CHAPTER 2

The Windows 2000 Native API

This introductory chapter about the Windows 2000 Native API focuses on the relationships among the operating system modules that form the environment of this basic programming interface. Emphasis is on the central interrupt gate mechanism employed by Windows 2000 to route kernel service requests from user-mode to kernel-mode and back. Additionally, the Win32K interface and some of the major runtime libraries associated with the Native API will be presented, along with some of the most frequently used data types. The chapter closes with hints for those who want to write applications that interface to the Native API via the `ntdll.dll` library.

The architecture of Windows 2000 has been described in detail elsewhere. Many things written about Windows NT also apply to Windows 2000, so both editions of *Inside Windows NT* (Custer 1993, Solomon 1998) are good introductory books, as is the follow-up volume *Inside Windows 2000* (Solomon and Russinovich 2000).

THE NT*() AND ZW*() FUNCTION SETS

One of the most interesting facts about the architecture of Windows 2000 is that it can emulate various operating systems. Windows 2000 comes with three built-in subsystems for Win32, POSIX, and OS/2 applications. The Win32 subsystem is clearly the most popular, and therefore it is frequently regarded by application developers as *the* operating system itself. They cannot really be blamed for this misconception—this point of view is correct for legacy operating systems such as Windows 95 or 98, with which the Win32 interface implementation is actually a fundamental part of the system. However, Windows 2000 is designed quite differently. Although the Win32 subsystem contains a system module named `kernel32.dll`, this is actually not the real operating system kernel. Instead, it is just one of the basic components of the Win32 subsystem. In many programming books, software development for Windows

NT and 2000 is reduced to the task of interfacing to the Win32 Application Programming Interface (API), concealing the fact that the NT platform exposes yet another more basic interface called the *Native API*. Developers writing kernel-mode device or file system drivers are already familiar with the Native API, because kernel-mode modules are located on a low system level where the subsystems are invisible. However, you don't have to go down to the driver level to access this interface—even ordinary Win32 applications can call down to the Native API at any time. There's no technical restriction—it's just that Microsoft doesn't support this kind of application development. Thus, little information has been available on this topic, and neither the Windows Platform Software Development Kit (SDK) nor the Windows 2000 Device Drive Kit (DDK) make the Native API available to Win32 applications. So this work has been left to others, and this book is another piece of the puzzle.

LEVELS OF "UNDOCUMENTEDNESS"

Much of the material presented in this book refers to so-called undocumented information. In its global sense, this means that this information isn't published by Microsoft. However, there are several grades of "undocumentedness" because of the large amount of information that could possibly be published about a huge operating system such as Windows 2000. My personal category system looks as follows:

- *Officially documented:* The information is available in one of Microsoft's books, papers, or development kits. The most prominent information sources are the SDK, DDK, and the Microsoft Developer Network (MSDN) Library.
- *Semidocumented:* Although not officially documented, the information can be extracted from files officially distributed by Microsoft. For example, many Windows 2000 functions and structures aren't mentioned in the SDK or DDK documentation, but appear in some header files or sample programs. For Windows 2000, the most important sources of semidocumentation are the header files `ntddk.h` and `ntdef.h`, which are part of the DDK.
- *Undocumented, but not hidden:* The information in question is neither found in the official documentation nor included in any form in the developer products, but parts of it are available for debugging tools. All symbolic information contained in executable or symbol files belongs to this category. The best examples are the `!processfields` and `!threadfields` commands of the Kernel Debugger, which dump the names and offsets of the undocumented `EPROCESS` and `ETHREAD` structures (see Chapter 1).

- *Completely undocumented:* Some information bits are so well hidden by Microsoft, that they can be unveiled only by reverse engineering and inference. This class contains many implementation-specific details that nobody except the Windows 2000 developers should care about, but it also includes information that might be invaluable for system programmers, particularly developers of debugging software. Unveiling system internals such as this is extremely difficult, but also incredibly interesting, for someone who loves puzzles of a million pieces.

The Windows 2000 internals discussed in this book are equally distributed on levels two, three, and four of this category system, so there should be something for everyone.

THE SYSTEM SERVICE DISPATCHER

The relationship between the Win32 subsystem API and the Native API is best explained by showing the dependencies between the Win32 core modules and the Windows 2000 kernel. Figure 2-1 illustrates the module relationships, using boxes for modules and arrows for dependencies. If an arrow points from module A to module B, this means that A depends on B, that is, module A calls functions inside module B. Modules connected by double arrows are mutually dependent on each other. In Figure 2-1, the modules `user32.dll`, `advapi32.dll`, `gdi32.dll`, `rpcrt4.dll`, and `kernel32.dll` represent the basic Win32 API providers. Of course, there are other DLLs that contribute to this API, such as `version.dll`, `shell32.dll`, and `comctl32.dll`, but for clarity, I have omitted them. An interesting property illustrated in Figure 2-1 is that all Win32 API calls are ultimately routed through `ntdll.dll`, which forwards them to `ntoskrnl.exe`.

The `ntdll.dll` module is the operating system component that hosts the Native API. To be more exact, `ntdll.dll` is the user-mode front end of the Native API. The "real" interface is implemented in `ntoskrnl.exe`. The file name already suggests that this is the *NT Operating System Kernel.* In fact, kernel mode drivers call into this module most of the time if they require operating system services. The main role of `ntdll.dll` is to make a certain subset of kernel functions available to applications running in user mode, including the Win32 subsystem DLLs. In Figure 2-1, the arrow pointing from `ntdll.dll` to `ntoskrnl.exe` is labeled `INT 2Eh` to indicate that Windows 2000 uses an interrupt gate to switch the CPU's privilege level from user mode to kernel mode. Kernel-mode programmers view user-mode code as offensive, buggy, and dangerous. Therefore, this kind of code must be kept away from kernel functions. Switching the privilege level from user mode to kernel mode and back in the course of an API call is one way to handle this problem in a controlled manner. The calling application never really touches any kernel bytes—it can only look at them.

