CHAPTER 6

Calling Kernel API Functions from User-Mode

In Chapter 2, I explained how Windows 2000 allows user-mode applications to call a subset of its kernel API functions—the Native API—by means of an interrupt gate mechanism. Chapters 4 and 5 relied heavily on a mechanism referred to as Device I/O Control (IOCTL) to carry out additional tasks that aren't allowed in user-mode. Both the Native API and IOCTL are quite powerful, but think of the benefit of being able to call almost `any` kernel-mode function as if it were located in a normal user-mode DLL. This is generally considered impossible. However, I will demonstrate in this chapter that it is possible with the help of a couple of wacky programming tricks. Again, IOCTL will come to the rescue to solve a problem that seems impossible at first sight. This chapter is revolutionary because it builds a general-purpose bridge from user-mode to kernel-mode, allowing the Win32 application to call kernel API functions just as if they were part of the Win32 API. Even better, an application can call internal kernel functions that are not even available to kernel-mode drivers, with the help of the symbol files coming with the Windows 2000 debugging tools. This "kernel call interface" works seamlessly in the background, almost completely unnoticed by the calling application.

A GENERAL KERNEL CALL INTERFACE

In Chapter 4, we used a kernel-mode driver to call selected kernel API functions on behalf of a user-mode program. For example, the `SPY_IO_PHYSICAL` function offered by the spy driver `w2k_spy.sys` is merely a wrapper around the memory manager's `MmGetPhysicalAddress()` function. Another example is `SPY_IO_HANDLE_INFO`, which is built upon the object manager's `ObReferenceObjectByHandle()` and `ObDereferenceObject()` functions. Although this technique works fine, it is quite tedious and inefficient to design a custom IOCTL function for every kernel API function that should be made available to user-mode code. Therefore, I have added a general-purpose IOCTL function to the spy device inside the sample driver `w2k_spy.sys` that calls arbitrary kernel-mode functions, given a symbolic name or an entry point plus a list of arguments. This sounds like a lot of work, but you will be surprised how simple the necessary code actually is. The only difficulty is that again we will need a good deal of inline assembly language (ASM).

DESIGNING A GATE TO KERNEL-MODE

If a program running in user-mode wants to call a kernel-mode function, it has to solve two problems. First, it must somehow jump across the barrier between user-mode and kernel-mode, and second, it must transfer data in and out. For the subset comprising the Native API, the `ntdll.dll` component takes over this duty, using an interrupt gate to accomplish the mode change and CPU registers to pass in a pointer to the caller's argument stack and to return the function's result to the caller. For kernel functions not included in the Native API, the operating system doesn't offer such a gate mechanism. Therefore, we will have to create our own. Part one of the problem is easily solved: The `w2k_spy.sys` driver introduced in Chapter 4 and extended in Chapter 5 crosses the user-to-kernel-mode border back and forth many times during its IOCTL transactions. And because IOCTL optionally allows passing data blocks in both directions, the date transfer problem is solved as well. In the end, the whole matter boils down to the following simple sequence of steps:

1. The user-mode application posts an IOCTL request, passing in information about the function to be called, as well as a pointer to its argument stack.
2. The kernel-mode driver dispatches the request, copies the arguments onto its own stack, calls the function, and passes the results back to the caller in the IOCTL output buffer.
3. The caller picks up the results of the IOCTL operation and proceeds as it would after a normal DLL function call.

The main problem with this scenario is that the kernel-mode module must cope with various data formats and calling conventions. Following is a list of situations the driver must be prepared for:

- The size of the argument stack depends on the target function. Because it is impractical to give the driver detailed knowledge about all functions it might possibly have to call, the caller must supply the size of the argument stack.

- Windows 2000 kernel API functions use three calling conventions: `__stdcall`, `__cdecl`, and `__fastcall`, which differ considerably in the way arguments are treated. `__stdcall` and `__cdecl` require all arguments to be passed in on the stack, whereas `__fastcall` aims at minimizing stack fumbling overhead by passing the first two arguments in the CPU registers `ECX` and `EDX`. On the other hand, `__stdcall` and `__fastcall` agree in the way arguments are removed from the stack, forcing the called code to take over the responsibility. `__cdecl`, however, leaves this task to the calling code. Although the stack cleanup problem can be easily solved by saving the stack pointer before the call and resetting it to its original position after returning, regardless of the calling convention, the driver is helpless with respect to the `__fastcall` convention. Therefore, the caller must specify on every call whether the `__fastcall` convention is in effect, to allow the driver to prepare the registers `ECX` and `EDX` if necessary.
- Windows 2000 kernel functions return results in various sizes, ranging from zero to 64 bits. The 64-bit register pair `EDX:EAX` transports the results back to the caller. Data is filled in from the least-significant end toward the most-significant end. For example, if a function returns a 16-bit `SHORT` data type, only register `AX` (comprising `AL` and `AH`) is significant. The upper half of `EAX` and the entire `EDX` contents are undefined. Because the driver is ignorant of the called function's I/O data, it must assume the worst case, which is 64-bits. Otherwise, the result may be truncated.
- The application might supply invalid arguments. In user-mode, this is usually benign. At worst, the application process is aborted with an error message box. Occasionally, this error results in system damage that requires a reboot for recovery. In kernel-mode, the most frequent programming error, known as "bad pointer," almost instantly results in a Blue Screen of Death, which might cause loss of user data. This problem can be addressed to a great extent by using the operating system's Structured Exception Handling (SEH) mechanism.

That said, let's examine how our spy driver handles function properties, arguments, and results. Listing 6-1 shows the involved IOCTL input and output structures, `SPY_CALL_INPUT` and `SPY_CALL_OUTPUT`. The latter is quite simple—it consists of a `ULARGE_INTEGER` structure that is used by Windows 2000 to represent a 64-bit value both as a single 64-bit integer and a pair of 32-bit halves. Please consult Listing 2-3 in Chapter 2 for the layout of this structure.

```
typedef struct _SPY_CALL_INPUT
    {
    BOOL  fFastCall;
    DWORD dArgumentBytes;
    PVOID pArguments;
    PBYTE pbSymbol;
    PVOID pEntryPoint;
    }
    SPY_CALL_INPUT, *PSPY_CALL_INPUT, **PPSPY_CALL_INPUT;

#define SPY_CALL_INPUT_ sizeof (SPY_CALL_INPUT)

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

typedef struct _SPY_CALL_OUTPUT
    {
    ULARGE_INTEGER uliResult;
    }
    SPY_CALL_OUTPUT, *PSPY_CALL_OUTPUT, **PPSPY_CALL_OUTPUT;

#define SPY_CALL_OUTPUT_ sizeof (SPY_CALL_OUTPUT)
```

LISTING 6-1. *Definition of* `SPY_CALL_INPUT` *and* `SPY_CALL_OUTPUT`

`SPY_CALL_INPUT` needs a bit more explanation. The purpose of the `fFastCall` member should be obvious. It signals to the spy driver that the function to be called obeys the `__fastcall` convention, so the first two arguments, if any, must not be passed in on the stack, but in CPU registers. `dArgumentBytes` specifies the number of bytes piled up on the argument stack, and `pArguments` points to the top of this stack. The remaining arguments, `pbSymbol` and `pEntryPoint`, are mutually exclusive, and tell the driver which function to execute. You can specify either a function name or a plain entry point address. The other member should always be set to `NULL`. If both values are non-`NULL`, `pbSymbol` takes precedence over `pEntryPoint`. Calling a function by name rather than by address adds an additional step, where the entry point of the specified symbolic name is determined. If it can be retrieved, the function is entered through this address. Passing in an entry point simply bypasses the symbol resolution step.

Finding the linear address associated with a symbol exported by a kernel-mode module sounds easier than it actually is. The powerful Win32 functions `GetModuleHandle()` and `GetProcAddress()`, which work fine with all components within the Win32 subsystem, do not recognize kernel-mode system modules and drivers. Implementing this part of the sample code was difficult, the details are covered in the next section of this chapter. For now, let's assume that a valid entry point is available, no matter how it has been supplied. Listing 6-2 shows the function `SpyCall()` that constitutes the core part of my kernel call interface. As you see, it is almost 100% assembly language. It is always unpleasant to resort to ASM in a C program, but some tasks simply can't be done in pure C. In this case, the problem is that `SpyCall()` needs total control of the stack and the CPU registers, and therefore it must bypass the C compiler and optimizer, which use the stack and registers as they see fit.

Before delving into the details of Listing 6-2, let me describe another special feature of the `SpyCall()` function that obscures the code. As explained in Chapter 2, the Windows 2000 system modules export some of their variables by name. Typical examples are `NtBuildNumber` and `KeServiceDescriptorTable`. The Portable Executable (PE) file format of Windows 2000/NT/9x provides a general-purpose mechanism for attaching symbols to addresses, regardless of whether an address points to code or data. Therefore, a Windows 2000 module is free to attach exported symbols to its global variables at will. A client module can dynamically link to them like it links to function symbols, and it is able to use these variables as if they were located in its own global data section. Of course, my kernel call interface would not be complete if it were not able to cope with this kind of symbol as well, so I decided that negative values of the `dArgumentBytes` member inside the `SPY_CALL_INPUT` structure should indicate that data is to be copied from the entry point instead of calling it. Valid values range from `-1` to `-9`, where `-1` means that the entry point address itself is copied to the `SPY_CALL_OUTPUT` buffer. For the remaining values, their one's complement states the number of bytes copied from the entry point, that is, `-2` copies a single `BYTE` or `CHAR`; `-3`, a 16-bit `WORD` or `SHORT`; `-5`, a 32-bit `DWORD` or `LONG`; and `-9` a 64-bit `DWORDLONG` or `LONGLONG`. You may wonder why it should be necessary to copy the entry point itself. Well, some kernel symbols, such as `KeServiceDescriptorTable` point to structures that exceed the 64-bit return value limit, so it is wiser to return the plain pointer rather than truncating the value to 64 bits.

```
void SpyCall (PSPY_CALL_INPUT  psci,
              PSPY_CALL_OUTPUT psco)
    {
    PVOID pStack;

    __asm
        {
        pushfd
        pushad
        xor     eax, eax
        mov     ebx, psco               ; get output parameter block
        lea     edi, [ebx.uliResult]    ; get result buffer
        mov     [edi  ], eax            ; clear result buffer (lo)
        mov     [edi+4], eax            ; clear result buffer (hi)
        mov     ebx, psci               ; get input parameter block
        mov     ecx, [ebx.dArgumentBytes]
```

(continued)

```
        cmp     ecx, -9                 ; call or store/copy?
        jb      SpyCall2
        mov     esi, [ebx.pEntryPoint]  ; get entry point
        not     ecx                     ; get number of bytes
        jecxz   SpyCall1                ; 0 -> store entry point
        rep     movsb                   ; copy data from entry point
        jmp     SpyCall5
SpyCall1:
        mov     [edi], esi              ; store entry point
        jmp     SpyCall5
SpyCall2:
        mov     esi, [ebx.pArguments]
        cmp     [ebx.fFastCall], eax    ; __fastcall convention?
        jz      SpyCall3
        cmp     ecx, 4                  ; 1st argument available?
        jb      SpyCall3
        mov     eax, [esi]              ; eax = 1st argument
        add     esi, 4                  ; remove argument from list
        sub     ecx, 4
        cmp     ecx, 4                  ; 2nd argument available?
        jb      SpyCall3
        mov     edx, [esi]              ; edx = 2nd argument
        add     esi, 4                  ; remove argument from list
        sub     ecx, 4
SpyCall3:
        mov     pStack, esp             ; save stack pointer
        jecxz   SpyCall4                ; no (more) arguments
        sub     esp, ecx                ; copy argument stack
        mov     edi, esp
        shr     ecx, 2
        rep     movsd
SpyCall4:
        mov     ecx, eax                ; load 1st __fastcall arg
        call    [ebx.pEntryPoint]       ; call entry point
        mov     esp, pStack             ; restore stack pointer
        mov     ebx, psco               ; get output parameter block
        mov     [ebx.uliResult.LowPart ], eax   ; store result (lo)
        mov     [ebx.uliResult.HighPart], edx   ; store result (hi)
SpyCall5:
        popad
        popfd
        }
    return;
    }
```

LISTING 6-2. *The Core Function of the Kernel Call Interface*

With the special case of accessing exported variables kept in mind, Listing 6-2 shouldn't be too difficult to understand. First, the 64-bit result buffer is cleared, guaranteeing that unused bits are always zero. Next, the `dArgumentBytes` member of the input data is compared with `-9` to find out whether the client requested a function call or a data copying operation. The function call handler starts at the label `SpyCall2`. After setting register `ESI` to the top of the argument stack by evaluating the `pArguments` member, it is time to check the calling convention. If `__fastcall` is required and there is at least one 32-bit value on the stack, `SpyCall()` removes it and stores it temporarily in `EAX`. If another 32-bit value is available, it is removed as well and stored in `EDX`. Any remaining arguments remain on the stack. Meanwhile, the label `SpyCall3` is reached. Now the current top-of-stack address is saved to the local variable `pStack`, and the argument stack (minus the arguments removed in the `__fastcall` case) is copied to the spy driver's stack using the fast i386 `REP MOVSD` instruction. Note that the direction flag that determines whether `MOVSD` proceeds upward or downward in memory can be assumed to be clear by default; that is, `ESI` and `EDI` are incremented after each copying step. The only thing left to do before executing the `CALL` instruction is to copy the first `__fastcall` argument from its preliminary location `EAX` to its final destination `ECX`. `SpyCall()` blindly copies `EAX` to `ECX` because this operation doesn't create havoc if the calling convention is `__stdcall` or `__cdecl`. The `MOV ECX, EAX` instruction is so fast that executing it in vain is much more efficient than jumping around it after testing the value of the `fFastCall` member.

After the call to the function's entry point returns, `SpyCall()` resets the stack pointer to the location saved off to the variable `pStack`. This takes care of the different stack cleanup policy of `__stdcall` and `__fastcall` versus `__cdecl`. A `__cdecl` function returns to the caller, with the `ESP` register pointing to the top of the argument stack, whereas an `__stdcall` or a `__fastcall` function resets it to its original address before the call. Forcing `ESP` to a previously backed-up address always cleans up the stack properly, no matter which calling convention is used. The last few ASM lines of `SpyCall()` store the function result returned in `EDX:EAX` to the caller's `SPY_CALL_OUTPUT` structure. No attempt is made to find out the correct result size. This is unnecessary because the caller knows exactly how many valid result bits it can expect. Copying too many bits does no harm—they are simply ignored by the caller.

