);

        hManager = w2kServiceLoad (awName, pwInfo, pwPath1, fStart);

        w2kMemoryDestroy (pvvd);
        w2kMemoryDestroy (pwPath1);
        }
    return hManager;
    }

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

BOOL WINAPI w2kServiceUnload (PWORD     pwName,
                              SC_HANDLE hManager)
    {
    SC_HANDLE hManager1 = hManager;
    BOOL      fOk       = FALSE;

    if (pwName != NULL)
        {
```

```
        if (hManager1 == NULL)
            {
            hManager1 = w2kServiceConnect ();
            }
        if (hManager1 != NULL)
            {
            w2kServiceStop (hManager1, pwName);
            fOk = w2kServiceRemove (hManager1, pwName);
            }
        }
    w2kServiceDisconnect (hManager1);
    return fOk;
    }

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

BOOL WINAPI w2kServiceUnloadEx (PWORD     pwPath,
                                SC_HANDLE hManager)
    {
    DWORD dName, dExtension;
    WORD  awName [MAX_PATH];
    PWORD pwName = NULL;

    if (pwPath != NULL)
        {
        dName = w2kPathName (pwPath, &dExtension);

        lstrcpyn (pwName = awName, pwPath + dName,
                  min (MAX_PATH, dExtension - dName + 1));
        }
    return w2kServiceUnload (pwName, hManager);
    }
```

LISTING 3-8. *Service and Driver Management Library Functions*

In Table 3-4, the functions defined in Listing 3-8 are listed, along with short descriptions. Some function names, such as `w2kServiceStart()` and `w2kServiceControl()`, are similar to certain SC Manager API functions—`StartService()` and `ControlService()`, in this case. This isn't coincidence—the respective functions are in fact found at the heart of these wrappers. The main difference is that `StartService()` and `ControlService()` operate on service handles, whereas `w2kServiceStart()` and `w2kServiceControl()` accept service names. The names are seamlessly converted to handles by internally calling `w2kServiceOpen()` and `w2kServiceClose()`, which in turn call `OpenService()` and `CloseServiceHandle()`.

TABLE 3-4. *SC Manager Wrappers Provided by* `w2k_lib.dll`

NAME	DESCRIPTION
w2kServiceAdd	Add a service/driver to the system
w2kServiceClose	Close a service handle
w2kServiceConnect	Connect to the Service Control Manager
w2kServiceContinue	Resume a paused service/driver
w2kServiceControl	Stop, pause, continue, interrogate, or notify a loaded service/driver
w2kServiceDisconnect	Disconnect from the Service Control Manager
w2kServiceLoad	Load and optionally start a service/driver
w2kServiceLoadEx	Load and optionally start a service/driver (automatic name generation)
w2kServiceManager	Open/close a temporary Service Control Manager handle
w2kServiceOpen	Obtain a handle to a loaded service/driver
w2kServicePause	Pause a running service/driver
w2kServiceRemove	Remove a service/driver from the system
w2kServiceStart	Start a loaded service/driver
w2kServiceStop	Stop a running service/driver
w2kServiceUnload	Stop and unload a service/driver
w2kServiceUnloadEx	Stop and unload a service/driver (automatic name generation)

The typical usage of the library functions in Table 3-4 is along the following guidelines:

- To load a service, call `w2kServiceLoad()` or `w2kServiceLoadEx()`. The latter generates the service and display names automatically from the file's path and version resource. The Boolean `fStart` argument decides whether the service should be started automatically after a successful load. On success, the function returns a manager handle for further requests. No error is reported if the service is already loaded or if `fStart` is `TRUE` and the service is already running. If an error occurs, the service is automatically unloaded, if necessary.
- To unload a service, call `w2kServiceUnload()` or `w2kServiceUnloadEx()`, using the manager handle returned by `w2kServiceLoad()` or `w2kServiceLoadEx()`. `w2kServiceUnloadEx()` generates the service name automatically from the file's path. If you have already closed this handle,

obtain a new one from `w2kServiceConnect()` or simply pass in `NULL` to work with a temporary handle. The manager handle will be closed automatically by `w2kServiceUnload()`. No error is reported if the service is already marked for deletion but cannot be deleted because open device handles are still existing.

- To control a service, call `w2kServiceStart()`, `w2kServiceStop()`, `w2kServicePause()`, or `w2kServiceContinue()`, using a manager handle returned by `w2kServiceLoad()` or `w2kServiceConnect()`. If you supply `NULL` for the manager handle, a temporary handle is used. No error is reported if the service is already in the requested state.
- To close a manager handle, call `w2kServiceDisconnect()`. You can request another manager handle at any time by calling `w2kServiceConnect()`.

`w2kServiceLoadEx()` is a very powerful function. It builds all parameters needed to load the service automatically, expecting nothing but the path of the executable file. The service name requested by the SC Manager's `CreateService()` function is derived from the file name by stripping the extension. To build an appropriate display name for a newly created service, `w2kServiceLoadEx()` attempts to read the value of the `FileDescription` string from the file version information. If no version resource is included in the executable, or the `FileDescription` string is not available, the service name is used by default. Unlike `w2kServiceLoad()`, `w2kServiceLoadEx()` evaluates environment variables embedded in the path. That is, if the path string contains substrings such as `%SystemRoot%` or `%TEMP%`, they are replaced by the current values of the corresponding environment variables. `w2kServiceUnloadEx()` is the counterpart of `w2kServiceLoadEx()`—it extracts the service name from the supplied path, as explained above, and passes it to `w2kServiceUnload()`. Both functions are ideally suited for applications that have to load and unload third-party device drivers on behalf of the user, knowing nothing about them but their executable paths. A sample application of this kind is included on the CD accompanying this book. The console-mode utility `w2k_load.exe` is a general-purpose kernel-mode device driver (un)loader that provides a simple command line interface for `w2kServiceLoadEx()` and `w2kServiceUnloadEx()`. The source files can be found on the CD in the directory tree `\src\w2k_load`. The relevant code is shown in Listing 3-9, proving that this utility is almost trivial because all the hard work is done inside `w2k_lib.dll` by the `w2kServiceLoadEx()` and `w2kServiceUnloadEx()` functions.