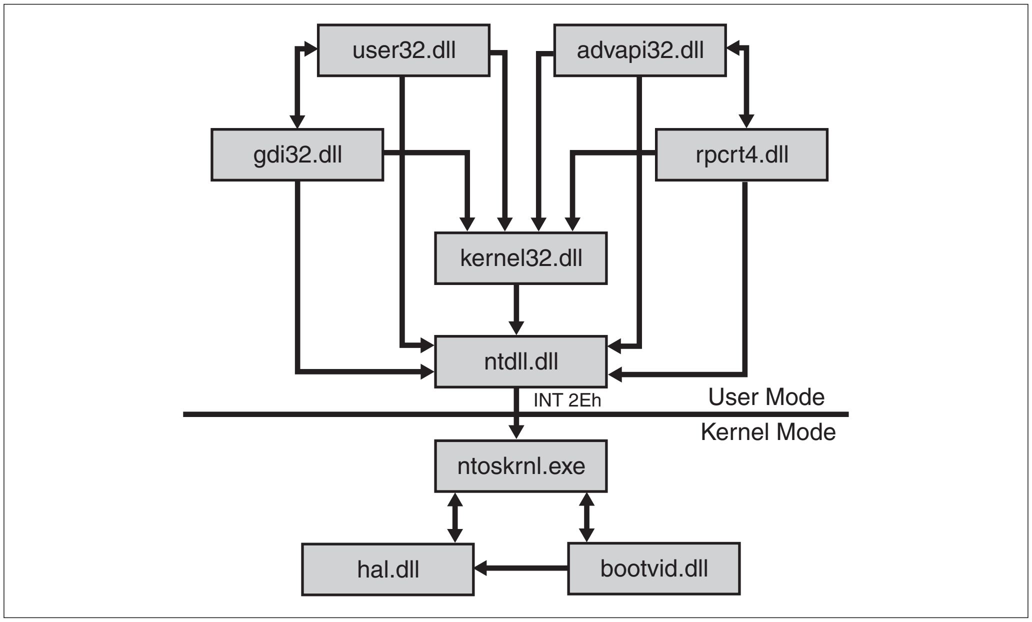

FIGURE 2-1. *System Module Dependencies*

For example, the Win32 API function `DeviceIoControl()` exported by `kernel32.dll` eventually calls the `ntdll.dll` export `NtDeviceIoControlFile()`. Disassembling this function reveals a surprisingly simple implementation, shown in Example 2-1. First, CPU register `EAX` is loaded with the magic number `0x38`, which is a dispatch ID. Next, register `EDX` is set up to point into the stack. The target address is the current value of the stack pointer `ESP` plus four, so `EDX` will point right behind the stack slot of the return address that has been saved on the stack immediately before entering `NtDeviceIoControlFile()`. Of course, this is the place where the arguments passed to the function are temporarily stored. The next instruction is a simple `INT 2Eh`, which branches to the interrupt handler stored in slot `0x2E` of the Interrupt Descriptor Table (IDT). Doesn't that look familiar? In fact, this code looks quite a bit like an old DOS `INT 21h` API call. However, the `INT 2Eh` interface of Windows 2000 is much more than a simple API call dispatcher—it serves as the main gate from user mode to kernel mode. Please note that this implementation of the mode switch is Intel i386 CPU specific. On the Alpha platform, different tricks are employed to achieve this transition.

```
NtDevi       ceIoControlFile:
        mov     eax, 38h
        lea     edx, [esp+4]
        int     2Eh
        ret     28h
```

EXAMPLE 2-1. *Implementation of* `ntdll.NtDeviceIoControlFile()`

The Windows 2000 Native API comprises 248 functions that are handled this way. That's 37 more than in Windows NT 4.0. You can easily recognize them by the function name prefix `Nt` in the `ntdll.dll` export list. There are 249 symbols of this kind exported by `ntdll.dll`. The reason for this mismatch is that one of the functions, `NtCurrentTeb()`, is a pure user-mode function and therefore isn't passed to the kernel. Table B-1 in Appendix B lists all available Native API functions, along with their `INT 2Eh` dispatch IDs, if any. The table also indicates which functions are exported by `ntoskrnl.exe`. Surprisingly, only a subset of the Native API can be called from kernel-mode modules. On the other hand, `ntoskrnl.exe` exports two `Nt*` symbols not provided by `ntdll.dll`, namely `NtBuildNumber` and `NtGlobalFlag`. Neither symbol refers to a function. Instead, they are pointers to `ntoskrnl.exe` variables that can be imported by a driver module using the C compiler's `extern` keyword. The Windows 2000 kernel exports many more variables in this manner, and the sample code following later will make use of some of them.

You may wonder why Table B-1 provides two columns for `ntdll.dll` and `ntoskrnl.exe`, respectively, labeled `ntdll.Nt*`, `ntdll.Zw*`, `ntoskrnl.Nt*`, and `ntoskrnl.Zw*`. The reason is that both modules export two sets of related Native API symbols. One of them comprises all names involving the `Nt` prefix, as listed in the leftmost column of Table B-1. The other set contains similar names, but with `Nt` replaced by `Zw`. Disassembly of `ntdll.dll` shows that each pair of symbols refers to exactly the same code. This may appear to be a waste of memory. However, if you disassemble `ntoskrnl.exe`, you will find that the `Nt*` symbols point to real code and the `Zw*` variants refer to `INT 2Eh` stubs such as the one shown in Example 2-1. This means that the `Zw*` function set is routed through the user-to-kernel-mode gate, and the `Nt*` symbols point directly to the code that is executed after the mode transition.

Two more things in Table B-1 should be noted. First, the function `NtCurrentTeb()` doesn't have a `Zw*` counterpart. This is not a big problem because the `Nt*` and `Zw*` functions exported by `ntdll.dll` are the same anyway. Second, `ntoskrnl.exe` doesn't consistently export `Nt/Zw` function pairs. Some of them come in either `Nt*` or `Zw*` versions only. I do not know the reason for this—I suppose that `ntoskrnl.exe` exports only the functions documented in the Windows 2000 DDK plus those required by other operating system modules. Note that the remaining Native API functions are nevertheless implemented inside `ntoskrnl.exe`. They don't feature a public entry point, but of course they may be reached from outside through the `INT 2Eh` gate.

The Service Descriptor Tables

The disassembled code in Example 2-1 has shown that `INT 2Eh` is invoked with two parameters passed in the CPU registers `EAX` and `EDX`. I have already mentioned that the magic number in `EAX` is a dispatch ID. Because all Native API calls except `NtCurrentTeb()` are squeezed through the same hole, the code handling the `INT 2Eh` must determine which call should be dispatched to which function. That's why the dispatch ID is provided. The interrupt handler inside `ntoskrnl.exe` uses the value in `EAX` as an index into a lookup table, where it finds the information required to route the call to its ultimate destination. This table is called a System Service Table (SST), and the corresponding C structure `SYSTEM_SERVICE_TABLE` is defined in Listing 2-1. This listing also comprises the definition of a structure named `SERVICE_DESCRIPTOR_TABLE`, which is a four-member array of SSTs, the first two of which serve special purposes.

Although both tables are fundamental data types, they are not documented in the Windows 2000 DDK, which leads to the following important statement: Many code snippets reprinted in this book contain undocumented data types and functions. Therefore, there's no guarantee that this information is authentic. This is true for all symbolic information, such as structure names, structure members, and arguments. When creating symbols, I attempt to use appropriate names, based on the naming scheme apparent through the small subset of known symbols (including those available from the symbol files). However, this heuristic approach is likely to fail on many occasions. Only the original source code contains the full information, but I don't have access to it. Actually, I don't *want* to see the source code, because this would require a Non-Disclosure Agreement (NDA) with Microsoft, and the ties of an NDA would make it quite difficult to write a book about undocumented information.

So let's return to the secrets of the Service Descriptor Table (SDT). Its definition in Listing 2-1 shows that the first pair of slots is reserved for `ntoskrnl.exe` and the kernel-mode part of the Win32 subsystem buried inside the `win32k.sys` module. The calls dispatched through the `win32k` SST originate from `gdi32.dll` and `user32.dll`. `ntoskrnl.exe` exports a pointer to its main SDT via the symbol `KeServiceDescriptorTable`. The kernel maintains an alternative SDT named `KeServiceDescriptorTableShadow`, but this one is not exported. It is very simple to access the main SDT from a kernel-mode module—you need only two C instructions, as shown in Listing 2-2. The first is a simple variable declaration preceded by the `extern` keyword, which tells the linker that this variable is not part of the module and the corresponding symbol cannot be resolved at link time. All references to this symbol are linked dynamically as soon as the module is loaded into the address space of a process. The second C instruction in Listing 2-2 is such a reference. Assigning `KeServiceDescriptorTable` to a variable of type `PSERVICE_DESCRIPTOR_TABLE` causes the creation of a dynamic link to `ntoskrnl.exe`, similar to an API call into a DLL module.

```
typedef NTSTATUS (NTAPI *NTPROC) ();
typedef NTPROC *PNTPROC;
#define NTPROC_ sizeof (NTPROC)

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

typedef struct _SYSTEM_SERVICE_TABLE
    {
    PNTPROC ServiceTable;           // array of entry points
    PDWORD  CounterTable;           // array of usage counters
    DWORD   ServiceLimit;           // number of table entries
    PBYTE   ArgumentTable;          // array of byte counts
    }
        SYSTEM_SERVICE_TABLE,
     * PSYSTEM_SERVICE_TABLE,
    **PPSYSTEM_SERVICE_TABLE;

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

typedef struct _SERVICE_DESCRIPTOR_TABLE
    {
    SYSTEM_SERVICE_TABLE ntoskrnl;  // ntoskrnl.exe (native api)
    SYSTEM_SERVICE_TABLE win32k;    // win32k.sys (gdi/user support)
    SYSTEM_SERVICE_TABLE Table3;    // not used
    SYSTEM_SERVICE_TABLE Table4;    // not used
    }
        SERVICE_DESCRIPTOR_TABLE,
     * PSERVICE_DESCRIPTOR_TABLE,
    **PPSERVICE_DESCRIPTOR_TABLE;
```