One thing that should be noted about the code in Listing 6-2 is that it contains absolutely no provisions for invalid arguments. It does not even check the validity of the stack pointer itself. In kernel-mode, this is equivalent to playing with fire. However, how could the spy driver verify all arguments? A 32-bit value on the stack could be a counter value, a bit-field array, or maybe a pointer. Only the caller and the called target function know the argument semantics. The `SpyCall()` function is a simple pass-through layer that has no knowledge about the type of data it forwards. Adding context-sensitive argument checking to this function would amount to rewriting large parts of the operating system. Fortunately, Windows 2000 offers an easy way out of this dilemma: Structured Exception Handling (SEH).

SEH is an easy-to-use framework that enables a program to catch exceptions that would otherwise crash the system. An exception is an abnormal situation that forces the CPU to stop whatever it is currently doing. Typical operations that generate exceptions are reading from or writing to linear addresses that don't map to physical or paged-out memory, writing data to a code segment, attempting to execute instructions in a data segment, or dividing a number by zero. Some exceptions are benign. For example, accessing a memory location that has been swapped to a pagefile generates an exception that the system can handle by bringing the target page back to physical memory. However, most exceptions are fatal, because the operating system has no idea how to recover from the exception, so the system simply shuts down. This reaction might seem harsh, but sometimes it is better to halt an imminent catastrophe before things become worse. With SEH, the program that caused the exception is granted a second chance. Using the Microsoft-specific C construct `__try /__except`, an arbitrary sequence of instructions can be guarded against exceptions. If an exception puts the system into a critical state, a custom handler inside the program is invoked, allowing the programmer to provide a more useful reaction than just triggering a Blue Screen.

Obviously, SEH is also able to work around the parameter validation problem of our spy device. Listing 6-3 shows a wrapper that puts the `SpyCall()` function into a SEH frame. The guarded code is enclosed in the braces of the `__try` clause. Of course, not only the `SpyCall()` instruction is protected; all subordinate code that is executed in the context of the call is protected as well. If an exception is thrown, the code inside the `__except` clause is entered, as demanded by the filter expression `EXCEPTION_EXECUTE_HANDLER`. The exception handler in Listing 6-3 is trivial. It just causes `SpyCallEx()` to return the status code `STATUS_ACCESS_VIOLATION` instead of `STATUS_SUCCESS`, which will in turn result in failure of the `DeviceIoControl()` call on the user-mode side. No Blue Screen appears; the only problem remaining after the exception is that the results of the called function are undefined, but this is something the caller should be prepared for anyway.

```
NTSTATUS SpyCallEx (PSPY_CALL_INPUT  psci,
                    PSPY_CALL_OUTPUT psco)
    {
    NTSTATUS ns = STATUS_SUCCESS;

    __try
        {
        SpyCall (psci, psco);
        }
    __except (EXCEPTION_EXECUTE_HANDLER)
        {
        ns = STATUS_ACCESS_VIOLATION;
        }
    return ns;
    }
```

LISTING 6-3. *Adding Structured Exception Handling to the Kernel Call Interface*

Although SEH catches the most common parameter errors, you should not expect it to be a remedy against any garbage a client application might possibly deliver to a kernel API function. Some bad function arguments silently wreck the system without causing an exception. For example, a function that copies a string can easily overwrite vital parts of system memory if the destination buffer pointer is set to the wrong address. This kind of bug might remain undetected for a long time, until the system suddenly and unexpectedly breaks down when the program execution eventually rushes into the modified memory area. While testing the spy driver, I occasionally managed to get the test application hung in its IOCTL call to the spy device. The application didn't respond anymore and even refused to be removed from memory. Even worse, the system became unable to shut down. This is almost as annoying as a Blue Screen!

LINKING TO SYSTEM MODULES AT RUNTIME

After implementing the basic kernel call interface, the next problem is to resolve symbolic function names to linear addresses required in the ASM `CALL` instruction in Listing 6-2. This step is very important because you cannot be sure that the entry points of the various kernel API functions remain unchanged over a longer period. Whenever possible, functions should be called by name. Calling a system function by address is certainly exceptional, typically restricted to functions that are not exported by the target module. In most cases, it is more desirable to use the symbolic name, which is provided somewhere in the module's export section.

LOOKING UP NAMES EXPORTED BY A PE IMAGE

For a Win32 programmer, linking at runtime to a function exported by a DLL is an everyday task. For example, if you want to write a DLL that uses the enhanced features of Windows 2000, but also runs on legacy systems such as Windows 95 or 98 with reduced functionality, you should link to the special functions at runtime, silently falling back to default behavior if these functions aren't available. In this case, you would just call `GetModuleHandle()` if the DLL is already in memory and is guaranteed to stay there long enough, or `LoadLibrary()` if it has to be loaded or must be protected against premature unloading. The returned module handle can in turn be used in a sequence of `GetProcAddress()` calls that retrieve the entry points of all DLL functions the application wants to call. So it seems only logical to try the same with kernel functions exported by `ntoskrnl.exe`, `hal.dll`, or other system modules. However, neither of the above functions works in this situation! `GetModuleHandle()` reports that no such module is loaded, and `GetProcAddress()` returns `NULL` all the time if you pass in a hard-coded module handle, for example, `(HMODULE) 0x80400000` for `ntoskrnl.exe`. On second thought, this seems reasonable; these functions are designed for Win32 components that run in user-mode and therefore are loaded into the lower half of the 4-GB linear address space. Why should they care about kernel-mode components that are out of reach for Win32 applications anyway?

If the Win32 subsystem is ignorant about the modules in kernel memory, the next logical step is to let a kernel-mode driver do the work—the usual strategy applied throughout this book. The undocumented `MmGetSystemRoutineAddress()` function, exported by `ntoskrnl.exe`, obviously does the job, but, unfortunately, it isn't available on Windows NT 4.0. Because the main premise of this book's sample code is to remain compatible with the Windows 2000 predecessor to the greatest extent possible, I chose to reject this special feature looking up the function entries without the help of the system. The Windows 2000 runtime library provides some limited support for image file parsing, such as the undocumented `RtlImageNtHeader()` function, whose prototype is shown in Listing 6-4. This simple function takes the base address of a module image mapped to linear memory (i.e., a pointer to its `IMAGE_DOS_HEADER` structure, as defined in the Win32 SDK header file `winnt.h`) and returns a pointer to the `Portable PE` header referenced by the DOS header's `e_lfanew` member at file offset `0x3C`. This function must be used with care, because it performs only minimal sanity checks on the input pointer. It tests it for `NULL` and `0xFFFFFFFF` and verifies that the memory block it points to contains the `MZ` signature at the beginning. This means that if you pass in a bogus address that is neither `NULL` nor `0xFFFFFFFF`, a Blue Screen will be triggered immediately when `RtlImageNtHeader()` reads the DOS header signature. Oddly, Windows NT 4.0 runs this code in an SEH frame, whereas Windows 2000 doesn't.

```
PIMAGE_NT_HEADERS NTAPI RtlImageNtHeader (PVOID Base);
```

LISTING 6-4. *The Prototype of* RtlImageNtHeader()

Listing 6-4 shows that `RtlImageNtHeader()` returns a pointer to an `IMAGE_NT_HEADERS` structure. The entire set of PE file structures is defined in `winnt.h`. Unfortunately, the DDK header files do not have them, so it is necessary to add these definitions manually. My spy driver contains the structures it needs for symbol lookup (Listing 6-5) in its header file `w2k_spy.h`. `IMAGE_NT_HEADERS` is simply a concatenation of the PE signature "`PE\0\0`," an `IMAGE_FILE_HEADER`, and an `IMAGE_OPTIONAL_HEADER`. The latter ends with an array of `IMAGE_DATA_DIRECTORY` structures providing fast lookup of file sections with special duties. The first array entry, identified by the index `IMAGE_DIRECTORY_ENTRY_EXPORT` defined at the very beginning of Listing 6-5, points to the export section that contains the names and addresses of the functions exported by the module. This is the section where we must look up the function names passed to the kernel call interface.

```
#define IMAGE_DIRECTORY_ENTRY_EXPORT             0
#define IMAGE_DIRECTORY_ENTRY_IMPORT             1
#define IMAGE_DIRECTORY_ENTRY_RESOURCE           2
#define IMAGE_DIRECTORY_ENTRY_EXCEPTION          3
#define IMAGE_DIRECTORY_ENTRY_SECURITY           4
#define IMAGE_DIRECTORY_ENTRY_BASERELOC          5
#define IMAGE_DIRECTORY_ENTRY_DEBUG              6
#define IMAGE_DIRECTORY_ENTRY_COPYRIGHT          7
#define IMAGE_DIRECTORY_ENTRY_GLOBALPTR          8
#define IMAGE_DIRECTORY_ENTRY_TLS                9
#define IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG       10
#define IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT      11
#define IMAGE_DIRECTORY_ENTRY_IAT               12
#define IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT      13
#define IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR    14

#define IMAGE_NUMBEROF_DIRECTORY_ENTRIES        16

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

typedef struct _IMAGE_FILE_HEADER
    {
    WORD  Machine;
    WORD  NumberOfSections;
    DWORD TimeDateStamp;
    DWORD PointerToSymbolTable;
    DWORD NumberOfSymbols;
    WORD  SizeOfOptionalHeader;
    WORD  Characteristics;
    }
    IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

(continued)

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

typedef struct _IMAGE_DATA_DIRECTORY
    {
    DWORD VirtualAddress;
    DWORD Size;
    }
    IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;

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

typedef struct _IMAGE_OPTIONAL_HEADER
    {
    WORD                 Magic;
    BYTE                 MajorLinkerVersion;
    BYTE                 MinorLinkerVersion;
    DWORD                SizeOfCode;
    DWORD                SizeOfInitializedData;
    DWORD                SizeOfUninitializedData;
    DWORD                AddressOfEntryPoint;
    DWORD                BaseOfCode;
    DWORD                BaseOfData;
    DWORD                ImageBase;
    DWORD                SectionAlignment;
    DWORD                FileAlignment;
    WORD                 MajorOperatingSystemVersion;
    WORD                 MinorOperatingSystemVersion;
    WORD                 MajorImageVersion;
    WORD                 MinorImageVersion;
    WORD                 MajorSubsystemVersion;
    WORD                 MinorSubsystemVersion;
    DWORD                Win32VersionValue;
    DWORD                SizeOfImage;
    DWORD                SizeOfHeaders;
    DWORD                CheckSum;
    WORD                 Subsystem;
    WORD                 DllCharacteristics;
    DWORD                SizeOfStackReserve;
    DWORD                SizeOfStackCommit;
    DWORD                SizeOfHeapReserve;
    DWORD                SizeOfHeapCommit;
    DWORD                LoaderFlags;
    DWORD                NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory
                         [IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
    }
    IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;

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

typedef struct _IMAGE_NT_HEADERS
    {
```

```
    DWORD                 Signature;
    IMAGE_FILE_HEADER     FileHeader;
    IMAGE_OPTIONAL_HEADER OptionalHeader;
    }
    IMAGE_NT_HEADERS, *PIMAGE_NT_HEADERS;

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

typedef struct _IMAGE_EXPORT_DIRECTORY
    {
    DWORD Characteristics;
    DWORD TimeDateStamp;
    WORD  MajorVersion;
    WORD  MinorVersion;
    DWORD Name;
    DWORD Base;
    DWORD NumberOfFunctions;
    DWORD NumberOfNames;
    DWORD AddressOfFunctions;
    DWORD AddressOfNames;
    DWORD AddressOfNameOrdinals;
    }
    IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

LISTING 6-5. *A Subset of the Basic PE File Structures*

The layout of the export section inside a PE file is governed by the `IMAGE_EXPORT_DIRECTORY` structure, found at the bottom of Listing 6-5. Basically, it consists of a header composed of the members of the `IMAGE_EXPORT_DIRECTORY`, plus three variable-length arrays and a sequence of zero-terminated ANSI strings. An export item is usually identified by the following three parameters:

1. A zero-terminated symbolic name, consisting of 8-bit ANSI characters
2. A 16-bit ordinal number
3. A 32-bit target offset relative to the beginning of the file image

The export mechanism is not restricted to functions. It is merely a means to assign a symbol to an address inside the PE image. For functions, the symbol is attached to its entry point. For public variables, the symbol references its base address. The assignments are achieved by filling three parallel arrays with the characteristic parameters of the symbols. In Figure 6-1, these arrays are referred to as Array of Target Addresses, Array of Name Offsets, and Array of Ordinal Numbers. They correspond to the `IMAGE_EXPORT_DIRECTORY` members `AddressOfFunctions`, `AddressOfNames`, and `AddressOfNameOrdinals`, respectively, which supply the array offsets relative to the image base address. The `Name` member contains the offset of a symbol string that names the PE file itself. If the executable file is renamed, this entry can be used to retrieve its original name. Figure 6-1 is just a common example of an export section arrangement—the order of the arrays and the symbol string sub-section is not fixed. A PE file writer can shuffle them around to its liking, as long as the members of the `IMAGE_EXPORT_DIRECTORY` reference them correctly. The same is true for the string referenced by the `Name` member. Although it is usually located at the beginning of the name string sequence, this is not a requirement. Never rely on assumptions about the locations of the variable portions of the export section.

The `NumberOfFunctions` and `NumberOfNames` members of the `IMAGE_EXPORT_DIRECTORY` specify the number of entries in the `AddressOfFunctions` and `AddressOfNames` arrays, respectively. No count is specified for the `AddressOfNameOrdinals` array, because it always contains as many entries as the `AddressOfNames` array. The maintenance of separate entry counts for addresses and names suggests that it might be possible to build executables that export unnamed addresses. I have never seen such a file, but it is a good idea to keep this possibility in mind while accessing the arrays. Again, don't rely on assumptions!

The process of looking up the address of an exported function or variable by name requires the following steps, given a module base address (i.e., an `HMODULE` in Win32 lingo):

1. Call `RtlImageNtHeader()` with the module's base address to get at its `IMAGE_NT_HEADERS`. If this function returns `NULL`, the address does not reference a valid PE image.
2. Use the constant `IMAGE_DIRECTORY_ENTRY_EXPORT` as an index into the `DataDirectory` of the `OptionalHeader` member to find out the offset of the export section.
3. Locate the name array inside the export section by evaluating the `AddressOfNames` member of the `IMAGE_EXPORT_DIRECTORY` header.
4. Enumerate the names until a match is found or the end of the array indicated by `NumberOfNames` is reached.
5. If a matching name is available, use the name array index to read the associated ordinal number from the array of ordinals. The values in this array are zero-based, so you can use the name's ordinal immediately as an index into the address array.
6. Add the module's base address to the offset retrieved from the address array.

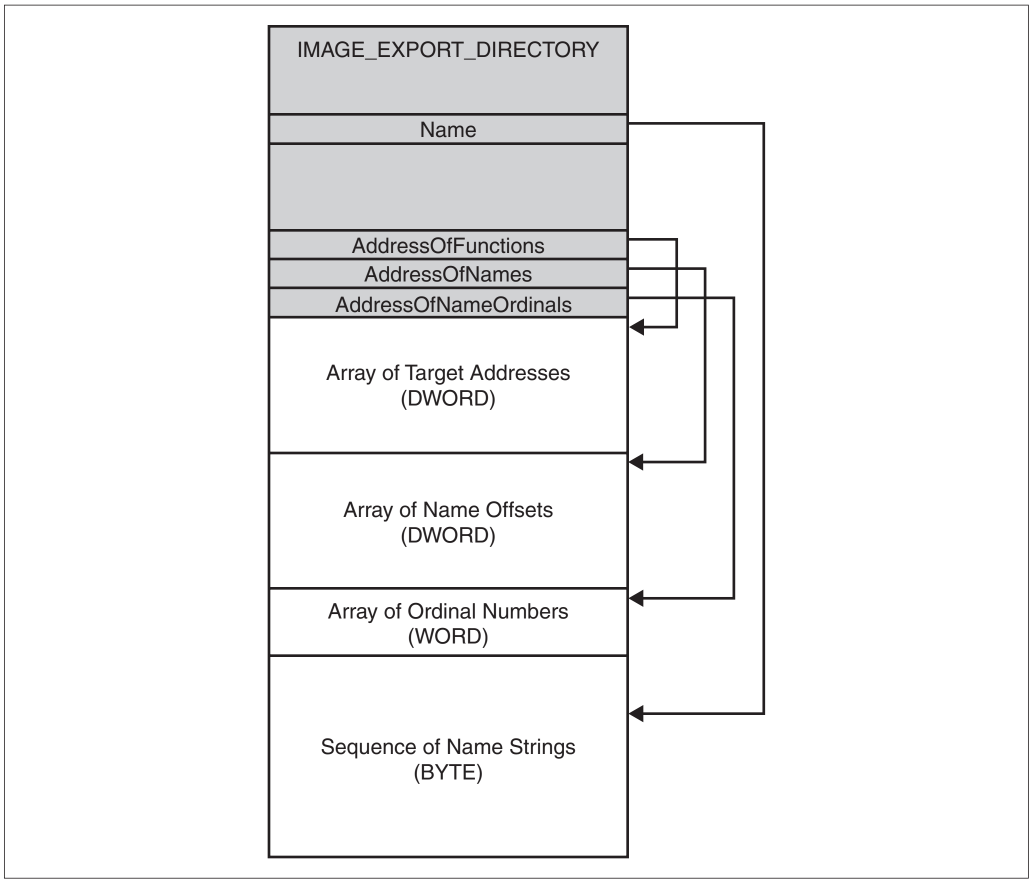
IMAGE_EXPORT_DIRECTORY
Name
AddressOfFunctions
AddressOfNames
AddressOfNameOrdinals
Array of Target Addresses (DWORD)
Array of Name Offsets (DWORD)
Array of Ordinal Numbers (WORD)
Sequence of Name Strings (BYTE)

FIGURE 6-1. *Typical Layout of a PE File's Export Section*

This sequence of steps appears fairly simple. However, it contains one unknown quantity: the module base address. Whereas the above actions basically reflect the behavior of the Win32 `GetProcAddress()` function, finding the module address means mimicking the behavior of `GetModuleHandle()`. If you scan the function names exported by `ntoskrnl.exe,` you won't be able to find anything that sounds even remotely like a function that might do the trick. The reason is that the Windows 2000 kernel provides a comprehensive function for this and many other tasks that involve access to internal system data. This function is called `NtQuerySystemInformation()`.

Locating System Modules and Drivers in Memory

`NtQuerySystemInformation()` is one of the most essential API functions for Windows 2000 system programmers, and there is hardly any built-in administration utility that does not make use of it—yet you won't find it mentioned anywhere in the Device Driver Kit (DDK) documentation. There is a single mention in the comments to the `CONFIGURATION_INFORMATION` structure inside `ntddk.h`, proving that this function exists, but that's it. If an "undocumentedness coefficient" would exist that were defined as the usefulness of a function divided by its frequency of occurrence in the Microsoft documentation, `NtQuerySystemInformation()` would certainly be ranked at the top. Along with many other wonderful things, this function can return a list of loaded system modules, including all system core components and kernel-mode drivers.

The spy driver source files contain the bare minimum of code and type definitions required to obtain the loaded-module list from `NtQuerySystemInformation()`. From the caller's point of view, it is a simple function. It expects four arguments, as shown in Listing 6-6. The `SystemInformationClass` is a numeric zero-based value that specifies the type of information to be queried. The information—which can be of variable length, depending on the information class—is copied to the `System Information` buffer supplied by the caller. The buffer length is specified by the `SystemInformationLength` argument. On success, the actual number of bytes copied to the buffer is written to the variable pointed to by `ReturnLength`. The problem with this function is that it doesn't report how many bytes it wanted to copy if it finds out that the buffer is too small. Thus, the caller must apply a trial-and-error heuristic until the returned status code changes from `STATUS_INFO_LENGTH_MISMATCH` (`0xC0000004`) to `STATUS_SUCCESS` (`0x00000000`).

Listing 6-6 doesn't show `NtQuerySystemInformation()` itself, but rather its twin, `ZwQuerySystemInformation()`, which is identical except for the function name prefix. You might recall from Chapter 2 that the `Nt*` and `Zw*` variants of the Native API functions work exactly the same if called from user-mode. The interface module `ntdll.dll` routes each pair through the same `INT 2Eh` stub. In kernel-mode, however, things are different. In this case, Native API calls are handled by `ntoskrnl.exe`, using different execution paths for `Nt*` and `Zw*` functions. The `Zw*` variants are again routed through the `INT 2Eh` interrupt gate, exactly as `ntdll.dll`. The `Nt*` variants, however,