```
// ====================================================================
// GLOBAL STRINGS
// ====================================================================

WORD awUsage  [] =
    L"\r\n"
    L"Usage: " SW(MAIN_MODULE) L" <driver path>\r\n"
    L"       " SW(MAIN_MODULE) L" <driver path> %s\r\n"
    L"       " SW(MAIN_MODULE) L" <driver name> %s\r\n";

WORD awUnload [] = L"/unload";

WORD awOk     [] = L"OK\r\n";
WORD awError  [] = L"ERROR\r\n";

// ====================================================================
// COMMAND HANDLERS
// ====================================================================

BOOL WINAPI DriverLoad (PWORD pwPath)
    {
    SC_HANDLE hManager;
    BOOL      fOk = FALSE;

    _printf (L"\r\nLoading \"%s\" ... ", pwPath);

    if ((hManager = w2kServiceLoadEx (pwPath, TRUE)) != NULL)
        {
        w2kServiceDisconnect (hManager);
        fOk = TRUE;
        }
    _printf (fOk ? awOk : awError);
    return fOk;
    }

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

BOOL WINAPI DriverUnload (PWORD pwPath)
    {
    BOOL fOk = FALSE;

    _printf (L"\r\nUnloading \"%s\" ... ", pwPath);

    fOk = w2kServiceUnloadEx (pwPath, NULL);
    _printf (fOk ? awOk : awError);
    return fOk;
    }

// ====================================================================
// MAIN PROGRAM
// ====================================================================
```

```
DWORD Main (DWORD argc, PTBYTE *argv, PTBYTE *argp)
    {
    if (argc == 2)
        {
        DriverLoad (argv [1]);
        }
    else
        {
        if ((argc == 3) && (!lstrcmpi (argv [2], awUnload)))
            {
            DriverUnload (argv [1]);
            }
        else
            {
            _printf (awUsage, awUnload, awUnload);
            }
        }
    return 0;
    }

// ====================================================================
// END OF PROGRAM
// ====================================================================
```

LISTING 3-9. *Loading and Unloading Device Drivers*

The remaining library functions listed in Table 3-4 are working on a lower level and are used internally by `w2k_lib.dll`. Of course, you can call them from your applications, if you like. Their usage should be obvious from the source code in Listing 3-8.

ENUMERATING SERVICES AND DRIVERS

From time to time it might be necessary to know which services and drivers are currently loaded inside the system and what state they are in. For this purpose, the SC Manager provides another powerful function named `EnumServicesStatus()`. This function requires a manager handle, as usual, and fills an array of `ENUM_SERVICE_STATUS` structures with information about each currently loaded service or driver. The list can be filtered by service/driver type and state. If the buffer supplied by the caller isn't large enough to hold all entries at once, the function can be called repeatedly until all items have been retrieved. It is difficult to compute the required buffer size in advance because the buffer has to provide extra space of unknown size for the strings that are referenced by the members of the `ENUM_SERVICE_STATUS` structures. Fortunately, `EnumServicesStatus()` returns the number of bytes needed to return the remaining entries, so the correct buffer size can be determined by trial and error. Listing 3-10 shows the definitions of the `SERVICE_STATUS` and `ENUM_SERVICE_STATUS` structures, which are declared in the Win32 header file `WinSvc.h`.