LISTING 2-1. *Structure of the Service Descriptor Table*

```
// Import SDT pointer
extern PSERVICE_DESCRIPTOR_TABLE KeServiceDescriptorTable;

// Create SDT reference
PSERVICE_DESCRIPTOR_TABLE psdt = KeServiceDescriptorTable;
```

LISTING 2-2. *Accessing the Service Descriptor Table*

The ServiceTable member of each SST contained in an SDT points to an array of function pointers of type NTPROC, which is a convenient placeholder for the Native API functions, similar to the PROC type used in Win32 programming. NTPROC is defined at the top of Listing 2-1. Native API functions typically return an NTSTATUS code and use the NTAPI calling convention, which is synonymous to __stdcall. The ServiceLimit member holds the number of entries found in the ServiceTable array. On Windows 2000, its default value is 248. The ArgumentTable is an array of BYTEs, each one corresponding to a ServiceTable slot and indicating the number of argument bytes (!) available on the caller's stack. This information, along with the pointer supplied in register EDX, is required by the kernel when it copies the arguments from the caller's stack to its own, as described below. The CounterTable member is not used in the free build of Windows 2000. In the debug build, this member points to an array of DWORDs that represent usage counters for each function. This information can be used for profiling purposes.

It is easy to display the contents of the SDT using the Windows 2000 Kernel Debugger. Please refer to Chapter 1 if you haven't yet set up this very useful application. In Example 2-2, I have first issued the command dd KeServiceDescriptorTable. The debugger resolves this public symbol to 0x8046AB80 and displays a hex dump of the next 32 DWORDs at this address. Only the first four rows are significant, corresponding to the four SDT members in Listing 2-1. For better readability, they are printed in boldface. If you take a closer look, you will see that the fifth row looks exactly like the first—could this be another SDT? This is a great occasion for a test of the Kernel Debugger's ln command (List Nearest Symbols). In Example 2-2. right after the hex dump of KeServiceDescriptorTable, I have entered the command ln 8046abc0. Obviously, the debugger knows the address 0x8046abc0 well and converts it to the symbol KeServiceDescriptorTableShadow, proving that this is indeed the second SDT maintained by the kernel. The obvious difference between the SDTs is that the latter contains entries for win32k.sys, whereas the former doesn't. In both tables, the members Table3 and Table4 are empty. ntoskrnl.exe provides a convenient API function named KeAddSystemServiceTable() to fill these slots.

```
kd> dd KeServiceDescriptorTable
dd KeServiceDescriptorTable
8046ab80  804704d8 00000000 000000f8 804708bc
8046ab90  00000000 00000000 00000000 00000000
8046aba0  00000000 00000000 00000000 00000000
8046abb0  00000000 00000000 00000000 00000000
8046abc0  804704d8 00000000 000000f8 804708bc
8046abd0  a01859f0 00000000 0000027f a0186670
```

```
8046abe0  00000000 00000000 00000000 00000000
8046abf0  00000000 00000000 00000000 00000000
kd> ln 8046abc0
ln 8046abc0
(8047b3a0)   ntoskrnl!KeServiceDescriptorTableShadow

kd> ln 804704d8
ln 804704d8
(8046cd00)   ntoskrnl!KiServiceTable

kd> ln 804708bc
ln 804708bc
(8046d0e4)   ntoskrnl!KiArgumentTable

kd> ln a01859f0
ln a01859f0
(a016d8c0)   win32k!W32pServiceTable

kd> ln a0186670
ln a0186670
(a016e544)   win32k!W32pArgumentTable

kd> dd KiServiceTable
dd KiServiceTable
804704d8  804ab3bf 804ae86b 804bdef3 8050b034
804704e8  804c11f4 80459214 8050c2ff 8050c33f
804704f8  804b581c 80508874 8049860a 804fc7e2
80470508  804955f7 8049c8a6 80448472 804a8d50
80470518  804b6bfb 804f0cef 804fcb95 8040189a
80470528  804d06cb 80418f66 804f69d4 8049e0cc
80470538  8044c422 80496f58 804ab849 804aa9da
80470548  80465250 804f4bd5 8049bc80 804ca7a5

kd> db KiArgumentTable
db KiArgumentTable
804708bc  18 20 2c 2c 40 2c 40 44-0c 18 18 08 04 04 0c 10  . ,,@,@D........
804708cc  18 08 08 0c 08 08 04 04-04 0c 04 20 08 0c 14 0c  ........... ....
804708dc  2c 10 0c 1c 20 10 38 10-14 20 24 1c 14 10 20 10  ,... .8.. $... .
804708ec  34 14 08 04 04 04 0c 08-28 04 1c 18 18 18 08 18  4.......(.......
804708fc  0c 08 0c 04 10 00 0c 10-28 08 08 10 00 1c 04 08  ........(.......
8047090c  0c 04 10 00 08 04 08 0c-28 10 04 0c 0c 28 24 28  ........(....($(
8047091c  30 0c 0c 0c 18 0c 0c 0c-0c 30 10 0c 0c 0c 0c 10  0........0......
8047092c  10 0c 0c 14 0c 14 18 14-08 14 08 08 04 2c 1c 24  .............,.$

kd> ln 8044c422
ln 8044c422
(80449c90)   ntoskrnl!NtClose
```

EXAMPLE 2-2. *Examination of the Service Descriptor Tables*

Note that I have truncated the output lines of the `ln` command, to demonstrate only the essential information.

At address `0x8046AB88` of the `KeServiceDescriptorTable` hex dump, where the `ServiceLimit` member should be located, the value `0xF8`—248 in decimal notation—shows up, as expected. The values of `ServiceTable` and `ArgumentTable` are pointers to the addresses `0x804704d8` and `0x804708bc`, respectively. This is another case for the `ln` command, revealing the names `KiServiceTable` and `KiArgumentTable`, respectively. None of these symbols is exported by `ntoskrnl.exe`, but the debugger recognizes them by looking into the Windows 2000 symbol files. The `ln` command can also be applied to the pointers in the `win32k` SST. For the `ServiceTable` and `ArgumentTable` members, the debugger reports `W32pServiceTable` and `W32pArgumentTable`, respectively. Both symbols are taken from the symbol file of `win32k.sys`. If the debugger refuses to resolve these addresses, issue the `.reload` command to force a reload of all available symbol files and try again.