```
NTSTATUS NTAPI ZwQuerySystemInformation (DWORD  SystemInformationClass,
                                         PVOID  SystemInformation,
                                         DWORD  SystemInformationLength,
                                         PDWORD ReturnLength);
```

Listing 6-6. *The Prototype of* `NtQuerySystemInformation()`

bypass this gate. In the glossary of the DDK documentation, Microsoft provides the following description for the `Zw*` function set (Microsoft 2000f):

> *"A set of entry points parallel to the executive's system services. A call to a* ***ZwX***xx *entry point from kernel-mode code (including calls from other system services or drivers) supplies the corresponding system service, except the caller's access rights and the arguments to the* ***Zw*** *'alias' are not checked for validity, and the call does not cause the previous mode to be set to user mode."* (Windows 2000 DDK \ Kernel-Mode Drivers \ Design Guide \ Kernel-Mode Glossary \ Z \ Zw routines.)

The last passage about the "previous mode" is important. Peter G. Viscarola and W. Anthony Mason put it in different, more clarifying words:

> *"Although either variant of the function may typically be called from Kernel mode, the* Zw *variant is used in place of the* Nt *version to cause the previous mode (and hence the mode in which the request was issued) to be set to Kernel mode." (Viscarola and Mason 1999, p. 18).*

The side effect of this previous-mode handling is that calling `NtQuerySystemInformation()` from a kernel-mode driver without any additional provisions returns an error status of `STATUS_ACCESS_VIOLATION` (`0xC0000005`), whereas `ZwQuerySystemInformation()` succeeds or at least returns `STATUS_INFO_LENGTH_MISMATCH`.

In Listing 6-7, the constant and type definitions required for the `SystemModuleInformation` class are shown. The list of loaded modules is returned in the form of a `MODULE_LIST` structure, composed of a 32-bit module count and an array of `MODULE_INFO` structures, one for each module.

```
#define SystemModuleInformation 11 // SYSTEMINFOCLASS

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

typedef struct _MODULE_INFO
    {
    DWORD dReserved1;
    DWORD dReserved2;
    PVOID pBase;
    DWORD dSize;
    DWORD dFlags;
    WORD  wIndex;
    WORD  wRank;
    WORD  wLoadCount;
```

(continued)

```
    WORD  wNameOffset;
    BYTE  abPath [MAXIMUM_FILENAME_LENGTH];
    }
    MODULE_INFO, *PMODULE_INFO, **PPMODULE_INFO;

#define MODULE_INFO_ sizeof (MODULE_INFO)

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

typedef struct _MODULE_LIST
    {
    DWORD       dModules;
    MODULE_INFO aModules [];
    }
    MODULE_LIST, *PMODULE_LIST, **PPMODULE_LIST;

#define MODULE_LIST_ sizeof (MODULE_LIST)
```

LISTING 6-7. `SystemModuleInformation` *Definitions*

Now everything is set up for a `ZwQuerySystemInformation()` call. Listing 6-8 contains the `SpyModuleList()` function that implements the usual trial-and-error loop required for this API function, along with two simple memory management functions, `SpyMemoryCreate()` and `SpyMemoryDestroy()`, that internally call the Windows 2000 Executive functions `ExAllocatePoolWithTag()` and `ExFreePool()`. The code starts out with a 4,096-byte buffer and doubles its size if the status code says `STATUS_INFO_LENGTH_MISMATCH`. All other status codes break the loop. The optional arguments `pdData` and `pns` provide more information about the returned value. If `SpyModuleList()` yields `NULL`, indicating failure, the `NTSTATUS` buffer pointed to by `pns` receives an error status code and `*pdData` is set to zero. On success, `*pdData` specifies the number of bytes copied to the buffer, and `*pns` reports `STATUS_SUCCESS`.

```
#define SPY_TAG '>YPS' // SPY> read backwards

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

PVOID SpyMemoryCreate (DWORD dSize)
    {
    return ExAllocatePoolWithTag (PagedPool, max (dSize, 1),
                                  SPY_TAG);
    }

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

```
PVOID SpyMemoryDestroy (PVOID pData)
    {
    if (pData != NULL) ExFreePool (pData);
    return NULL;
    }

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

PMODULE_LIST SpyModuleList (PDWORD    pdData,
                            PNTSTATUS pns)
    {
    DWORD        dSize;
    DWORD        dData = 0;
    NTSTATUS     ns    = STATUS_INVALID_PARAMETER;
    PMODULE_LIST pml   = NULL;

    for (dSize = PAGE_SIZE; (pml == NULL) && dSize; dSize <<= 1)
        {
        if ((pml = SpyMemoryCreate (dSize)) == NULL)
            {
            ns = STATUS_NO_MEMORY;
            break;
            }
        ns = ZwQuerySystemInformation (SystemModuleInformation,
                                       pml, dSize, &dData);
        if (ns != STATUS_SUCCESS)
            {
            pml   = SpyMemoryDestroy (pml);
            dData = 0;

            if (ns != STATUS_INFO_LENGTH_MISMATCH) break;
            }
        }
    if (pdData != NULL) *pdData = dData;
    if (pns    != NULL) *pns    = ns;
    return pml;
    }
```

LISTING 6-8. *Obtaining a module list from* `ZwQuerySystemInformation()`

The remaining actions to be taken to retrieve the base address of a given module are quite simple. Listing 6-9 defines two more functions: `SpyModuleFind()` is an enhanced `SpyModuleList()` wrapper that scans the module list returned by `ZwQuerySystemInformation()` for a specified module file name, and `SpyModuleBase()` in turn wraps `SpyModuleFind()`, extracting just the base address of the module in question from its `MODULE_INFO` and discarding the rest. The `SpyModuleHeader()` function concluding Listing 6-9 calls `SpyModuleBase()` and passes the result to `RtlImageNtHeader()`. This function provides the first step to the export section of a loaded module.

```
PMODULE_LIST SpyModuleFind (PBYTE     pbModule,
                            PDWORD    pdIndex,
                            PNTSTATUS pns)
    {
    DWORD        i;
    DWORD        dIndex = -1;
    NTSTATUS     ns     = STATUS_INVALID_PARAMETER;
    PMODULE_LIST pml    = NULL;

    if ((pml = SpyModuleList (NULL, &ns)) != NULL)
        {
        for (i = 0; i < pml->dModules; i++)
            {
            if (!_stricmp (pml->aModules [i].abPath +
                           pml->aModules [i].wNameOffset,
                           pbModule))
                {
                dIndex = i;
                break;
                }
            }
        if (dIndex == -1)
            {
            pml = SpyMemoryDestroy (pml);
            ns  = STATUS_NO_SUCH_FILE;
            }
        }
    if (pdIndex != NULL) *pdIndex = dIndex;
    if (pns     != NULL) *pns     = ns;
    return pml;
    }

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

PVOID SpyModuleBase (PBYTE     pbModule,
                     PNTSTATUS pns)
    {
    PMODULE_LIST pml;
    DWORD        dIndex;
    NTSTATUS     ns    = STATUS_INVALID_PARAMETER;
    PVOID        pBase = NULL;

    if ((pml = SpyModuleFind (pbModule, &dIndex, &ns)) != NULL)
        {
        pBase = pml->aModules [dIndex].pBase;
        SpyMemoryDestroy (pml);
        }
    if (pns != NULL) *pns = ns;
    return pBase;
    }

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

```
PIMAGE_NT_HEADERS SpyModuleHeader (PBYTE     pbModule,
                                   PPVOID    ppBase,
                                   PNTSTATUS pns)
    {
    PVOID             pBase = NULL;
    NTSTATUS          ns    = STATUS_INVALID_PARAMETER;
    PIMAGE_NT_HEADERS pinh  = NULL;

    if (((pBase = SpyModuleBase (pbModule, &ns)) != NULL) &&
        ((pinh  = RtlImageNtHeader (pBase))      == NULL))
        {
        ns = STATUS_INVALID_IMAGE_FORMAT;
        }
    if (ppBase != NULL) *ppBase = pBase;
    if (pns    != NULL) *pns    = ns;
    return pinh;
    }
```

LISTING 6-9. *Looking Up Information About a Specified Module*

RESOLVING SYMBOLS OF EXPORTED FUNCTIONS AND VARIABLES

The previous subsections explained how a PE file image is searched for a symbolic name of an exported function or variable and how the base address of a loaded system module or driver can be determined. Now it is time to put the loose ends together. Essentially, looking up a symbol exported by a given module is a three-step procedure:

1. Find out the linear base address of the module.
2. Search the export section of this module for the symbol.
3. Add the symbol offset to the module address.

The first step was discussed at some length above. Listing 6-10 provides the implementation details concerning the remaining steps. `SpyModuleExport()` expects a file name, such as `ntoskrnl.exe, hal.dll, ntfs.sys,` or similar, for the `pbModule` argument, and returns a pointer to the module's `IMAGE_EXPORT_DIRECTORY` structure, provided that the module is present in kernel memory and features an export section. The optional `ppBase` and `pns` arguments return additional information: `*ppBase` returns the module base address on success, and `*pns` reports a diagnostic error status on failure. First, `SpyModuleExport()` calls `SpyModuleHeader()` to locate the `IMAGE_NT_HEADERS;` then it evaluates the PE `DataDirectory` that contains the characteristic parameters of the export section in its first slot. If the

VirtualAddress member of this IMAGE_DATA_DIRECTORY entry (cf. Listing 6-5) is non-*NULL*, and the Size member states a reasonable value, the PE image contains an export section. In this case, SpyModuleExport() uses the PTR_ADD() macro included at the top of Listing 6-10 to add the module base address to the Virtual Address, yielding the absolute linear address of the IMAGE_EXPORT_DIRECTORY. Otherwise, it returns NULL and sets the status code to STATUS_DATA_ERROR (0xC000003E).