```
typedef struct _SERVICE_STATUS
    {
    DWORD dwServiceType;
    DWORD dwCurrentState;
    DWORD dwControlsAccepted;
    DWORD dwWin32ExitCode;
    DWORD dwServiceSpecificExitCode;
    DWORD dwCheckPoint;
    DWORD dwWaitHint;
    }
    SERVICE_STATUS, *LPSERVICE_STATUS;

typedef struct _ENUM_SERVICE_STATUS
    {
    LPTSTR         lpServiceName;
    LPTSTR         lpDisplayName;
    SERVICE_STATUS ServiceStatus;
    }
    ENUM_SERVICE_STATUS;
```

LISTING 3-10. *Definition of* `ENUM_SERVICE_STATUS` *and* `SERVICE_STATUS`

The `w2kServiceList()` function in Listing 3-11 is another goodie from the `w2k_lib.dll` utility library on the companion CD. It hides the required actions mentioned above and returns a ready-to-use structure with all requested data plus a couple of extras. It returns a pointer to a `W2K_SERVICES` structure, defined in `w2k_lib.h` and included at the top of Listing 3-11. Along with the `ENUM_SERVICE_STATUS` array `aess[]`, this structure contains four additional members. dEntries indicates how many entries have been copied to the status array, and `dBytes` specifies the total size of the returned `W2K_SERVICES` structure. `dDisplayName` and `dServiceName` are set to the maximum lengths of the `lpDisplayName` and `lpServiceName` strings in `aess[]`, respectively. These values are very convenient if you are writing a console-mode application that outputs a service/driver list to the screen with proper alignment of the name columns.

To report an accurate snapshot of the system, `w2kServiceList()` attempts to retrieve all entries in a single call to `EnumServicesStatus()`. To this end, it starts out with a zero-length buffer, which will usually yield an `ERROR_MORE_DATA` status. In this case, `EnumServicesStatus()` returns the required buffer size. After allocating an appropriately sized buffer, `w2kServiceList()` tries again. This time, `EnumServicesStatus()` should succeed. However, a small probability exists that another entry has been added to the list in the meantime, so this procedure is repeated in a loop until everything is correct or an error other than `ERROR_MORE_DATA` is returned.

```
typedef struct _W2K_SERVICES
    {
    DWORD               dEntries;     // number of entries in aess[]
    DWORD               dBytes;       // overall number of bytes
    DWORD               dDisplayName; // maximum display name length
    DWORD               dServiceName; // maximum service name length
    ENUM_SERVICE_STATUS aess [];      // service/driver status array
    }
    W2K_SERVICES, *PW2K_SERVICES, **PPW2K_SERVICES;

#define W2K_SERVICES_ sizeof (W2K_SERVICES)

PW2K_SERVICES WINAPI w2kServiceList (BOOL fDriver,
                                     BOOL fWin32,
                                     BOOL fActive,
                                     BOOL fInactive)
    {
    SC_HANDLE     hManager;
    DWORD         dType, dState, dBytes, dResume, dName, i;
    PW2K_SERVICES pws = NULL;

    if ((pws = w2kMemoryCreate (W2K_SERVICES_)) != NULL)
        {
        pws->dEntries     = 0;
        pws->dBytes       = 0;
        pws->dDisplayName = 0;
        pws->dServiceName = 0;

        if ((fDriver || fWin32) && (fActive || fInactive))
            {
            if ((hManager = w2kServiceConnect ()) != NULL)
                {
                dType   = (fDriver ? SERVICE_DRIVER : 0) |
                          (fWin32  ? SERVICE_WIN32  : 0);

                dState  = (fActive && fInactive
                           ? SERVICE_STATE_ALL
                           : (fActive
                              ? SERVICE_ACTIVE
                              : SERVICE_INACTIVE));

                dBytes  = pws->dBytes;
                while (pws != NULL)
                    {
                    pws->dEntries     = 0;
                    pws->dBytes       = dBytes;
                    pws->dDisplayName = 0;
                    pws->dServiceName = 0;

                    dResume = 0;
```

(continued)