The remaining parts of Example 2-2 are hex dumps of the first 128 bytes of `KiServiceTable` and `KiArgumentTable`. If the things I said about the Native API so far are correct, then the `NtClose()` function should be addressed by slot 24 of `KiServiceTable`, located at address `0x80470538`. The value found there is `0x8044c422`, marked boldface in the results of the `dd KiServiceTable` command. Applying the `ln` command to this address yields `NtClose()`. As a final test, let's examine slot 24 of `KiArgumentTable` at address `0x804708d4`. In the Windows 2000 DDK, `ZwClose()` is documented as receiving a single argument of type `HANDLE`, so the number of argument bytes on the caller's stack should amount to four. It doesn't come as a big surprise that this is exactly the value found in the argument table, marked boldface in the results of the `db KiArgumentTable` command.

THE INT 2Eh SYSTEM SERVICE HANDLER

The interrupt handler lurking at the kernel-mode side of the `INT 2Eh` gate is labeled `KiSystemService()`. Again, this is an internal symbol not exported by `ntoskrnl.exe`, but contained in the Windows 2000 symbol files. Therefore, the Kernel Debugger can resolve it without problem. Essentially, `KiSystemService()` performs the following steps:

1. Retrieve the SDT pointer from the current thread's control block.
2. Determine which one of the four SSTs in the SDT should be used. This is done by testing bits 12 and 13 of the dispatch ID in register `EAX` and selecting the corresponding SDT member. IDs in the range `0x0000-0x0FFF` are mapped to the `ntoskrnl` table; the range `0x1000-0x1FFF` is assigned to

the win32k table. The remaining ranges 0x2000-0x2FFF and 0x3000-0x3FFF are reserved for the additional SDT members Table3 and Table4. If an ID exceeds 0x3FFF, the unwanted bits are masked off before dispatching.

3. Check bits 0 to 11 of the dispatch ID in register EAX against the ServiceLimit member of the selected SST. If the ID is out of range, an error code of STATUS_INVALID_SYSTEM_SERVICE is returned. In an unused SST, this member is zero, yielding an error code for all possible dispatch IDs.
4. Check the argument stack pointer in register EDX against the value of MmUserProbeAddress. This is a public variable exported from ntoskrnl.exe and usually evaluates to 0x7FFF0000. If the argument pointer is not below this address, STATUS_ACCESS_VIOLATION is returned.
5. Look up the number of argument stack bytes in the ArgumentTable referenced by the SST, and copy all function arguments from the caller's stack to the current kernel-mode stack.
6. Look up the service function pointer in the ServiceTable referenced by the SST, and call this function.
7. Transfer control to the internal function KiServiceExit() after returning from the service call.

It is interesting to see that the INT 2Eh handler doesn't use the global SDT addressed by KeServiceDescriptorTable, but uses a thread-specific pointer instead. Obviously, threads can have different SDTs associated to them. On thread initialization, KeInitializeThread() writes the KeServiceDescriptorTable pointer to the thread control block. However, this default setting may be changed later to a different value, such as KeServiceDescriptorTableShadow, for example.

THE WIN32 KERNEL-MODE INTERFACE

The discussion of the SDT in the previous section has shown that a second main kernel-mode interface exists along with the Native API. This interface connects the Graphics Device Interface (GDI) and the Window Manager (USER) of the Win32 subsystem to a kernel-mode component called Win32K, introduced with Windows NT 4.0, and residing in the file win32k.sys. This component has been added to overcome an inherent performance limit of the Win32 display engine, caused by the original Windows NT subsystem design. On Windows NT 3.x, the client-server model imposed on the Win32 subsystem and the kernel involved frequent switches from user-mode to kernel-mode and back. By moving considerable parts of the display engine to the kernel-mode module win32k.sys, much of this overhead could be eliminated.

WIN32K DISPATCH IDS

Now that `win32k.sys` has entered the scene, it's time for an update of Figure 2-1. Figure 2-2 is based on the original drawing, but with a `win32k.sys` box added to the left of `ntoskrnl.exe`. I have also added arrows pointing from `gdi32.dll` and `user32.dll` to `win32k.sys`. Of course, this is not 100 percent correct, because the `INT 2Eh` calls inside these modules are actually directed to `ntoskrnl.exe`, which owns the interrupt handler. However, the calls are ultimately handled by `win32k.sys`, and this is what the arrows should indicate.

As pointed out earlier, the Win32K interface is also based on the `INT 2Eh` dispatcher, much like the Native API. The only difference is that Win32K uses a different range of dispatch IDs. Although all Native API calls involve dispatch IDs that range from `0x0000` to `0x0FFF`, Win32K dispatch IDs are numbers between `0x1000` and `0x1FFF`. As Figure 2-2 demonstrates, the primary Win32K clients are `gdi32.dll` and `user32.dll`. Therefore, it should be possible to find out the symbolic names associated to the Win32K dispatch IDs by disassembling these modules. As it turns out, only a small subset of `INT 2Eh` calls has public names in their export sections, so it is again time for a Kernel Debugger session. In Example 2-3, I have issued the command `dd W32pServiceTable`. To be sure that the `win32k.sys` symbols are available, it is preceded by a `.reload` command.

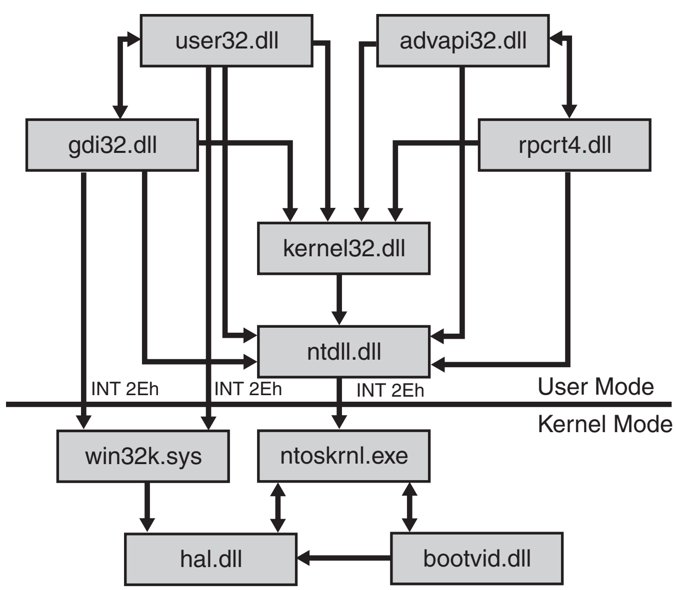

FIGURE 2-2. *System Module Dependencies, including* `win32k.sys`

```
kd> .reload
.reload
Loading Kernel Symbols...
Unable to read image header for fdc.sys at f0798000 - status c0000001
Unable to read image header for ATMFD.DLL at beaaf000 - status c0000001
Loading User Symbols...
Unable to read selector for PCR for Processor 0
PPEB is NULL (Addr= 0000018c)

kd> dd W32pServiceTable
dd W32pServiceTable
a01859f0  a01077f0 a011f59e a000788a a01141e1
a0185a00  a0121264 a0107e05 a01084df a010520b
a0185a10  a0120a6f a008c9eb a00befa2 a007cb5c
a0185a20  a0085c9b a001e4e7 a0120fd1 a0122d19
a0185a30  a0085d0c a0122e73 a0027671 a006d1f0
a0185a40  a0043fe0 a009baeb a007eb9b a009eb05
a0185a50  a0043392 a007c14f a01229cc a0027470
a0185a60  a001ad09 a00af751 a004e9f5 a004ef53

kd> ln a01077f0
ln a01077f0
(a00b316e)   win32k!NtGdiAbortDoc   |  (a00ba173)      win32k!IsRectEmpty
```

EXAMPLE 2-3. *Examination of the Win32K System Services*

In the last three lines of Example 2-3, I have applied the `ln` command to the first entry in the `W32pServiceTable` hex dump. So the Win32K function with dispatch ID zero is obviously called `NtGdiAbortDoc()`. You can repeat this procedure for all 639 dispatch IDs, but it is better to automate the symbol lookup. I have done this for you, and the results are collected in Appendix B, Table B-2. The symbol mapping from `gdi32.dll` and `user32.dll` to `win32k.sys` is simple: A GDI symbol is converted to a Win32K symbol by adding the prefix `NtGdi`, and a USER symbol is converted by adding `NtUser`. However, there are some minor exceptions. For example, if a GDI symbol starts out with `Gdi`, the prefix is reduced to `Nt`, probably to avoid the character sequence `NtGdiGdi`. In some other instances, the character case is different (e.g., `EnableEUDC()` and `NtGdiEnableEudc()`), or a trailing W marking a Unicode function is missing (e.g., `CopyAcceleratorTableW()` and `NtUserCopyAcceleratorTable()`).