```
#define PTR_ADD(_base,_offset) \
        ((PVOID) ((PBYTE) (_base) + (DWORD) (_offset)))

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

PIMAGE_EXPORT_DIRECTORY SpyModuleExport (PBYTE     pbModule,
                                         PPVOID    ppBase,
                                         PNTSTATUS pns)
    {
    PIMAGE_NT_HEADERS       pinh;
    PIMAGE_DATA_DIRECTORY   pidd;
    PVOID                   pBase = NULL;
    NTSTATUS                ns    = STATUS_INVALID_PARAMETER;
    PIMAGE_EXPORT_DIRECTORY pied  = NULL;

    if ((pinh = SpyModuleHeader (pbModule, &pBase, &ns)) != NULL)
        {
        pidd = pinh->OptionalHeader.DataDirectory
               + IMAGE_DIRECTORY_ENTRY_EXPORT;

        if (pidd->VirtualAddress &&
            (pidd->Size >= IMAGE_EXPORT_DIRECTORY_))
            {
            pied = PTR_ADD (pBase, pidd->VirtualAddress);
            }
        else
            {
            ns = STATUS_DATA_ERROR;
            }
        }
    if (ppBase != NULL) *ppBase = pBase;
    if (pns    != NULL) *pns    = ns;
    return pied;
    }
```

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

PVOID SpyModuleSymbol (PBYTE     pbModule,
                       PBYTE     pbName,
                       PPVOID    ppBase,
                       PNTSTATUS pns)
    {
    PIMAGE_EXPORT_DIRECTORY pied;
    PDWORD                  pdNames, pdFunctions;
    PWORD                   pwOrdinals;
    DWORD                   i, j;
    PVOID                   pBase    = NULL;
    NTSTATUS                ns       = STATUS_INVALID_PARAMETER;
    PVOID                   pAddress = NULL;

    if ((pied = SpyModuleExport (pbModule, &pBase, &ns)) != NULL)
        {
        pdNames     = PTR_ADD (pBase, pied->AddressOfNames);
        pdFunctions = PTR_ADD (pBase, pied->AddressOfFunctions);
        pwOrdinals  = PTR_ADD (pBase, pied->AddressOfNameOrdinals);

        for (i = 0; i < pied->NumberOfNames; i++)
            {
            j = pwOrdinals [i];

            if (!strcmp (PTR_ADD (pBase, pdNames [i]), pbName))
                {
                if (j < pied->NumberOfFunctions)
                    {
                    pAddress = PTR_ADD (pBase, pdFunctions [j]);
                    }
                break;
                }
            }
        if (pAddress == NULL)
            {
            ns = STATUS_PROCEDURE_NOT_FOUND;
            }
        }
    if (ppBase != NULL) *ppBase = pBase;
    if (pns    != NULL) *pns    = ns;
    return pAddress;
    }
```

Listing 6-10. *Looking Up Symbols in a Module's Export Section*

SpyModuleSymbol() does the final work. Here you find the code that accesses the various items shown in Figure 6-1. After requesting an IMAGE_EXPORT_DIRECTORY pointer from SpyModuleExport(), the linear addresses of the address, name, and ordinal arrays are determined, again with the help of the PTR_ADD() macro. Fortunately, the PE file format specifies pointers to its internal data structures consistently as offsets from the base address of the image, so the PTR_ADD() macro constitutes a convenient general-purpose shortcut whenever a linear address must be computed from such an offset. It is important to note the role of the ordinal number array during address lookup. If the symbol has been found in the name array, the variable i contains the zero-based index of the array entry pointing to the symbol name. This value cannot be used as is to retrieve the associated address—it must be converted by means of the ordinal number array. The code line j = pwOrdinals [i]; does the trick. The resulting zero-based ordinal number j is the index that finally selects the correct address. Note that ordinal numbers are 16-bit quantities, whereas the other two arrays contain 32-bit numbers. If the symbol passed to SpyModuleSymbol() as its pbName argument cannot be resolved, a NULL pointer is returned, along with a status code of STATUS_PROCEDURE_NOT_FOUND (0xC000007A).

Although it looks like SpyModuleSymbol() provides everything we need to call kernel functions by name, I'm putting one more wrapper around it. Listing 6-11 shows the ultimate achievement: The function SpyModuleSymbolEx() takes a single string composed of a module/symbol pair in the form "module!symbol" and resolves it with the help of SpyModuleSymbol(). The largest part of the code is busy parsing the input string into a module name and a symbol. If no "!" separator is found, SpyModuleSymbolEx() assumes that ntoskrnl.exe is the target module, because this is certainly the most frequently used option.

```
PVOID SpyModuleSymbolEx (PBYTE     pbSymbol,
                         PPVOID    ppBase,
                         PNTSTATUS pns)
    {
    DWORD     i;
    BYTE      abModule [MAXIMUM_FILENAME_LENGTH] = "ntoskrnl.exe";
    PBYTE     pbName   = pbSymbol;
    PVOID     pBase    = NULL;
    NTSTATUS ns        = STATUS_INVALID_PARAMETER;
    PVOID     pAddress = NULL;

    for (i = 0; pbSymbol [i] && (pbSymbol [i] != '!'); i++);

    if (pbSymbol [i++])
        {
```

```
        if (i <= MAXIMUM_FILENAME_LENGTH)
            {
            strcpyn (abModule, pbSymbol, i);
            pbName = pbSymbol + i;
            }
        else
            {
            pbName = NULL;
            }
        }
    if (pbName != NULL)
        {
        pAddress = SpyModuleSymbol (abModule, pbName, &pBase, &ns);
        }
    if (ppBase != NULL) *ppBase = pBase;
    if (pns    != NULL) *pns    = ns;
    return pAddress;
    }
```

LISTING 6-11. *A Powerful Symbol Lookup Function*

THE BRIDGE TO USER-MODE

Now the evolution of the kernel call interface will slowly come to an end—at least as far as kernel-mode is concerned. Let me sum up what we have so far:

- A function named `SpyCallEx()` (Listing 6-3) that receives a `SPY_CALL_INPUT` control block containing a target address and some function arguments. It calls the specified address and returns any results in a `SPY_CALL_OUTPUT` control block.
- A mechanism to look up exported system functions and variables by name, represented by the function `SpyModuleSymbolEx()` (Listing 6-11).

So the last question is: "How do we make this stuff accessible to user-mode applications?" The answer is, of course: "Via Device I/O Control," as usual. To this end, the spy device provides a couple of IOCTL functions, summarized in Table 6-1. This is yet another excerpt from Table 4-2 in Chapter 4, which is a complete summary of all IOCTL functions offered by `w2k_spy.sys`. Listing 6-12 excerpts the relevant portions of the `SpyDispatcher()` function, which is shown in Listing 4-7 in Chapter 4.

The last row in Table 6-1 names the `SPY_IO_CALL` function that will serve as the bridge to user-mode. The remaining functions are there just for fun. I thought that once the spy device has access to this sort of valuable information, it would be nice to make it available to applications as well. As in Chapters 4 and 5, short descriptions of all newly introduced IOCTL functions follow.

TABLE 6-1. *IOCTL Functions Associated with the Kernel Call Interface*

FUNCTION NAME	ID	IOCTL CODE	DESCRIPTION
SPY_IO_MODULE_INFO	19	0x8000604C	Returns information about loaded system modules
SPY_IO_PE_HEADER	20	0x80006050	Returns IMAGE_NT_HEADERS data
SPY_IO_PE_EXPORT	21	0x80006054	Returns IMAGE_EXPORT_ DIRECTORY data
SPY_IO_PE_SYMBOL	22	0x80006058	Returns the address of an exported system symbol
SPY_IO_CALL	23	0x8000E05C	Calls a function inside a loaded module

```
NTSTATUS SpyDispatcher (PDEVICE_CONTEXT pDeviceContext,
                        DWORD           dCode,
                        PVOID           pInput,
                        DWORD           dInput,
                        PVOID           pOutput,
                        DWORD           dOutput,
                        PDWORD          pdInfo)
    {
    SPY_MEMORY_BLOCK smb;
    SPY_PAGE_ENTRY   spe;
    SPY_CALL_INPUT   sci;
    PHYSICAL_ADDRESS pa;
    DWORD            dValue, dCount;
    BOOL             fReset, fPause, fFilter, fLine;
    PVOID            pAddress;
    PBYTE            pbName;
    HANDLE           hObject;
    NTSTATUS         ns = STATUS_INVALID_PARAMETER;

    MUTEX_WAIT (pDeviceContext->kmDispatch);

    *pdInfo = 0;

    switch (dCode)
        {
```

```
// ===================================================
// unrelated IOCTL functions omitted (cf. Listing 4-7)
// ===================================================
        case SPY_IO_MODULE_INFO:
            {
            if ((ns = SpyInputPointer (&pbName,
                                       pInput, dInput))
                == STATUS_SUCCESS)
                {
                ns = SpyOutputModuleInfo (pbName,
                                          pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_PE_HEADER:
            {
            if ((ns = SpyInputPointer (&pAddress,
                                       pInput, dInput))
                == STATUS_SUCCESS)
                {
                ns = SpyOutputPeHeader (pAddress,
                                        pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_PE_EXPORT:
            {
            if ((ns = SpyInputPointer (&pAddress,
                                       pInput, dInput))
                == STATUS_SUCCESS)
                {
                ns = SpyOutputPeExport (pAddress,
                                        pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_PE_SYMBOL:
            {
            if ((ns = SpyInputPointer (&pbName,
                                       pInput, dInput))
                == STATUS_SUCCESS)
                {
                ns = SpyOutputPeSymbol (pbName,
                                        pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_CALL:
            {
```

(continued)

```
            if ((ns = SpyInputBinary (&sci, SPY_CALL_INPUT_,
                                      pInput, dInput))
                == STATUS_SUCCESS)
                {
                ns = SpyOutputCall (&sci,
                                    pOutput, dOutput, pdInfo);
                }
            break;
            }
// ====================================================
// unrelated IOCTL functions omitted (cf. Listing 4-7)
// ====================================================
        }
    MUTEX_RELEASE (pDeviceContext->kmDispatch);
    return ns;
    }
```

LISTING 6-12. *Excerpt from the Spy Driver's Hook Command Dispatcher*

THE IOCTL FUNCTION SPY_IO_MODULE_INFO

The IOCTL `SPY_IO_MODULE_INFO` function receives a module base address and sends back a `SPY_MODULE_INFO` structure if the address points to a valid PE image. The definition of this structure plus the related `SpyOutputModuleInfo()` helper function called by the `SpyDispatcher()` in Listing 6-12 are shown in Listing 6-13. `SpyOutputModuleInfo()` is based on `SpyModuleFind()` (Listing 6-9), which returns `MODULE_INFO` data obtained from `ZwQuerySystemInformation()`. The `MODULE_INFO` is converted to `SPY_MODULE_INFO` format and sent off to the caller.

```
typedef struct _SPY_MODULE_INFO
    {
    PVOID pBase;
    DWORD dSize;
    DWORD dFlags;
    DWORD dIndex;
    DWORD dLoadCount;
    DWORD dNameOffset;
    BYTE  abPath [MAXIMUM_FILENAME_LENGTH];
    }
    SPY_MODULE_INFO, *PSPY_MODULE_INFO, **PPSPY_MODULE_INFO;

#define SPY_MODULE_INFO_ sizeof (SPY_MODULE_INFO)

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

```
NTSTATUS SpyOutputModuleInfo (PBYTE  pbModule,
                              PVOID  pOutput,
                              DWORD  dOutput,
                              PDWORD pdInfo)
    {
    SPY_MODULE_INFO smi;
    PMODULE_LIST    pml;
    PMODULE_INFO    pmi;
    DWORD           dIndex;
    NTSTATUS        ns = STATUS_INVALID_PARAMETER;

    if ((pbModule != NULL) && SpyMemoryTestAddress (pbModule) &&
        ((pml = SpyModuleFind (pbModule, &dIndex, &ns)) != NULL))
        {
        pmi = pml->aModules + dIndex;

        smi.pBase       = pmi->pBase;
        smi.dSize       = pmi->dSize;
        smi.dFlags      = pmi->dFlags;
        smi.dIndex      = pmi->wIndex;
        smi.dLoadCount  = pmi->wLoadCount;
        smi.dNameOffset = pmi->wNameOffset;

        strcpyn (smi.abPath, pmi->abPath, MAXIMUM_FILENAME_LENGTH);

        ns = SpyOutputBinary (&smi, SPY_MODULE_INFO_,
                              pOutput, dOutput, pdInfo);

        SpyMemoryDestroy (pml);
        }
    return ns;
    }
```

LISTING 6-13. *Implementation of* `SPY_IO_MODULE_INFO`

THE IOCTL FUNCTION SPY_IO_PE_HEADER

The IOCTL `SPY_IO_PE_HEADER` function is merely an IOCTL wrapper for the `ntoskrnl.exe` API function `RtlImageNtHeader()`, as Listing 6-14 proves. Like `SPY_IO_MODULE_INFO`, it expects a module base address. The returned data is the module's `IMAGE_NT_HEADERS` structure.