```
                    if (EnumServicesStatus (hManager, dType, dState,
                                            pws->aess, pws->dBytes,
                                            &dBytes, &pws->dEntries,
                                            &dResume))
                        break;

                    dBytes += pws->dBytes;
                    pws     = w2kMemoryDestroy (pws);

                    if (GetLastError () != ERROR_MORE_DATA) break;

                    pws = w2kMemoryCreate (W2K_SERVICES_ + dBytes);
                    }
                w2kServiceDisconnect (hManager);
                }
            else
                {
                pws = w2kMemoryDestroy (pws);
                }
            }
        if (pws != NULL)
            {
            for (i = 0; i < pws->dEntries; i++)
                {
                dName = lstrlen (pws->aess [i].lpDisplayName);
                pws->dDisplayName = max (pws->dDisplayName, dName);

                dName = lstrlen (pws->aess [i].lpServiceName);
                pws->dServiceName = max (pws->dServiceName, dName);
                }
            }
        }
    return pws;
    }
```

LISTING 3-11. *Enumerating Services and Drivers*

`w2kServiceList()` expects four Boolean arguments determining the contents of the returned list. With the `fDriver` and `fWin32` arguments, you can choose the inclusion of drivers and services, respectively. If both flags are set, the list will contain both drivers and services. The `fActive` and `fInactive` flags impose a state filter onto the list. If `fActive` is set, the list contains all modules that currently are in the running or paused state. The `fInactive` parameter selects the remaining modules, that is, those that are currently loaded but stopped. If all four arguments are `FALSE`, the function returns a `W2K_SERVICES` structure with an empty status array. The sample code CD contains a simple service and driver browser, designed as a Win32 console-mode application and based on the `w2kServiceList()` function of `w2k_lib.dll`. It uses the `dDisplayName` and `dServiceName` members of the returned `W2K_SERVICES`

structure (see Listing 3-11) for proper horizontal alignment of all names. You can find the source code of this utility in the CD's directory tree `\src\w2k_svc`. The program can be run from the CD by executing `\bin\w2k_svc.exe`. Example 3-4 resulted from running it on my machine, requesting a list of all active kernel-mode drivers by specifying the command switches `/drivers /active`.

```
D:\> w2k_svc /drivers /active

// w2k_svc.exe
// SBS Windows 2000 Service List V1.00
// 08-27-2000 Sven B. Schreiber
// sbs@orgon.com

Found 29 active drivers:

   1. Alerter . . . . . . . . . . . . . . . . . . . . . . . . Alerter
   2. Computer Browser . . . . . . . . . . . . . . . . . . . Browser
   3. Creative Service for CDROM Access . . . . . . . . . . . Creative Service
   4. DHCP Client . . . . . . . . . . . . . . . . . . . . . . Dhcp
   5. Logical Disk Manager . . . . . . . . . . . . . . . . . dmserver
   6. DNS Client . . . . . . . . . . . . . . . . . . . . . . Dnscache
   7. Event Log . . . . . . . . . . . . . . . . . . . . . . . Eventlog
   8. COM+ Event System . . . . . . . . . . . . . . . . . . . EventSystem
   9. Server . . . . . . . . . . . . . . . . . . . . . . . . lanmanserver
  10. Workstation . . . . . . . . . . . . . . . . . . . . . . lanmanworkstation
  11. TCP/IP NetBIOS Helper Service . . . . . . . . . . . . . LmHosts
  12. Messenger . . . . . . . . . . . . . . . . . . . . . . . Messenger
  13. Network Connections . . . . . . . . . . . . . . . . . . Netman
  14. Removable Storage . . . . . . . . . . . . . . . . . . . NtmsSvc
  15. Plug and Play . . . . . . . . . . . . . . . . . . . . . PlugPlay
  16. IPSEC Policy Agent . . . . . . . . . . . . . . . . . . . PolicyAgent
  17. Protected Storage . . . . . . . . . . . . . . . . . . . ProtectedStorage
  18. Remote Access Connection Manager . . . . . . . . . . . . RasMan
  19. Remote Registry Service . . . . . . . . . . . . . . . . RemoteRegistry
  20. Remote Procedure Call (RPC) . . . . . . . . . . . . . . RpcSs
  21. Security Accounts Manager . . . . . . . . . . . . . . . SamSs
  22. Task Scheduler . . . . . . . . . . . . . . . . . . . . . Schedule
  23. RunAs Service . . . . . . . . . . . . . . . . . . . . . seclogon
  24. System Event Notification . . . . . . . . . . . . . . . SENS
  25. Print Spooler . . . . . . . . . . . . . . . . . . . . . Spooler
  26. Telephony . . . . . . . . . . . . . . . . . . . . . . . TapiSrv
  27. Distributed Link Tracking Client . . . . . . . . . . . . TrkWks
  28. Windows Management Instrumentation . . . . . . . . . . . WinMgmt
  29. Windows Management Instrumentation Driver Extensions . Wmi
```

EXAMPLE 3-4. *Running the Service List Utility* `w2k_svc.exe`

In the next chapter, we will start developing a real-world kernel-mode driver that spies on kernel memory and cracks essential memory management data structures. This project accompanies you while reading Chapters 4, 5, and 6, and the driver is enhanced incrementally in each chapter. The final result is a versatile Windows 2000 kernel spy, complemented by several nice client applications.