Documenting the complete Win32K API in detail would be a tremendous effort. The function set is almost three times larger than the Native API. Maybe someday someone will pick up the pieces and write a great reference handbook, like Gary Nebbett did for the Native API (Nebbett 2000). For the scope of this book, the above information should suffice, however.

THE WINDOWS 2000 RUNTIME LIBRARY

The `Nt*()` and `Zw*()` functions making up the Native API are an essential, but nevertheless minor, part of the code found inside `ntdll.dll`. This DLL exports no fewer than 1179 symbols. 249/248 of them belong to the `Nt*()/Zw*()` sets, so there are still 682 functions left that are not routed through the `INT 2Eh` gate. Obviously, this large group of functions doesn't rely on the Windows 2000 kernel. So what purpose do they serve?

THE C RUNTIME LIBRARY

If you study the symbols in the export section of `ntdll.dll`, you will find many lowercase function names that look quite familiar to a C programmer. These well-known names, such as `memcpy()`, `sprintf()`, and `qsort()`, are members of the C Runtime Library incorporated into `ntdll.dll`. The same is true for `ntoskrnl.exe`, which features a similar set of C Runtime functions, although these sets are not identical. Table B-3 in Appendix B lists the union of both sets and points out which ones are available from which module.

You can link to these functions by simply adding the file `ntdll.lib` from the Windows 2000 DDK to the list of import libraries that should be scanned by the linker during symbol resolution. If you prefer using dialogs, you can choose the **Settings...** entry from the **Project** menu of Visual C/C++, click the **Link** tab, select the category **General,** and append `ntdll.dll` to the **Object/library modules** list. Alternatively, you can add the line `#pragma comment (linker, "/defaultlib:ntdll.lib")` somewhere to your source code. This has the same effect, but has the advantage that other developers can rebuild your project with default Visual C/C++ settings.

Disassembling the code of some of the C Runtime functions available from both `ntdll.dll` and `ntoskrnl.exe` shows that `ntdll.dll` does not rely on `ntoskrnl.exe` here, like it did with respect to the Native API functions. Instead, both modules implement the functions separately. The same applies to all other functions presented in this section. Note that some of these functions in Table B-3 aren't intended for import by name. For example, if you are using the shift operators `>>` and `<<` on 64-bit `LARGE_INTEGER` numbers in a kernel-mode driver, the compiler and linker will automatically import the `_allshr()` and `_allshl()` functions from `ntoskrnl.exe`, respectively.

THE EXTENDED RUNTIME LIBRARY

Along with the standard C Runtime, Windows 2000 provides an extended set of runtime functions. Again, both `ntdll.dll` and `ntoskrnl.exe` implement them separately, and, again, the implemented sets overlap, but don't match exactly. The functions belonging to this group share the common name prefix `Rtl` (for Runtime Library). Table B-4 in Appendix B lists them all, using the same layout as Table B-3. The Windows 2000 Runtime Library contains helper functions for common tasks that go beyond the capabilities of C Runtime. For example, some of them handle security issues, others manipulate Windows 2000–specific data structures, and still others support memory management. It is hard to understand why Microsoft documents just 115 out of these 406 extremely useful functions in the Windows 2000 DDK.

THE FLOATING-POINT EMULATOR

I'll conclude this gallery of API functions with another function set provided by `ntdll.dll`, just to show how many interesting functions are buried inside this goldmine. Table 2-1 lists a set of names that should look somewhat familiar to assembly language programmers. Take one of the names starting with `__e` and strip this prefix—you get an assembly language mnemonic of the floating-point unit (FPU) built into the i386-compatible CPUs. In fact, `ntdll.dll` contains a full-fledged floating-point emulator, represented by the functions in Table 2-1. This proves again that this DLL is an immense repository of code and almost invites a system spelunker to disassembly.

TABLE 2-1. *The Floating Point Emulator Interface of* ntdll.dll

FUNCTION NAMES			
__eCommonExceptions	__eFIST32	__eFLD64	__eFSTP32
__eEmulatorInit	__eFISTP16	__eFLD80	__eFSTP64
__eF2XM1	__eFISTP32	__eFLDCW	__eFSTP80
__eFABS	__eFISTP64	__eFLDENV	__eFSTSW
__eFADD32	__eFISUB16	__eFLDL2E	__eFSUB32
__eFADD64	__eFISUB32	__eFLDLN2	__eFSUB64
__eFADDPreg	__eFISUBR16	__eFLDPI	__eFSUBPreg
__eFADDreg	__eFISUBR32	__eFLDZ	__eFSUBR32
__eFADDtop	__eFLD1	__eFMUL32	__eFSUBR64
__eFCHS	__eFIDIVR16	__eFMUL64	__eFSUBreg
__eFCOM	__eFIDIVR32	__eFMULPreg	__eFSUBRPreg
__eFCOM32	__eFILD16	__eFMULreg	__eFSUBRreg
__eFCOM64	__eFILD32	__eFMULtop	__eFSUBRtop
__eFCOMP	__eFILD64	__eFPATAN	__eFSUBtop
__eFCOMP32	__eFIMUL16	__eFPREM	__eFTST

(continued)

TABLE 2-1. *(continued)*

FUNCTION NAMES			
__eFCOMP64	__eFIMUL32	__eFPREM1	__eFUCOM
__eFCOMPP	__eFINCSTP	__eFPTAN	__eFUCOMP
__eFCOS	__eFINIT	__eFRNDINT	__eFUCOMPP
__eFDECSTP	__eFIST16	__eFRSTOR	__eFXAM
__eFIDIVR16	__eFIST32	__eFSAVE	__eFXCH
__eFIDIVR32	__eFISTP16	__eFSCALE	__eFXTRACT
__eFILD16	__eFISTP32	__eFSIN	__eFYL2X
__eFILD32	__eFISTP64	__eFSQRT	__eFYL2XP1
__eFILD64	__eFISUB16	__eFST	__eGetStatusWord
__eFIMUL16	__eFISUB32	__eFST32	NPXEMULATORTABLE
__eFIMUL32	__eFISUBR16	__eFST64	RestoreEm87Context
__eFINCSTP	__eFISUBR32	__eFSTCW	SaveEm87Context
__eFINIT	__eFLD1	__eFSTENV	
__eFIST16	__eFLD32	__eFSTP	

For more information about the floating-point instruction set, please consult the original documentation of the Intel CPUs 80386 and up. For example, the Pentium manuals can be downloaded in PDF format from Intel's Web site at http://developer.intel.com/design/pentium/manuals/. The manual explaining the machine code instruction set is called *Intel Architecture Software Developer's Manual. Volume 2: Instruction Set Reference* (Intel 1999b). Another great reference book with detailed FPU information is Robert L. Hummel's aged but still applicable i486 handbook (Hummel 1992).