```
NTSTATUS SpyOutputPeHeader (PVOID  pBase,
                            PVOID  pOutput,
                            DWORD  dOutput,
                            PDWORD pdInfo)
```

(continued)

```
    {
    PIMAGE_NT_HEADERS pinh;
    NTSTATUS          ns = STATUS_INVALID_PARAMETER;

    if ((pBase != NULL) && SpyMemoryTestAddress (pBase) &&
        ((pinh = RtlImageNtHeader (pBase)) != NULL))
        {
        ns = SpyOutputBinary (pinh, IMAGE_NT_HEADERS_,
                              pOutput, dOutput, pdInfo);
        }
    return ns;
    }
```

LISTING 6-14. *Implementation of* `SPY_IO_PE_HEADER`

THE IOCTL FUNCTION SPY_IO_PE_EXPORT

The IOCTL `SPY_IO_PE_EXPORT` function is more interesting than the previous one. In short, it returns the `IMAGE_EXPORT_DIRECTORY` associated with a module base address to the caller. A close look at its implementation in Listing 6-15 reveals a strong similarity to the `SpyModuleExport()` function in Listing 6-10. However, `SpyOutputPeExport()` does a lot of additional work. The reason for this is that the `IMAGE_EXPORT_DIRECTORY` contains relative addresses throughout, as explained earlier. The caller can't make much use of these offsets after the data has been copied to a separate buffer, because the base address to which the offsets relate has changed. Without additional address information from the PE header, it is impossible to compute a new matching base address. To save the caller from this excess work, `SpyOutputPeExport()` converts all offsets that point into the export section to offsets relative to the beginning of this section by subtracting its `VirtualAddress` specified in the `IMAGE_DATA_DIRECTORY`. The entries in the address array must be handled differently because they refer to other sections in the PE image. Therefore, `SpyOutputPeExport()` relocates them to absolute linear addresses by adding the image base address.

```
NTSTATUS SpyOutputPeExport (PVOID  pBase,
                            PVOID  pOutput,
                            DWORD  dOutput,
                            PDWORD pdInfo)
    {
    PIMAGE_NT_HEADERS       pinh;
    PIMAGE_DATA_DIRECTORY   pidd;
    PIMAGE_EXPORT_DIRECTORY pied;
    PVOID                   pData;
    DWORD                   dData, dBias, i;
    PDWORD                  pdData;
    NTSTATUS                ns = STATUS_INVALID_PARAMETER;

    if ((pBase != NULL) && SpyMemoryTestAddress (pBase) &&
        ((pinh = RtlImageNtHeader (pBase)) != NULL))
        {
        pidd = pinh->OptionalHeader.DataDirectory
               + IMAGE_DIRECTORY_ENTRY_EXPORT;

        if (pidd->VirtualAddress &&
            (pidd->Size >= IMAGE_EXPORT_DIRECTORY_))
            {
            pData = (PBYTE) pBase + pidd->VirtualAddress;
            dData = pidd->Size;

            if ((ns = SpyOutputBinary (pData, dData,
                                       pOutput, dOutput, pdInfo))
                == STATUS_SUCCESS)
                {
                pied  = pOutput;
                dBias = pidd->VirtualAddress;

                pied->Name                  -= dBias;
                pied->AddressOfFunctions    -= dBias;
                pied->AddressOfNames        -= dBias;
                pied->AddressOfNameOrdinals -= dBias;

                pdData = PTR_ADD (pied, pied->AddressOfFunctions);

                for (i = 0; i < pied->NumberOfFunctions; i++)
                    {
                    pdData [i] += (DWORD) pBase;
                    }
                pdData = PTR_ADD (pied, pied->AddressOfNames);

                for (i = 0; i < pied->NumberOfNames; i++)
                    {
                    pdData [i] -= dBias;
                    }
                }
            }
        else
            {
            ns = STATUS_DATA_ERROR;
            }
        }
    return ns;
    }
```

LISTING 6-15. *Implementation of* `SPY_IO_PE_EXPORT`

The IOCTL Function SPY_IO_PE_SYMBOL

The IOCTL SPY_IO_PE_SYMBOL function makes the symbol lookup engine of the kernel call interface accessible to user-mode applications. Its implementation, shown in Listing 6-16, isn't extraordinarily exciting, because it is an IOCTL wrapper for the SpyModuleSymbolEx() function in Listing 6-11. The caller must pass in a pointer to a string in the form "module!symbol," or simply "symbol" if the symbol should be looked up in the export section of ntoskrnl.exe, and gets back a pointer to the symbol's associated linear address, or NULL if the symbol is invalid or an error occurs.

The IOCTL Function SPY_IO_CALL

Finally, this is the IOCTL SPY_IO_CALL function we have been waiting for. Listing 6-17 provides the implementation details. This function calls SpyModuleSymbolEx() if the passed-in symbol string address is OK, and continues with SpyCallEx() if the symbol could be resolved. Like SPY_IO_PE_SYMBOL, this function expects the symbol name to be specified as "module!symbol" or simply "symbol," with the latter variant defaulting to ntoskrnl.exe. This time, however, the symbol string must be supplied as part of a properly initialized SPY_CALL_INPUT structure. On success, SPY_IO_CALL returns a SPY_CALL_OUTPUT structure containing either the result of the function call if the symbol refers to an API function or the value of the target variable if the symbol specifies a public variable such as NtBuildNumber or KeService DescriptorTable.

If SPY_IO_CALL fails, no data is returned. The caller must be prepared to handle this situation properly. Ignoring this error would mean returning bogus data from a kernel function call. If this data is passed in turn to another kernel function, problems may occur. If you are lucky, the faulty data is caught by the exception handler inside SpyCallEx(). If you are not so lucky, the entire process may hang persistently inside the spy device IOCTL call. As usual, however, there is a probability of a Blue Screen. But don't worry—the next section shows how the kernel call interface is properly used in user-mode applications.

```
NTSTATUS SpyOutputPeSymbol (PBYTE  pbSymbol,
                            PVOID  pOutput,
                            DWORD  dOutput,
                            PDWORD pdInfo)
    {
    PVOID    pAddress;
    NTSTATUS ns = STATUS_INVALID_PARAMETER;

    if ((pbSymbol != NULL) && SpyMemoryTestAddress (pbSymbol)
        &&
        ((pAddress = SpyModuleSymbolEx (pbSymbol, NULL, &ns))
         != NULL))
        {
        ns = SpyOutputPointer (pAddress,
                               pOutput, dOutput, pdInfo);
        }
    return ns;
    }
```

LISTING 6-16. *Implementation of* `SPY_IO_PE_SYMBOL`

```
NTSTATUS SpyOutputCall (PSPY_CALL_INPUT psci,
                        PVOID           pOutput,
                        DWORD           dOutput,
                        PDWORD          pdInfo)
    {
    SPY_CALL_OUTPUT sco;
    NTSTATUS        ns = STATUS_INVALID_PARAMETER;

    if (psci->pbSymbol != NULL)
        {
        psci->pEntryPoint =
            (SpyMemoryTestAddress (psci->pbSymbol)
             ? SpyModuleSymbolEx  (psci->pbSymbol, NULL, &ns)
             : NULL);
        }
    if ((psci->pEntryPoint != NULL)              &&
        SpyMemoryTestAddress (psci->pEntryPoint) &&
        ((ns = SpyCallEx (psci, &sco)) == STATUS_SUCCESS))
        {
        ns = SpyOutputBinary (&sco, SPY_CALL_OUTPUT_,
                              pOutput, dOutput, pdInfo);
        }
    return ns;
    }
```

LISTING 6-17. *Implementation of* `SPY_IO_CALL`

ENCAPSULATING THE CALL INTERFACE IN A DLL

Although it is good news that `w2k_spy.sys` exports an IOCTL call interface for kernel functions, this interface is somewhat clumsy to operate. Suppose you want to call a simple function such as `MmGetPhysicalAddress()` or `MmIsAddressValid()`. First, you must fill a `SPY_CALL_INPUT` structure with information about the function and its arguments. Next, you must issue a Win32 `DeviceIoControl()` call. If this function reports `ERROR_SUCCESS`, the returned `SPY_CALL_OUTPUT` structure must be evaluated. Otherwise, the error must be handled properly. Doesn't sound very appealing, does it? Fortunately, we have DLLs, so the solution to this problem is to hide the IOCTL mechanism in a DLL that does the dirty work. That's the purpose of the `w2k_call.dll` project included on this book's sample CD. The code snippets reprinted in this section are excerpts from the files `w2k_call.c` and `w2k_call.h`, found on the CD in the `\src\w2k_call` directory.

Handling IOCTL Function Calls

Before anything else, the `DeviceIoControl()` calls must be encapsulated in a convenient way, because this is the bottleneck through which all kernel function calls must pass. Listing 6-18 shows the wrapper function `w2kSpyControl()`, which contains a `DeviceIoControl()` invocation at its heart. Altogether, this function carries out the following tasks:

- Validates the input/output parameters
- Loads the spy device driver and opens the spy device, if not yet done
- Invokes `DeviceIoControl()`
- Tests the output data for the expected size
- Sets the Win32 last-error code appropriately

If successful, the system's last-error code, to be retrieved by the application via `GetLastError()`, is set to `ERROR_SUCCESS (0)`. Otherwise, it is set according to the following strategy:

- If the input or output parameters are invalid, the last-error value is `ERROR_INVALID_PARAMETER` (`87`), indicating "The parameter is incorrect" according to the `winerror.h` header file in the Platform Software Development Kit (SDK).
- If the spy device can't be initialized, the last-error value is `ERROR_GEN_FAILURE (31)`, indicating "A device attached to the system is not functioning."
- If the size of the data returned by the spy device doesn't match the caller's buffer size, the last-error value is `ERROR_DATATYPE_MISMATCH`, indicating "Data supplied is of wrong type."
- In all other cases, `w2kSpyControl()` preserves the last-error value set by the `DeviceIoControl()` function, whatever it might be. Usually, it will be the `NTSTATUS` returned by the spy device, but mapped to a more or less appropriate Win32 status code.

```
BOOL WINAPI w2kSpyControl (DWORD dCode,
                           PVOID pInput,
                           DWORD dInput,
                           PVOID pOutput,
                           DWORD dOutput)
    {
    DWORD dInfo = 0;
    BOOL  fOk   = FALSE;

    SetLastError (ERROR_INVALID_PARAMETER);

    if (((pInput  != NULL) || (!dInput )) &&
        ((pOutput != NULL) || (!dOutput)))
        {
        if (w2kSpyStartup (FALSE, NULL))
            {
            if (DeviceIoControl (ghDevice, dCode,
                                 pInput,   dInput,
                                 pOutput,  dOutput,
                                 &dInfo,   NULL))
                {
                if (dInfo == dOutput)
                    {
                    SetLastError (ERROR_SUCCESS);
                    fOk = TRUE;
                    }
                else
                    {
                    SetLastError (ERROR_DATATYPE_MISMATCH);
                    }
                }
            }
        else
            {
            SetLastError (ERROR_GEN_FAILURE);
            }
        }
    return fOk;
    }
```

LISTING 6-18. *The Basic* `DeviceIoControl()` *Wrapper*

The `w2kSpyStartup()` call in Listing 6-18, issued immediately before `DeviceIoControl()`, deserves some more attention. Because `w2k_call.dll` relies on the services of a kernel-mode driver, this driver must somehow be brought into memory before the first IOCTL transaction. Moreover, a device handle must be opened, identifying the target device to be accessed via `DeviceIoControl()`. To keep the DLL as flexible as possible, I opted for a mixed model in which the caller can either take full control of the loading/unloading and opening/closing of the spy device or rely on a default mechanism, leaving the device management responsibilities to the DLL. This automatism is quite simple: Loading the driver and opening the device is delayed until the first IOCTL transaction is requested. As soon as the DLL is unloaded, it automatically closes the device handle, but keeps the kernel-mode driver in memory. The latter decision constitutes a defensive strategy. As long as the caller doesn't supply any information as to how the driver should be handled, `w2k_call.dll` assumes that other clients might use the driver as well, so it can't unload the driver without impairing the operation of the other applications. As explained in Chapter 4 in the context of the memory spy application, the problem candidates are `not` the processes that still have open handles to the spy device. The Windows 2000 service control manager will delay the driver shutdown until all handles have been closed. The problem is that it won't allow any new device handles to be opened.

A `w2k_call.dll` client application can control the state of the spy device by means of the API function pair `w2kSpyStartup()` and `w2kSpyCleanup()`, shown in Listing 6-19. Because these functions might be called concurrently in a multithreading scenario, they use a critical-section object for serialization. Only one thread at a time can load/open or close/unload the spy device. If, for example, two threads call `w2kSpyStartup()` at approximately the same time, only one of them will be admitted to open the device. The other one is suspended, and will find the device up and running after resuming execution.

```
BOOL WINAPI w2kSpyLock (void)
    {
    BOOL fOk = FALSE;

    if (gpcs != NULL)
    {
        EnterCriticalSection (gpcs);
        fOk = TRUE;
        }
    return fOk;
    }

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

BOOL WINAPI w2kSpyUnlock (void)
    {
    BOOL fOk = FALSE;

    if (gpcs != NULL)
        {
        LeaveCriticalSection (gpcs);
        fOk = TRUE;
        }
    return fOk;
    }

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

BOOL WINAPI w2kSpyStartup (BOOL      fUnload,
                           HINSTANCE hInstance)
    {
    HINSTANCE hInstance1;
    SC_HANDLE hControl;
    BOOL      fOk = FALSE;

    w2kSpyLock ();

    hInstance1 = (hInstance != NULL ? hInstance : ghInstance);

    if ((ghDevice == INVALID_HANDLE_VALUE) &&
        w2kFilePath (hInstance1, awSpyFile, awDriver, MAX_PATH)
        &&
        ((hControl = w2kServiceLoad (awSpyDevice, awSpyDisplay,
                                     awDriver, TRUE))
         != NULL))
        {
        ghDevice = CreateFile (awSpyPath,
                               GENERIC_READ    | GENERIC_WRITE,
                               FILE_SHARE_READ | FILE_SHARE_WRITE,
                               NULL, OPEN_EXISTING,
                               FILE_ATTRIBUTE_NORMAL, NULL);

        if ((ghDevice == INVALID_HANDLE_VALUE) && fUnload)
            {
            w2kServiceUnload (awSpyDevice, hControl);
            }
        else
            {
            w2kServiceDisconnect (hControl);
            }
        }
    fOk = (ghDevice != INVALID_HANDLE_VALUE);

    w2kSpyUnlock ();
    return fOk;
    }

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

BOOL WINAPI w2kSpyCleanup (BOOL fUnload)
    {
```

(continued)

```
    BOOL fOk = FALSE;

    w2kSpyLock ();

    if (ghDevice != INVALID_HANDLE_VALUE)
        {
        CloseHandle (ghDevice);
        ghDevice = INVALID_HANDLE_VALUE;
        }
    if (fUnload)
        {
        w2kService Unload (awSpyDevice, NULL);
        }
    w2kSpyUnlock ();
    return fOk;
    }
```

LISTING 6-19. *The Spy Device Management Functions*

TYPE-SPECIFIC CALL INTERFACE FUNCTIONS

The `DeviceIoControl()` calls and the spy device management automatism have now been stowed in a set of functions, with `w2kSpyControl()` constituting their main entry point. The next step is to provide functions that perform `SPY_IO_CALLs` to the spy device. Listing 6-20 shows the basic implementation of the user-mode side of the kernel call interface, represented by the functions `w2kCallExecute(), w2kCall(),` and `w2kCallV()`. Regarding its input arguments, the former is the user-mode equivalent of `SpyCallEx(),` shown in Listing 6-3. In fact, the implementation of `w2kCallExecute()` shows that it calls the spy device's `SPY_IO_CALL` function via `w2kSpyControl()` after ensuring that the input control block contains either a symbol name string or an entry point address. From Listing 6-12, we know that `SPY_IO_CALL` is implemented by `SpyOutputCall()` (Listing 6-17), which in turn relies on `SpyModuleSymbolEx()` and `SpyCallEx()`.