OTHER API FUNCTION CATEGORIES

Along with the functions listed explicitly in Appendix B and Table 2-1, `ntdll.dll` and `ntoskrnl.exe` export numerous other functions specific to various components of the kernel. Rather than add more lengthy tables to this book, I'm including a short one that lists the available function name prefixes with their associated categories (Table 2-2). The `ntdll.dll` and `ntoskrnl.exe` columns contain the entry N/A (not applicable) for modules that do not export functions of this category.

TABLE 2-2. *Function Prefix to Function Category Mapping*

PREFIX	ntdll.dll	ntoskrnl.exe	CATEGORY
__e		N/A	Floating-point emulator
Cc	N/A		Cache manager
Csr		N/A	Client-server runtime library
Dbg			Debugging support
Ex	N/A		Executive support
FsRtl	N/A		File system runtime library
Hal	N/A		Hardware Abstraction Layer (HAL) dispatcher
Inbv	N/A		System initialization/VGA boot driver (bootvid.dll)
Init	N/A		System initialization
Interlocked	N/A		Thread-safe variable manipulation
Io	N/A		I/O manager
Kd	N/A		Kernel Debugger support
Ke	N/A		Kernel routines
Ki			Kernel interrupt handling
Ldr			Image loader
Lpc	N/A		Local Procedure Call (LPC) facility
Lsa	N/A		Local Security Authority (LSA)
Mm	N/A		Memory manager
Nls			National Language Support (NLS)
Nt			NT Native API
Ob	N/A		Object manager
Pfx			Prefix handling
Po	N/A		Power manager
Ps	N/A		Process support
READ_REGISTER_	N/A		Read from register address
Rtl			Windows 2000 runtime library
Se	N/A		Security handling
WRITE_REGISTER_	N/A		Write to register address
Zw			Alternative Native API
<other>			Helper functions and C runtime library

NA, Not applicable.

Many kernel functions use a uniform naming scheme of type `Prefix OperationObject()`. For example, the function `NtQueryInformationFile()` belongs to the Native API because of its `Nt` prefix, and obviously it executes a `QueryInformation operation on a File object`. Not all functions obey this rule, but many do, so it is usually easy to guess what a function does by simply parsing its name.

FREQUENTLY USED DATA TYPES

When writing software that interacts with the Windows 2000 kernel—whether in user-mode via `ntdll.dll` or in kernel-mode via `ntoskrnl.exe`—you will have to deal with a couple of basic data types that are rarely seen in the Win32 world. Many of them appear repeatedly in this book. The following section outlines the most frequently used types.

INTEGRAL TYPES

Traditionally, integral data types come in several different variations. Neither the Win32 Platform SDK header files nor the SDK documentation commit themselves to a special nomenclature—they mix fundamental C/C++ types with several derived types. Table 2-3 lists the commonly used integral types, showing their equivalence relationships. In the "MASM" column, the assembly language type names expected by the Microsoft Macro Assembler (MASM) are shown. The Win32 Platform SDK defines `BYTE, WORD,` and `DWORD` as aliases for the corresponding fundamental C/C++ data types. The columns "Alias #1" and "Alias #2" contain other frequently used aliases. For example, `WCHAR` represents the basic Unicode character type. The last column, "Signed," lists the usual aliases of the corresponding signed data types. It is important to keep in mind that ANSI characters of type `CHAR` are signed quantities, whereas the Unicode `WCHAR` is unsigned. This inconsistency can lead to unexpected side effects when the compiler converts these types to other integral values in arithmetic or logical expressions.

The MASM `TBYTE` type (read "10-byte") in the last row of Table 2-3 is an 80-bit floating-point number used in high-precision floating-point unit (FPU) operations. Microsoft Visual C/C++ doesn't offer an appropriate fundamental data type to Win32 programs—the 80-bit *long double* type featured by Microsoft's 16-bit compilers is now treated like a *double,* that is, i.e. a signed 64-bit number with an 11-bit exponent and a 52-bit mantissa, according to the IEEE `real*8` specification. Please note that the MASM `TBYTE` type has nothing to do with the Win32 `TBYTE` (read "text byte"), which is a convenient macro that can define a `CHAR` or `WCHAR` type, depending on the absence or presence of a `#define UNICODE` line in the source code.

TABLE 2-3. *Equivalent Integral Data Types*

BITS	MASM	FUNDAMENTAL	ALIAS #1	ALIAS #2	SIGNED
8	BYTE	unsigned char	UCHAR		CHAR
16	WORD	unsigned short	USHORT	WCHAR	SHORT
32	DWORD	unsigned long	ULONG		LONG
32	DWORD	unsigned int	UINT		INT
64	QWORD	unsigned __int64	ULONGLONG	DWORDLONG	LONGLONG
80	TBYTE	N/A			

The Windows 2000 Device Driver Kit (DDK) is more consistent in its use of aliases. You will usually come across the type names in the "Alias #1" and "Signed" columns throughout the header files and documentation. As a long-term assembly language programmer, I've grown accustomed to using the MASM types. Therefore, you will frequently find the names listed in the "MASM" column in the header files on the companion CD of this book.

Because 64-bit integer handling is somewhat awkward in a 32-bit programming environment, Windows 2000 usually does not employ the fundamental `__int64` type and its derivatives. Instead, the DDK header file `ntdef.h` defines a neat union/structure combination that allows different interpretations of a 64-bit quantity as either a pair of 32-bit chunks or a 64-bit monolith. Listing 2-3 shows the definition of the `LARGE_INTEGER` and `ULARGE_INTEGER` types, representing signed and unsigned integers, respectively. The sign is controlled by using `LONGLONG/ULONGLONG` for the 64-bit QuadPart member or `LONG/ULONG` for the 32-bit HighPart member.

STRINGS

In Win32 programming, the basic types `PSTR` and `PWSTR` are commonly used for ANSI and Unicode strings. `PSTR` is defined as `CHAR*`, and `PWSTR` is a `WCHAR*` (see Table 2-3). Depending on the absence or presence of the `#define UNICODE` directive in the source code, the additional `PTSTR` pseudo-type evaluates to `PSTR` or `PWSTR`, respectively, allowing maintenance of ANSI and Unicode versions of an application with a single set of source files. Basically, these strings are simply pointers to zero-terminated `CHAR` or `WCHAR` arrays. If you are working with the Windows 2000 kernel, you have to deal with quite different string representations. The most common type is the UNICODE_STRING, which is a three-part structure defined in Listing 2-4.

```
typedef union _LARGE_INTEGER
    {
    struct
        {
        ULONG LowPart;
        LONG HighPart;
        };
    LONGLONG QuadPart;
    }
    LARGE_INTEGER, *PULARGE_INTEGER;

typedef union _ULARGE_INTEGER
    {
    struct
        {
        ULONG LowPart;
        ULONG HighPart;
        };
    ULONGLONG QuadPart;
    }
   U LARGE_INTEGER, *PULARGE_INTEGER;
```

LISTING 2-3. LARGE_INTEGER *and* ULARGE_INTEGER

```
typedef struct _UNICODE_STRING
    {
    USHORT Length;
    USHORT MaximumLength;
    PWSTR  Buffer;
    }
    UNICODE_STRING, *PUNICODE_STRING;

typedef struct _STRING
    {
    USHORT Length;
    USHORT MaximumLength;
    PCHAR  Buffer;
    }
    STRING, *PSTRING;

typedef STRING ANSI_STRING, *PANSI_STRING;
typedef STRING OEM_STRING,  *POEM_STRING;
```

LISTING 2-4. *Structured String Types*

The Length member specifies the current length of the string in bytes—not characters! The MaximumLength member indicates the size of the memory block addressed by the Buffer member where the string data resides, again in bytes, not characters. Because Unicode characters are 16 bits wide, the Length is always twice the number of string characters. Usually, the string pointed to by the Buffer member is zero-terminated. However, some kernel-mode modules might rely entirely on the Length value and won't take care of adding the terminating zero character, so be careful in case of doubt.