```
BOOL WINAPI w2kCallExecute (PSPY_CALL_INPUT  psci,
                            PSPY_CALL_OUTPUT psco)
    {
    BOOL fOk = FALSE;

    SetLastError (ERROR_INVALID_PARAMETER);

    if (psco != NULL)
        {
        psco->uliResult.QuadPart = 0;
        if ((psci != NULL)
            &&
            ((psci->pbSymbol    != NULL) ||
             (psci->pEntryPoint != NULL)))
            {
            fOk = w2kSpyControl (SPY_IO_CALL,
                                 psci, SPY_CALL_INPUT_,
                                 psco, SPY_CALL_OUTPUT_);
            }
        }
    return fOk;
    }

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

BOOL WINAPI w2kCall (PULARGE_INTEGER puliResult,
                     PBYTE           pbSymbol,
                     PVOID           pEntryPoint,
                     BOOL            fFastCall,
                     DWORD           dArgumentBytes,
                     PVOID           pArguments)
    {
    SPY_CALL_INPUT  sci;
    SPY_CALL_OUTPUT sco;
    BOOL            fOk = FALSE;

    sci.fFastCall      = fFastCall;
    sci.dArgumentBytes = dArgumentBytes;
    sci.pArguments     = pArguments;
    sci.pbSymbol       = pbSymbol;
    sci.pEntryPoint    = pEntryPoint;

    fOk = w2kCallExecute (&sci, &sco);

    if (puliResult != NULL) *puliResult = sco.uliResult;
    return fOk;
    }

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

BOOL WINAPI w2kCallV (PULARGE_INTEGER puliResult,
                      PBYTE           pbSymbol,
                      BOOL            fFastCall,
                      DWORD           dArgumentBytes,
                      ...)
    {
    return w2kCall (puliResult, pbSymbol, NULL, fFastCall,
                    dArgumentBytes, &dArgumentBytes + 1);
    }
```

LISTING 6-20. *The Basic Call Interface Functions*

The SpyCall() and w2kCallV() functions in Listing 6-20 are the core functions of the kernel call interface inside w2k_call.dll, serving as a basis for several more specific functions. The main purpose of w2kCall() is to put the values of its arguments into a SPY_CALL_INPUT structure before calling w2kCallExecute() and to return the resulting ULARGE_INTEGER value. As explained earlier, not all bits of the result must be valid, depending on the result type of the called kernel function. w2kCallV() is a simple w2kCall() wrapper, featuring a variable argument list (hence the trailing V in the function name). Because the argument list of w2kCall() is tailored to the general case of kernel API invocations, it is overkill for many common function types. The most common type is the __stdcall (or NTAPI) function that returns an NTSTATUS value. In this case, the fFastCall argument is always FALSE and only the lower half of the returned 64-bit ULARGE_INTEGER contains valid data. Therefore, the w2kCallNT() function in Listing 6-21 does a much better job here. Please note how w2kCallNT() handles errors reported by w2kCall(). If w2kCall() returns FALSE, this means that w2kSpyControl() failed, indicating that the result of the function call is invalid. In this case, it would be nonsense to retrieve the LowPart value of the uliResult structure, because it contains unpredictable garbage. Therefore, w2kCallNT() defaults to STATUS_IO_DEVICE_ERROR (0xC0000185). After all, the caller must be prepared for return values other than STATUS_SUCCESS (0x00000000), so reporting this error code appears to be a reasonable decision. Other kernel functions that don't return NTSTATUS codes require a much more cautious selection of default return values in case of failure.

Listing 6-22 is a collection of five additional interface functions for __stdcall API functions that return the basic data types BYTE, WORD, DWORD, DWORDLONG, and PVOID. A trailing number in the function name indicates the number of significant return value bits. w2kCallP() is equivalent to w2kCall32(), except that the 32-bit return value is typecast to a pointer. It is not necessary to provide separate functions for the signed versions of the basic data types or for pointers to various types,

```
NTSTATUS WINAPI w2kCallNT (PBYTE pbSymbol,
                           DWORD dArgumentBytes,
                           ...)
    {
    ULARGE_INTEGER uliResult;

    return (w2kCall (&uliResult, pbSymbol, NULL, FALSE,
                     dArgumentBytes, &dArgumentBytes + 1)

            ? uliResult.LowPart
            : STATUS_IO_DEVICE_ERROR);
    }
```

LISTING 6-21. *A Simplified Interface for* NTAPI/NTSTATUS *Function Types*

because these smallish differences will be addressed by the automatic typecasting performed by the compiler. Note that all functions in Listing 6-22 expect a default return value to be passed in as the first argument. This is necessary because the call interface has no idea what value would be best to be returned if the call into kernel-mode fails, so this responsibility is up to the caller.

```
BYTE WINAPI w2kCall08 (BYTE  bDefault,
                       PBYTE pbSymbol,
                       BOOL  fFastCall,
                       DWORD dArgumentBytes,
                       ...)
    {
    ULARGE_INTEGER uliResult;

    return (w2kCall (&uliResult, pbSymbol, NULL, fFastCall,
                     dArgumentBytes, &dArgumentBytes + 1)

            ? (BYTE) uliResult.LowPart
            : bDefault);
    }

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

WORD WINAPI w2kCall16 (WORD  wDefault,
                       PBYTE pbSymbol,
                       BOOL  fFastCall,
                       DWORD dArgumentBytes,
                       ...)
    {
    ULARGE_INTEGER uliResult;

    return (w2kCall (&uliResult, pbSymbol, NULL, fFastCall,
                     dArgumentBytes, &dArgumentBytes + 1)

            ? (WORD) uliResult.LowPart
            : wDefault);
    }

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

DWORD WINAPI w2kCall32 (DWORD dDefault,
                        PBYTE pbSymbol,
                        BOOL  fFastCall,
                        DWORD dArgumentBytes,
                        ...)
    {
    ULARGE_INTEGER uliResult;
```

(continued)

```
    return (w2kCall (&uliResult, pbSymbol, NULL, fFastCall,
                     dArgumentBytes, &dArgumentBytes + 1)

            ? uliResult.LowPart
            : dDefault);
    }

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

QWORD WINAPI w2kCall64 (QWORD qDefault,
                        PBYTE pbSymbol,
                        BOOL  fFastCall,
                        DWORD dArgumentBytes,
                        ...)
    {
    ULARGE_INTEGER uliResult;

    return (w2kCall (&uliResult, pbSymbol, NULL, fFastCall,
                     dArgumentBytes, &dArgumentBytes + 1)

            ? uliResult.QuadPart
            : qDefault);
    }

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

PVOID WINAPI w2kCallP (PVOID pDefault,
                       PBYTE pbSymbol,
                       BOOL  fFastCall,
                       DWORD dArgumentBytes,
                       ...)
    {
    ULARGE_INTEGER uliResult;

    return (w2kCall (&uliResult, pbSymbol, NULL, fFastCall,
                     dArgumentBytes, &dArgumentBytes + 1)

            ? (PVOID) uliResult.LowPart
            : pDefault);
    }
```

LISTING 6-22. *More Interface Functions for Common Function Types*

DATA-COPYING INTERFACE FUNCTIONS

Before we get to the more interesting task of defining substitutes for a couple of real kernel API functions, some more lines of boilerplate code are required. I mentioned earlier that the kernel call interface of the spy device can also handle public variables exported by the kernel modules. In the description of Listing 6-2, where the

SpyCall() function was shown, I explained that a negative value for the argument stack size, supplied via the dArgumentBytes member of the SPY_CALL_INPUT structure, is interpreted as the one's complement of the size of an exported variable. In this case, SpyCall() doesn't call the specified entry point, but copies the appropriate number of bytes from this address to the result buffer. If dArgumentBytes is set to -1, yielding a one's complement of zero, the entry point address itself is copied to the buffer.

Listing 6-23 shows the data-copying functions exported by w2k_call.dll. This function set closely corresponds to the set of call interface functions in Listing 6-22. However, these functions require fewer input arguments. Copying the value of an exported variable requires no more than the name of the variable—no input parameters are required and no calling convention applies.

```
BOOL WINAPI w2kCopy (PULARGE_INTEGER puliResult,
                     PBYTE           pbSymbol,
                     PVOID           pEntryPoint,
                     DWORD           dBytes)
    {
    return w2kCall (puliResult, pbSymbol, pEntryPoint, FALSE,
                    0xFFFFFFFF - dBytes, NULL);
    }

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

BYTE WINAPI w2kCopy08 (BYTE  bDefault,
                       PBYTE pbSymbol)
    {
    ULARGE_INTEGER uliResult;

    return (w2kCopy (&uliResult, pbSymbol, NULL, 1)
            ? (BYTE) uliResult.LowPart
            : bDefault);
    }

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

WORD WINAPI w2kCopy16 (WORD  wDefault,
                       PBYTE pbSymbol)
    {
    ULARGE_INTEGER uliResult;

    return (w2kCopy (&uliResult, pbSymbol, NULL, 2)
            ? (WORD) uliResult.LowPart
            : wDefault);
    }

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

(continued)

```
DWORD WINAPI w2kCopy32 (DWORD dDefault,
                        PBYTE pbSymbol)
    {
    ULARGE_INTEGER uliResult;

    return (w2kCopy (&uliResult, pbSymbol, NULL, 4)
            ? uliResult.LowPart
            : dDefault);
    }

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

QWORD WINAPI w2kCopy64 (QWORD qDefault,
                        PBYTE pbSymbol)
    {
    ULARGE_INTEGER uliResult;

    return (w2kCopy (&uliResult, pbSymbol, NULL, 8)
            ? uliResult.QuadPart
            : qDefault);
    }

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

PVOID WINAPI w2kCopyP (PVOID pDefault,
                       PBYTE pbSymbol)
    {
    ULARGE_INTEGER uliResult;

    return (w2kCopy (&uliResult, pbSymbol, NULL, 4)
            ? (PVOID) uliResult.LowPart
            : pDefault);
    }

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

PVOID WINAPI w2kCopyEP (PVOID pDefault,
                        PBYTE pbSymbol)
    {
    ULARGE_INTEGER uliResult;

    return (w2kCopy (&uliResult, pbSymbol, NULL, 0)
            ? (PVOID) uliResult.LowPart
            : pDefault);
    }
```

LISTING 6-23. *Data-Copying Interface Functions for the Basic Data Types*

In Listing 6-23, `w2kCopy()` is the main workhorse, much like the `w2kCall()` function in case of a function invocation. Again, `w2k_call.dll` provides separate functions for the basic data types `BYTE, WORD, DWORD, DWORDLONG,` and `PVOID,` with a trailing number in the function name indicating the number of significant return value bits. `w2kCopyP()` returns a pointer value, and `w2kCopyEP()` handles the special case of querying an entry point address. Calling `w2kCopyEP()` is equivalent to calling the spy device's `SPY_IO_PE_SYMBOL` function. Yes, this `is` redundant, but having two alternative ways home is always better than none at all, isn't it?

IMPLEMENTING KERNEL API THUNKS

Meanwhile, the basic framework for the simple and easy implementation of kernel API function substitutes is available. I call these substitutes "thunks," which is the usual term in Windows lingo for a short piece of code that serves as a front-end to a function implemented in a different part of the system. Another common term is "proxy," but it is too tightly associated with the Microsoft Component Object Model (COM), so that using it here might be distracting. Let's start with two very simple Windows 2000 Memory Manager functions that have been my primary test objects during the development of the `w2k_call.dll` module: `MmGetPhysicalAddress()` and `MmIsAddressValid()`. Listing 6-24 shows how their thunks are implemented with the help of `w2kCall64()` and `w2kCall08()`. To avoid confusion with the original target functions, I am prefixing all thunk names with an underscore character.

```
PHYSICAL_ADDRESS WINAPI
_MmGetPhysicalAddress (PVOID BaseAddress)
    {
    PHYSICAL_ADDRESS pa;

    pa.QuadPart = w2kCall64 (0, "MmGetPhysicalAddress", FALSE,
                             4, BaseAddress);
    return pa;
    }

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

BOOLEAN WINAPI
_MmIsAddressValid (PVOID VirtualAddress)
    {
    return w2kCall08 (FALSE, "MmIsAddressValid", FALSE,
                      4, VirtualAddress);
    }
```

LISTING 6-24. *Sample Thunks for* `MmGetPhysicalAddress()` *and* `MmIsAddressValid()`

MmGetPhysicalAddress() receives a 32-bit linear address and returns a 64-bit PHYSICAL_ADDRESS structure, which is nothing but an alias for LARGE_INTEGER. Therefore, the thunk code calls w2kCall64(), indicating that 4 bytes are passed in on the argument stack, and putting the BaseAddress parameter on the list of arguments. The default value, to be returned in case of a fatal IOCTL error, is zero, which is the value that the original function returns on error. Because MmGetPhysicalAddress() uses the __stdcall convention, fFastCall is set to FALSE. The implementation of the MmIsAddressValid() thunk is similar, except that only the eight least significant bits of the SpyCallEx() result, corresponding to a BOOLEAN data type, are returned. The default return value is set to FALSE, which is a defensive choice. MmIsAddressValid() is typically called immediately before a memory access to avoid a potential page fault. Therefore, returning TRUE when the actual result of the function is indeterminable because of an IOCTL error would increase the risk of a Blue Screen.

That was easy. Now let's see how exported variables can be accessed in this framework. In Listing 6-25, two thunks, _NtBuildNumber() and _KeService DescriptorTable(), are shown. NtBuildNumber is exported by ntoskrnl.exe as a 16-bit WORD type, so the appropriate w2k_call.dll interface function is w2kCopy16(). The thunk returns zero in case of an error (if you can think of a more suitable value, please let me know). The _KeServiceDescriptorTable() thunk is a bit different, because the original KeServiceDescriptorTable address exported by ntoskrnl.exe points to a structure that comprises more than 64 bits. In this case, the best of the available options is to return the address of the KeServiceDescriptorTable itself, rather than reading an incomplete portion of the data it refers to. Therefore, the thunk makes use of the w2kCopyEP() helper function included in Listing 6-23.