The ANSI version of the Windows 2000 string structure is simply called STRING, as shown in Listing 2-4. For convenience, ntdef.h also defines the ANSI_STRING and OEM_STRING aliases to distinguish 8-byte strings containing characters of different code pages (default ANSI code page: 1252; default OEM code page: 437). However, the predominant string type of the Windows 2000 kernel is the UNICODE_STRING. You will come across 8-bit strings only occasionally.

In Figure 2-3, I have drawn two typical UNICODE_STRING examples. The sample on the left-hand side consists of two independent memory blocks: a UNICODE_STRING structure and an array of 16-bit PWCHAR Unicode characters. This is probably the most common string type found inside the Windows 2000 data areas. On the right-hand side, I have added a frequently occurring special case, in which both the UNICODE_STRING and the PWCHAR are part of the same memory block. Several kernel functions, including some inside the Native API, return structured system information in contiguous memory blocks. If the data includes strings, they are often stored as embedded UNICODE_STRINGs, as shown in the right half of Figure 2-3. For example, the NtQuerySystemInformation() function used in the sample code of Chapter 1 makes heavy use of this special string representation.

These string structures don't need to be manipulated manually. ntdll.dll and ntoskrnl.exe export a rich set of runtime API functions such as RtlCreatUnicodeString(), RtlInitUnicodeString(), RtlCopyUnicodeString(), and the like. Usually, an equivalent function is available for the STRING and ANSI_STRING types as well. Many of these functions are officially documented in the DDK, but some are not. However, it is usually easy to guess what the undocumented string functions do and what arguments they take. The main advantage of UNICODE_STRING and its siblings is the implicit specification of the size of the buffer containing the string. If you are passing a UNICODE_STRING to a function that converts its value in place, possibly increasing its length, this function simply has to examine the MaximumLength member to find out whether enough space is left for the result.

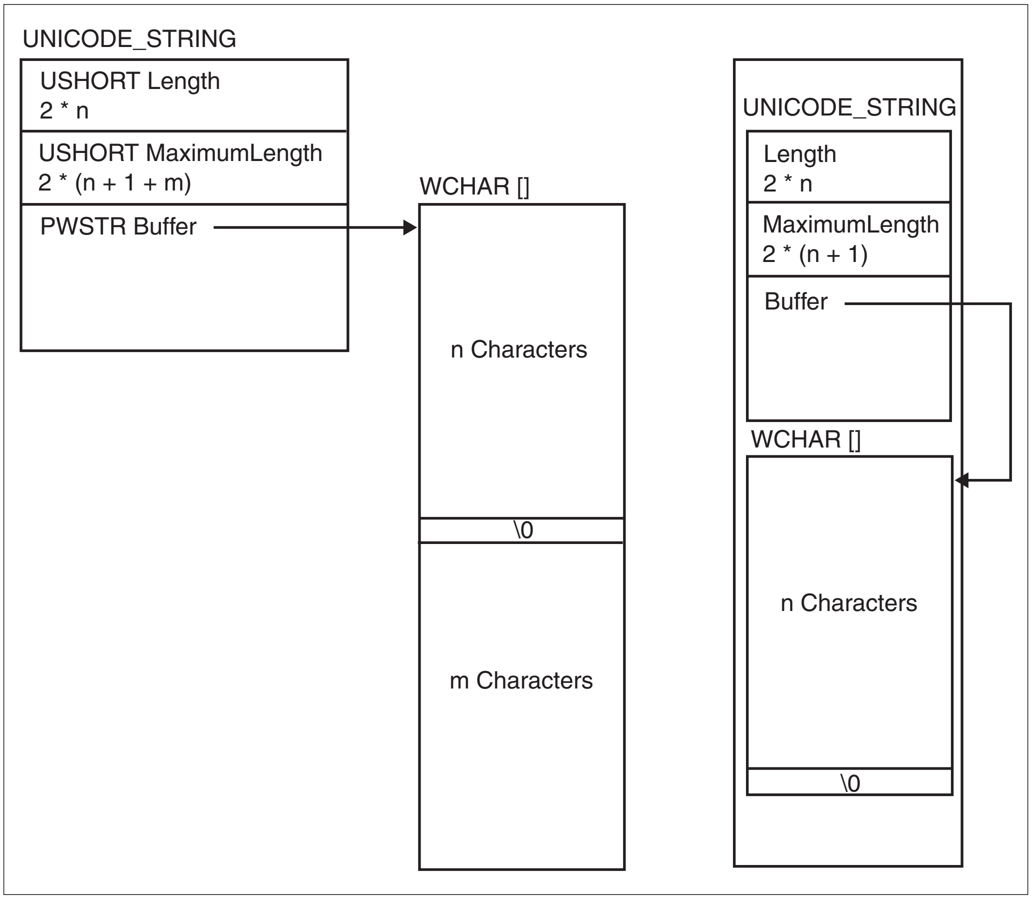

FIGURE 2-3. *Examples of* UNICODE_STRINGs

STRUCTURES

Several kernel API functions that work with objects expect them to be specified by an appropriately filled OBJECT_ATTRIBUTES structure, outlined in Listing 2-5. For example, the NtOpenFile() function doesn't have a PWSTR or PUNICODE_STRING argument for the path of the file to be opened. Instead, the ObjectName member of an OBJECT_ATTRIBUTES structure indicates the path. Usually, the setup of this structure is trivial. Along with the ObjectName, the Length and Attributes members are required. The Length must be set to sizeof (OBJECT_ATTRIBUTES), and the Attributes are a combination of OBJ_* values from ntdef.h, for example, OBJ_CASE_INSENSITIVE if the object name should be matched without regard to character case. Of course, the ObjectName is a UNICODE_STRING pointer, not a plain PWSTR. The remaining members can be set to NULL as long as they aren't needed.

Whereas the OBJECT_ATTRIBUTES structure specifies details about the input data of an API function, the IO_STATUS_BLOCK structure in Listing 2-6 provides information about the outcome of the requested operation. This structure is quite simple—the Status member contains an NTSTATUS code, which can assume the value STATUS_SUCCESS or any of the error codes defined in the DDK header file ntstatus.h. The Information member provides additional request-specific data in case of success. For example, if the function has returned a data block, this member is typically set to the size of this block.