You can imagine how excited I was when I realized that these thunks actually work! Then I thought: I'll try calling some very-low-level functions—that bang directly onto the hardware—that read and write I/O ports or the like. Fortunately, I had designed

```
WORD WINAPI
_NtBuildNumber (VOID)
    {
    return w2kCopy16 (0, "NtBuildNumber");
    }

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

PSERVICE_DESCRIPTOR_TABLE WINAPI
_KeServiceDescriptorTable (VOID)
    {
    return w2kCopyEP (NULL, "KeServiceDescriptorTable");
    }
```

LISTING 6-25. *Sample Thunks for* NtBuildNumber *and* KeServiceDescriptorTable

the `SpyModuleSymbolEx()` function in Listing 6-11 in a way that allows resolving symbols in `any` system module, including kernel-mode drivers. My next task was to call some functions exported by the Windows 2000 Hardware Abstraction Layer (HAL). After scanning the list of symbols contained in the export section of `hal.dll`, I decided to try two simple functions that are guaranteed to talk directly to the hardware: `HalMakeBeep()` and `HalQueryRealTimeClock()`. The `HalMakeBeep()` function reminded me of the old DOS days when it was possible to let the PC speaker squeak in many creative ways by programming some of the hardware chips on the motherboard. Actually, the implementation of `HalMakeBeep()` looks much like one of my old assembly language programs from 1987 that was able to play long sequences of music, given an array of tone pitches and durations. Operating the PC speaker involves programming a timer and a parallel I/O (PIO) chip at the I/O addresses `0x0042`, `0x0043`, and `0x0061`, so `HalMakeBeep()` was an ideal candidate for a first test of a thunk to a hardware-dependent function that would also guarantee immediate audible feedback.

Listing 6-26 shows the implementation of the `_HalMakeBeep()` thunk, an extraordinarily simple piece of code thanks to the `w2kCall08()` helper function. `HalMakeBeep()` starts a beep tone on the speaker with the requested pitch. If the pitch argument is set to zero, the beep is stopped. The function returns `TRUE` if the pitch value is valid, that is, zero or greater than 18. Note that the symbol string specified in the `w2kCall08()` call includes the name of the target module, which is `hal.dll` in this case. In Listings 6-24 and 6-25, no module was specified, because the symbols referenced there are exported by the default module `ntoskrnl.exe`.

Although `HalMakeBeep()` is a silly function, I was extremely happy to see the `_HalMakeBeep()` thunk working. The PC speaker beeped on my request! And this was Windows 2000, not DOS with this proof that a Win32 application can call a HAL function that does direct hardware access. I ported my old beep sequencer from DOS to Windows 2000, resulting in the code shown in Listing 6-27. `w2kBeep()` issues a single tone of the specified pitch and duration. `w2kBeepEx()` takes an array of pitch/duration values and plays them in sequence until coming across a zero-duration value. Both functions are exported by `w2k_call.dll`. Maybe you can use them to add musical background with a classic DOS feeling to your Win32 applications.

```
BOOLEAN WINAPI
_HalMakeBeep (DWORD Pitch)
    {
    return w2kCall08 (FALSE, "hal.dll!HalMakeBeep", FALSE,
                      4, Pitch);
    }
```

LISTING 6-26. *Thinking Down to* `HalMakeBeep()`

```
BOOL WINAPI w2kBeep (DWORD dDuration,
                     DWORD dPitch)
    {
    BOOL fOk = TRUE;

    if (!_HalMakeBeep (dPitch)) fOk = FALSE;
    Sleep (dDuration);
    if (!_HalMakeBeep (0     )) fOk = FALSE;
    return fOk;
    }

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

BOOL WINAPI w2kBeepEx (DWORD dData,
                       ...)
    {
    PDWORD pdData;
    BOOL   fOk = TRUE;

    for (pdData = &dData; pdData [0]; pdData += 2)
        {
        if (!w2kBeep (pdData [0], pdData [1])) fOk = FALSE;
        }
    return fOk;
    }
```

LISTING 6-27. *A Simple Beep Sequencer*

My next step was to try a more useful function, such as `HalQueryRealTime Clock()`. I remember that accessing the on-board real-time clock in a DOS application was at one time considered difficult. This involves reading and writing a couple of hardware I/O ports. Listing 6-28 shows the thunks to `HalQueryRealTimeClock()` and its sibling `HalSetRealTimeClock()`, along with the `TIME_FIELDS` structure on which both functions operate. The `TIME_FIELDS` structure is defined in `ntddk.h`.

```
typedef struct _TIME_FIELDS
    {
    SHORT Year;
    SHORT Month;
    SHORT Day;
    SHORT Hour;
    SHORT Minute;
    SHORT Second;
    SHORT Milliseconds;
    SHORT Weekday; // 0 = sunday
    }
    TIME_FIELDS, *PTIME_FIELDS;

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

#define TIME_FIELDS_ \
        sizeof (TIME_FIELDS)

VOID WINAPI
_HalQueryRealTimeClock (PTIME_FIELDS TimeFields)
    {
    w2kCallV (NULL, "hal.dll!HalQueryRealTimeClock", FALSE,
              4, TimeFields);
    return;
    }

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

VOID WINAPI
_HalSetRealTimeClock (PTIME_FIELDS TimeFields)
    {
    w2kCallV (NULL, "hal.dll!HalSetRealTimeClock", FALSE,
              4, TimeFields);
    return;
    }
```

LISTING 6-28. *Thunks for* `HalQueryRealTimeClock()` *and* `HalSetRealTimeClock()`

Listing 6-29 provides a typical application case of `_HalQueryRealTimeClock()`, displaying the current date and time in a console window.

```
VOID WINAPI DisplayTime (void)
    {
    TIME_FIELDS tf;

    _HalQueryRealTimeClock (&tf);

    printf (L"\r\nDate/Time: %02hd-%02hd-%04hd %02hd:%02hd:%02hd\r\n",
            tf.Month, tf.Day,    tf.Year,
            tf.Hour,  tf.Minute, tf.Second);
    return;
    }
```

LISTING 6-29. *Displaying the Current Date and Time*

Although it is great news that the kernel call interface works, it is also somewhat alarming. After all, we have been taught for years that Windows NT/2000 is a secure operating system where an application can't do anything it likes. The average Win32 programmer was cut off from the hardware. A more experienced NT programmer at least knew how to call Native API functions via `ntdll.dll`. An NT wizard was able to write kernel-mode drivers to do things that were not allowed in user-mode. Now, with the DLL presented here, all Win32 programmers are able to call arbitrary kernel functions just like any other Win32 API function. Is this a big security hole in the Windows 2000 kernel? No—the only 100% secure system is one that grants applications no access at all, which would be a useless system. As soon as there is a way to interact with the system, the system becomes vulnerable. And as soon as an operating system vendor allows third-party developers to add components to the system, it is possible to smuggle a direct bridge into the kernel, such as the `w2k_spy.sys` / `w2k_call.dll` pair. There is no such thing as a 100% secure system as long as the system interacts with its environment.

DATA ACCESS SUPPORT FUNCTIONS

I have added several dozen kernel API thunks to `w2k_call.dll`. For example, the entire set of string management functions exposed by the Windows 2000 runtime library is made available by this DLL. However, as you experiment with these predefined thunks or thunks that you have added yourself, you will find that calling kernel API functions from user-mode is a bit different from calling ordinary Win32 functions. The simplicity of the kernel call interface introduced here tends to obscure the fact that the calling application is still a user-mode program with limited privileges. For example, an application might call a kernel function that returns a pointer to a `UNICODE_STRING` structure. Most likely, this will be a pointer into kernel-mode memory, which is invisible to the calling application. Any attempts to access the string data will terminate the application with an exception, stating that the instruction at an address tried to read from a forbidden address. To solve this problem I have added support functions to `w2k_call.dll` that provide easy access to the most common types of data involved in kernel API calls.

The `w2kSpyRead()` function in Listing 6-30 is a general-purpose function that copies arbitrary memory data blocks to a caller-supplied buffer. It is based on the IOCTL function `SPY_IO_MEMORY_BLOCK` offered by the `w2k_spy.sys` spy device, briefly described in Chapter 4. Use this function to read the contents or individual members of structures allocated in kernel memory. It is important to note that `w2kSpyRead()` fails if the address range spanned by the memory block contains invalid addresses. "Invalid" means that neither physical nor pagefile memory is associated with this address. `w2kSpyClone()` is an enhanced version of `w2kSpyRead()` that automatically allocates a properly sized buffer and copies the kernel data to this buffer.

```
BOOL WINAPI w2kSpyRead (PVOID pBuffer,
                        PVOID pAddress,
                        DWORD dBytes)
    {
    SPY_MEMORY_BLOCK smb;
    BOOL             fOk = FALSE;

    if ((pBuffer != NULL) && (pAddress != NULL) && dBytes)
        {
        ZeroMemory (pBuffer, dBytes);

        smb.pAddress = pAddress;
        smb.dBytes   = dBytes;

        fOk = w2kSpyControl (SPY_IO_MEMORY_BLOCK,
                             &smb,    SPY_MEMORY_BLOCK_,
                             pBuffer, dBytes);
        }
    return fOk;
    }

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

PVOID WINAPI w2kSpyClone (PVOID pAddress,
                          DWORD dBytes)
    {
    PVOID pBuffer = NULL;

    if ((pAddress != NULL) && dBytes &&
        ((pBuffer = w2kMemoryCreate (dBytes)) != NULL) &&
        (!w2kSpyRead (pBuffer, pAddress, dBytes)))
        {
        pBuffer = w2kMemoryDestroy (pBuffer);
        }
    return pBuffer;
    }
```

LISTING 6-30. *General-Purpose Data Access Functions*

Reading strings requires a bit more work. Please recall that the most common string type used by kernel-mode components is the UNICODE_STRING structure, comprising a string buffer pointer and information about the buffer size and the number of bytes currently occupied by the string. Reading a UNICODE_STRING is usually a two-part task. First, the UNICODE_STRING structure must be copied to find out the size and address of the string buffer. In a second step, the string data is read. To simplify this common task, w2k_call.dll provides the function set contained in Listing 6-31. w2kStringAnsi() and w2kStringUnicode() allocate and initialize empty ANSI_STRING and UNICODE_STRING structures, respectively, including a string buffer of the specified size. For reasons of simplicity, the string header and buffer are integrated into a single memory block. These structures can be used as targets for string copying, as demonstrated by `w2kStringClone()`. This function creates a faithful copy of a `UNICODE_STRING` in user-mode memory. The `MaximumLength` of the copy is usually equal to the original, except if the source string has inconsistent parameters. For example, if the indicated `MaximumLength` is less than or equal to the value of the `Length` member, it is invalid and therefore is set to `Length+2`. However, the `MaximumLength` of the copy will never be smaller than the original `MaximumLength`.

```
PANSI_STRING WINAPI w2kStringAnsi (DWORD dSize)
    {
    PANSI_STRING pasData = NULL;

    if ((pasData = w2kMemoryCreate (ANSI_STRING_ + dSize))
        != NULL)
        {
        pasData->Length        = 0;
        pasData->MaximumLength = (WORD) dSize;
        pasData->Buffer        = PTR_ADD (pasData, ANSI_STRING_);

        if (dSize) pasData->Buffer [0] = 0;
        }
    return pasData;
    }

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

PUNICODE_STRING WINAPI w2kStringUnicode (DWORD dSize)
    {
    DWORD           dSize1  = dSize * WORD_;
    PUNICODE_STRING pusData = NULL;

    if ((pusData = w2kMemoryCreate (UNICODE_STRING_ + dSize1))
        != NULL)
        {
        pusData->Length        = 0;
        pusData->MaximumLength = (WORD) dSize1;
        pusData->Buffer        = PTR_ADD (pusData, UNICODE_STRING_);

        if (dSize) pusData->Buffer [0] = 0;
        }
    return pusData;
    }

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

```
PUNICODE_STRING WINAPI w2kStringClone (PUNICODE_STRING pusSource)
    {
    DWORD           dSize;
    UNICODE_STRING  usCopy;
    PUNICODE_STRING pusData = NULL;

    if (w2kSpyRead (&usCopy, pusSource, UNICODE_STRING_))
        {
        dSize = max (usCopy.Length + WORD_,
                     usCopy.MaximumLength) / WORD_;

        if (((pusData = w2kStringUnicode (dSize)) != NULL) &&
            usCopy.Length && (usCopy.Buffer != NULL))
            {
            if (w2kSpyRead (pusData->Buffer, usCopy.Buffer,
                                             usCopy.Length))
                {
                pusData->Length = usCopy.Length;
                pusData->Buffer  [usCopy.Length / WORD_] = 0;
                }
            else
                {
                pusData = w2kMemoryDestroy (pusData);
                }
            }
        }
    return pusData;
    }
```

LISTING 6-31. *String Management Functions*

Another way of copying a kernel string down to the application memory space is to use one of the kernel runtime functions. For example, you can use a combination of the `_RtlInitUnicodeString()` and `_RtlCopyUnicodeString()` thunks provided by `w2k_call.dll` to achieve a similar effect. However, calling `w2kStringClone()` is usually easier, because this function automatically allocates the memory required for the string copy.

ACCESSING NONEXPORTED SYMBOLS

What we have achieved so far is to enable an application to execute operations that formerly were reserved to kernel-mode drivers. Can we enhance an application with capabilities that not even a kernel-mode driver has? Can we call internal functions that are neither documented nor exported? This sounds dangerous, but, as I will show in this section, it is not as bad as it might seem, if handled with care.

LOOKING UP INTERNAL SYMBOLS

The kernel call interface described in the previous sections delegated the task of looking up the addresses of exported symbols to the spy device, which has full access to the PE images of the kernel modules residing in the upper half of the linear address space. However, if the function to be called or the global variable to be accessed is not exported, the spy device has no chance to find out its address. While writing this chapter and examining some disassembly listing emitted by the Kernel Debugger, I frequently thought: "What a pity that they don't export this nifty function!" What made me especially angry was that the Kernel Debugger showed me the exact function name, but my application code was absolutely ignorant of it. Of course, I could have used my kernel call interface to jump through the plain binary entry point of the function, but that's not good programming style. The next service pack might shift this entry point to a completely different address.

I reasoned that if the Debugger can do it, my application also should be able to do it. A sample DLL described in Chapter 1 put me on the right track. The `w2k_img.dll` provides everything needed to look up the address of any symbol defined by the Windows 2000 kernel modules, provided that the operating system's symbol files are properly installed. So I extended the `w2k_call.dll` by an API function that first resolves an internal symbol to its linear address and then uses `w2kCall()` to execute it. Of course, an analogous function is provided for global variables.

Listing 6-32 shows the complete set of extended call interface functions. Again, a separate convenience function is provided for each major function type, corresponding to the functions in Listings 6-20 to 6-22. `w2kXCall()` is the main workhorse. It calls the `w2k_img.dll` API function `imgTableResolve()` to retrieve the address of the supplied symbol and, if successful, specifies it in a subsequent invocation of `w2kCall()`. Because `w2kCall()` is supposed to call an address instead of a symbol, a `NULL` pointer is passed in for its `pbSymbol` argument. The `pEntryPoint` argument is set to the symbol address `pie->pAddress` just retrieved from the symbol files. As explained in Chapter 1, `w2k_img.dll` is able to determine the calling conventions of most internal functions, so the `fFastCall` argument can be set up automatically by testing the value of `pie->dConvention` for `IMG_CONVENTION_FASTCALL`. The number of argument bytes and the pointer to the arguments are forwarded as received from the caller. It would have been possible to retrieve the number of arguments from the symbol information as well, but this works with `__stdcall` and `__fastcall` functions only. `__cdecl` symbols don't encode the argument stack size in their decoration.

```
BOOL WINAPI w2kXCall (PULARGE_INTEGER puliResult,
                      PBYTE           pbSymbol,
                      DWORD           dArgumentBytes,
                      PVOID           pArguments)
    {
    PIMG_TABLE pit;
    PIMG_ENTRY pie;
    BOOL       fOk = FALSE;

    if (((pit = w2kSymbolsGlobal (NULL))         != NULL) &&
        ((pie = imgTableResolve (pit, pbSymbol)) != NULL) &&
        (pie->pAddress != NULL))
        {
        fOk = w2kCall (puliResult, NULL, pie->pAddress,
                       pie->dConvention == IMG_CONVENTION_FASTCALL,
                       dArgumentBytes, pArguments);
        }
    else
        {
        if (puliResult != NULL) puliResult->QuadPart = 0;
        }
    return fOk;
    }

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

BOOL WINAPI w2kXCallV (PULARGE_INTEGER puliResult,
                       PBYTE           pbSymbol,
                       DWORD           dArgumentBytes,
                       ...)
    {
    return w2kXCall (puliResult, pbSymbol,
                     dArgumentBytes, &dArgumentBytes + 1);
    }

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

NTSTATUS WINAPI w2kXCallNT (PBYTE pbSymbol,
                            DWORD dArgumentBytes,
                            ...)
    {
    ULARGE_INTEGER uliResult;

    return (w2kXCall (&uliResult, pbSymbol,
                      dArgumentBytes, &dArgumentBytes + 1)

            ? uliResult.LowPart
            : STATUS_IO_DEVICE_ERROR);
    }
```

(continued)

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

BYTE WINAPI w2kXCall08 (BYTE  bDefault,
                        PBYTE pbSymbol,
                        DWORD dArgumentBytes,
                        ...)
    {
    ULARGE_INTEGER uliResult;

    return (w2kXCall (&uliResult, pbSymbol,
                      dArgumentBytes, &dArgumentBytes + 1)

            ? (BYTE) uliResult.LowPart
            : bDefault);
    }

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

WORD WINAPI w2kXCall16 (WORD  wDefault,
                        PBYTE pbSymbol,
                        DWORD dArgumentBytes,
                        ...)
    {
    ULARGE_INTEGER uliResult;

    return (w2kXCall (&uliResult, pbSymbol,
                      dArgumentBytes, &dArgumentBytes + 1)

            ? (WORD) uliResult.LowPart
            : wDefault);
    }

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

DWORD WINAPI w2kXCall32 (DWORD dDefault,
                         PBYTE pbSymbol,
                         DWORD dArgumentBytes,
                         ...)
    {
    ULARGE_INTEGER uliResult;

    return (w2kXCall (&uliResult, pbSymbol,
                      dArgumentBytes, &dArgumentBytes + 1)

            ? uliResult.LowPart
            : dDefault);
    }

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

```
QWORD WINAPI w2kXCall64 (QWORD qDefault,
                         PBYTE pbSymbol,
                         DWORD dArgumentBytes,
                         ...)
    {
    ULARGE_INTEGER uliResult;

    return (w2kXCall (&uliResult, pbSymbol,
                      dArgumentBytes, &dArgumentBytes + 1)

            ? uliResult.QuadPart
            : qDefault);
    }

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

PVOID WINAPI w2kXCallP (PVOID pDefault,
                        PBYTE pbSymbol,
                        DWORD dArgumentBytes,
                        ...)
    {
    ULARGE_INTEGER uliResult;

    return (w2kXCall (&uliResult, pbSymbol,
                      dArgumentBytes, &dArgumentBytes + 1)

            ? (PVOID) uliResult.LowPart
            : pDefault);
    }
```

LISTING 6-32. *The Extended Call Interface*

Note in Listing 6-32 that `w2kXCall()` invokes `w2kSymbolsGlobal()` before doing anything else. This function is included in Listing 6-33, along with some helpers, and its purpose is to load the `ntoskrnl.exe` symbol as soon as the first `w2kXCall()` is executed. The table is stored in the global `PIMG_TABLE` variable named `gpit,` so subsequent calls can reuse it. With support of some helper functions, `w2kSymbolsLoad()` returns one of the status codes listed in Table 6-2 via the optional `*pdStatus` argument. To avoid jumping to an invalid address because of unmatched symbol information, `w2kSymbolsLoad()` carefully checks the time stamp and check sum of the symbol files against the corresponding fields in the memory-resident image of the target module using the `w2kPeCheck()` API function (not reprinted) and discards the symbol table if they don't match exactly.

```
PIMG_TABLE WINAPI w2kSymbolsLoad (PBYTE  pbModule,
                                  PDWORD pdStatus)
    {
    PVOID      pBase;
    DWORD      dStatus = W2K_SYMBOLS_UNDEFINED;
    PIMG_TABLE pit     = NULL;

    if ((pBase = imgModuleBaseA (pbModule)) == NULL)
        {
        dStatus = W2K_SYMBOLS_MODULE_NOT_FOUND;
        }
    else
        {
        if ((pit = imgTableLoadA (pbModule, pBase)) == NULL)
            {
            dStatus = W2K_SYMBOLS_LOAD_ERROR;
            }
        else
            {
            if (!w2kPeCheck (pbModule, pit->dTimeStamp,
                                       pit->dCheckSum))
                {
                dStatus = W2K_SYMBOLS_CHECKSUM_ERROR;
                pit     = imgMemoryDestroy (pit);
                }
            else
                {
                dStatus = W2K_SYMBOLS_OK;
                }
            }
        }
    if (pdStatus != NULL) *pdStatus = dStatus;
    return pit;
    }

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

PIMG_TABLE WINAPI w2kSymbolsGlobal (PDWORD pdStatus)
    {
    DWORD      dStatus = W2K_SYMBOLS_UNDEFINED;
    PIMG_TABLE pit     = NULL;

    w2kSpyLock ();

    if ((gdStatus == W2K_SYMBOLS_OK) && (gpit == NULL))
        {
            gpit = w2kSymbolsLoad (NULL, &gdStatus);
            }
        dStatus = gdStatus;
        pit     = gpit;

        w2kSpyUnlock ();

        if (pdStatus != NULL) *pdStatus = dStatus;
        return pit;
        }

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

DWORD WINAPI w2kSymbolsStatus (VOID)
    {
    DWORD dStatus = W2K_SYMBOLS_UNDEFINED;

    w2kSymbolsGlobal (&dStatus);
    return dStatus;
    }

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

VOID WINAPI w2kSymbolsReset (VOID)
    {
    w2kSpyLock ();

    gpit     = imgMemoryDestroy (gpit);
    gdStatus = W2K_SYMBOLS_OK;

    w2kSpyUnlock ();
    return;
    }
```

LISTING 6-33. *The Symbol Table Manager Functions*

The `w2kSymbolsStatus()` and `w2kSymbolsReset()` functions at the bottom of Listing 6-33 are used to load and unload the symbol table on demand. `w2kSymbolsStatus()` attempts to load the symbol table if it isn't already present and returns its status. If `w2k_call.dll` already tried to load the table without success, the function simply returns the last error status (Table 6-2) unless the symbol table is reset by a `w2kSymbolsReset()` call. The latter function also destroys the memory block occupied by the symbol table, if any, forcing a complete symbol reload on the next request that involves the `ntoskrnl.exe` symbol table.

TABLE 6-2. *w2kSymbolsLoad() Status Codes*

STATUS CODE	DESCRIPTION
W2K_SYMBOLS_OK	The module's symbol table has been loaded
W2K_SYMBOLS_MODULE_ERROR	The module is not resident in memory
W2K_SYMBOLS_LOAD_ERROR	The module's symbol files couldn't be loaded
W2K_SYMBOLS_VERSION_ERROR	The symbol files don't match the resident module image
W2K_SYMBOLS_UNDEFINED	The symbol table status is undefined

The `w2kXCopy*()` function set making up the extended copy interface is shown in Listing 6-34, which corresponds to Listing 6-23 above. `w2kXCopy()` simply calls `w2kXCall()` with a negative value for `dArgumentBytes`, and the remaining copy functions are merely wrappers with simplified argument lists.

```
BOOL WINAPI w2kXCopy (PULARGE_INTEGER puliResult,
                      PBYTE           pbSymbol,
                      DWORD           dBytes)
    {
    return w2kXCall (puliResult, pbSymbol,
                     0xFFFFFFFF - dBytes, NULL);
    }

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

BYTE WINAPI w2kXCopy08 (BYTE  bDefault,
                        PBYTE pbSymbol)
    {
    ULARGE_INTEGER uliResult;

    return (w2kXCopy (&uliResult, pbSymbol, 1)
            ? (BYTE) uliResult.LowPart
            : bDefault);
    }

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

WORD WINAPI w2kXCopy16 (WORD  wDefault,
                        PBYTE pbSymbol)
    {
    ULARGE_INTEGER uliResult;

    return (w2kXCopy (&uliResult, pbSymbol, 2)
            ? (WORD) uliResult.LowPart
            : wDefault);
    }
```

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

DWORD WINAPI w2kXCopy32 (DWORD dDefault,
                         PBYTE pbSymbol)
    {
    ULARGE_INTEGER uliResult;

    return (w2kXCopy (&uliResult, pbSymbol, 4)
            ? uliResult.LowPart
            : dDefault);
    }

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

QWORD WINAPI w2kXCopy64 (QWORD qDefault,
                         PBYTE pbSymbol)
    {
    ULARGE_INTEGER uliResult;

    return (w2kXCopy (&uliResult, pbSymbol, 8)
            ? uliResult.QuadPart
            : qDefault);
    }

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

PVOID WINAPI w2kXCopyP (PVOID pDefault,
                        PBYTE pbSymbol)
    {
    ULARGE_INTEGER uliResult;

    return (w2kXCopy (&uliResult, pbSymbol, 4)
            ? (PVOID) uliResult.LowPart
            : pDefault);
    }

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

PVOID WINAPI w2kXCopyEP (PVOID pDefault,
                         PBYTE pbSymbol)
    {
    ULARGE_INTEGER uliResult;

    return (w2kXCopy (&uliResult, pbSymbol, 0)
            ? (PVOID) uliResult.LowPart
            : pDefault);
    }
```

LISTING 6-34. *The Extended Copy Interface*

IMPLEMENTING KERNEL FUNCTION THUNKS

The same guidelines apply to the implementation of thunks for internal kernel functions as for exported API functions, except that only functions inside `ntoskrnl.exe` can be called. This restriction is imposed by the symbol table manager inside `w2k_call.dll`, not by the call interface itself. To simplify matters, only the `ntoskrnl.exe` symbol table is loaded, because this is the module where the most interesting symbols are found (of course, `w2k_call.dll` could have been enhanced to load multiple tables on request). Listing 6-35 comprises two sample thunks for internal functions of the Windows 2000 object manager that return information about type objects (object types will be discussed in detail in Chapter 7).

Listing 6-36 shows three thunks for some very important internal data structures that will be used by the sample code in Chapter 7. Note that I have prefixed the names of all thunks that use the extended kernel call interface with two underscores. This is just a reminder that this function will work only with a proper set of symbol files. If you install a service pack without also updating the symbol files, `w2kSymbolsLoad()` will refuse to load any symbols and the thunks will fail and return default values. On the other hand, the thunks with a single leading underscore should continue to work with unmatched symbol files, because they resolve symbols on the basis of the memory-resident export tables of the new modules. However, they may fail as well after an update if the updated modules fail to export all referenced API functions or some argument lists have been changed.

```
NTSTATUS WINAPI
__ObQueryTypeInfo (POBJECT_TYPE      ObjectType,
                   POBJECT_TYPE_INFO TypeInfo,
   /* bytes     */ DWORD             TypeInfoLength,
   /* init to 0 */ PDWORD            ReturnLength)
    {
    return w2kXCallNT ("ObQueryTypeInfo",
                       16, ObjectType, TypeInfo, TypeInfoLength,
                           ReturnLength);
    }

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

NTSTATUS WINAPI
__ObQueryTypeName (POBJECT                  Object,
                   POBJECT_NAME_INFORMATION NameString,
       /* bytes */ DWORD                    NameStringLength,
                  PDWORD                  ReturnLength)
    {
    return w2kXCallNT ("ObQueryTypeName",
                      16, Object, NameString, NameStringLength,
                          ReturnLength);
    }
```

LISTING 6-35. *Sample Thunks for* `ObQueryTypeInfo()` *and* `ObQueryTypeName()`

```
PERESOURCE WINAPI
__ObpRootDirectoryMutex (VOID)
    {
    return w2kXCopyP (NULL, "ObpRootDirectoryMutex");
    }

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

POBJECT_DIRECTORY WINAPI
__ObpRootDirectoryObject (VOID)
    {
    return w2kXCopyP (NULL, "ObpRootDirectoryObject");
    }

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

POBJECT_DIRECTORY WINAPI
__ObpTypeDirectoryObject (VOID)
    {
    return w2kXCopyP (NULL, "ObpTypeDirectoryObject");
    }
```

LISTING 6-36. *Sample Thunks for Some Internal Variables*

This should suffice for now. You may be a bit disappointed that I am not adding sample code here to demonstrate the usage of the `w2k_call.dll` API functions. Don't worry—you *will* get your sample code in the next chapter.