Another ubiquitous Windows 2000 data type is the LIST_ENTRY structure, shown in Listing 2-7. The kernel uses this simple structure to arrange objects in doubly linked lists. It is quite common that one object is part of several lists, resulting in multiple LIST_ENTRY structures used in the object's definition. The Flink member is the forward link, pointing to the next item, and the Blink member is the backward link, addressing the previous one. The links always point to another LIST_ENTRY, not to the owner object itself. Usually, the linked lists are circular, that is, the last Flink points to the first LIST_ENTRY in the chain, and the first Blink points to the end of the list. This makes it easy to traverse a linked list in both directions from either end or even from a list item somewhere in the middle. If a program walks down a list of objects, it has to save the address of the starting point to find out when it is time to stop. If a list contains just a single entry, its LIST_ENTRY must reference itself—that is, both the Flink and Blink members point to their own LIST_ENTRY.

```
typedef struct _OBJECT_ATTRIBUTES
    {
    ULONG           Length;
    HANDLE          RootDirectory;
    PUNICODE_STRING ObjectName;
    ULONG           Attributes;
    PVOID           SecurityDescriptor;
    PVOID           SecurityQualityOfService;
    }
    OBJECT_ATTRIBUTES, *POBJECT_ATTRIBUTES;
```

LISTING 2-5. *The* OBJECT_ATTRIBUTES *structure*

```
typedef struct _IO_STATUS_BLOCK
    {
    NTSTATUS Status;
    ULONG    Information;
    }
    IO_STATUS_BLOCK, *PIO_STATUS_BLOCK;
```

LISTING 2-6. *The* IO_STATUS_BLOCK *structure*

```
typedef struct _LIST_ENTRY
 {
 struct _LIST_ENTRY *Flink;
 struct _LIST_ENTRY *Blink;
 }
 LIST_ENTRY, *PLIST_ENTRY;
```

LISTING 2-7. *The* `LIST_ENTRY` *Structure*

Figure 2-4 illustrates the relationships between the members of object lists. Objects A1, A2, and A3 are part of a three-item list. Note how `A3.Flink` points back to A1 and `A1.Blink` points to A3. Object B1 on the right-hand side is the only member of an orphaned list. Hence, its `Flink` and `Blink` members point to the same address inside Object B1. Typical examples of doubly linked lists are process and thread lists. The internal variable `PsActiveProcessHead` is a `LIST_ENTRY` structure inside the `.data` section of `ntoskrnl.exe` that addresses the first (and—by virtue of its `Blink` pointer—also the last) member of the system's process list. You can walk down this list in a Kernel Debugger console window by first issuing the command `dd PsActiveProcessHead,` and then using copy and paste to set up subsequent `dd` commands for the `Flink` or `Blink` values. Of course, this is an annoying way of exploring Windows 2000 processes, but it might help gaining insight into the basic system architecture. The Windows 2000 Native API features much more convenient ways of enumerating processes, such as `NTQuerySystemInformation()` function.

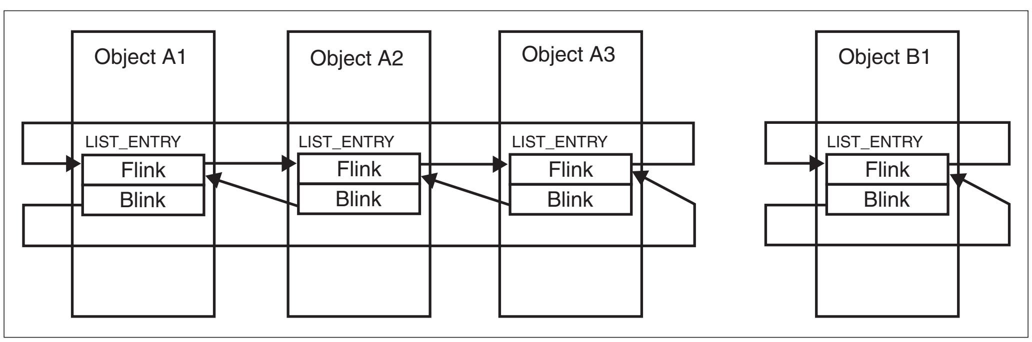

FIGURE 2-4. *Examples of Doubly Linked Lists*

API functions operating on processes and threads, such as `NtOpenProcess()` and `NtOpenThread()`, use the `CLIENT_ID` structure shown in Listing 2-8 to jointly specify process and thread IDs. Although defined as `HANDLE` types, the `UniqueProcess` and `UniqueThread` members aren't handles in the strict sense. Instead, they are integral process and thread IDs, as returned by the standard Win32 API functions `GetCurrentProcessId()` and `GetCurrentThreadId()`, which have `DWORD` return values.

The `CLIENT_ID` structure is also used by the Windows 2000 Executive to globally identify a thread in the system. For example, if you are issuing the Kernel Debugger's `!thread` command to display the parameters of the current thread, it will list its `CLIENT_ID` in the first output line as "Cid ppp.ttt," where "ppp" is the value of the `UniqueProcess` member, and "ttt" is the `UniqueThread` ID.

INTERFACING TO THE NATIVE API

For kernel-mode drivers, interfacing to the Native API is normal, just as calling Win32 API functions is in a user-mode application. The header and library files provided by the Windows 2000 DDK contain everything needed to call into the Native API exposed by `ntoskrnl.exe`. On the other hand, the Win32 Platform SDK contains almost no support for applications that want to use Native API functions exported by `ntdll.dll`. I say "almost" because one important item is actually included: It is the import library `ntdll.lib`, supplied in the `\Program Files\Microsoft Platform SDK\Lib` directory. Without the library, it would be difficult to call functions exported by `ntdll.dll`.

ADDING THE NTDLL.DLL IMPORT LIBRARY TO A PROJECT

Before you can successfully compile and link user-mode code that uses `ntdll.dll` API functions, you must consider the following four important points:

1. The Platform SDK header files don't contain prototypes for these functions.
2. Several basic data structures used by these functions are missing from the SDK files.

```
typedef struct _CLIENT_ID
    {
    HANDLE UniqueProcess;
    HANDLE UniqueThread;
    }
    CLIENT_ID, *PCLIENT_ID;
```

LISTING 2-8. *The* `CLIENT_ID` *Structure*

3. The SDK and DDK header files are incompatible—you cannot add `#include <ntddk.h>` to your Win32 C source files.
4. `ntdll.lib` is not included in the default list of import libraries offered by Visual C/C++.

The last problem is easily solved. Just edit the project settings of your application, or add the line `#pragma comment (linker, "/defaultlib:ntdll.lib")` to your source code, as explained in the section The Windows 200 Runtime Library earlier in this chapter. This linker pragma adds `ntdll.lib` to the `/defaultlib` settings of the linker command at compile time. The problem with the missing definitions is much more difficult. Because it is not possible to merge the SDK and DDK header files in programs written in plain C, the least expensive solution is to write a custom header file that contains just as many definitions as needed to call the required `ntdll.dll` API functions. Fortunately, you don't have to start from scratch. The `w2k_def.h` file in the `\src\common\include` directory of the sample CD contains much of the basic information you may need. This header file will play an important role in Chapters 6 and 7. Because it is designed to be compatible to both user-mode and kernel-mode projects, you must insert the line `#define _USER_MODE_` somewhere before the `#include <w2k_def.h>` line in user-mode code to enable the definitions that are present in the DDK but missing from the SDK.

Considerable information about Native API programming has already been published elsewhere. Three good sources of detailed information on this topic are listed below in chronological order of publication:

- Mark Russinovich has published an article titled "Inside the Native API" on the *sysinternals.com* Web site, available for download at http://www.sysinternals.com/ntdll.htm (Russinovich 1998).
- The November 1999 issue of *Dr. Dobb's Journal* `(DDJ)` contains my article "Inside Windows NT System Data," which details, among other things, how to interface to `ntdll.dll` and provides lots of sample code that facilitates this task (Schreiber 1999). The sample code can be downloaded from the *DDJ* Web site at http://www.ddj.com/ftp/1999/1999_11/ntinfo.zip. Please note that this article targets Windows NT 4.0 only.
- Gary Nebbett's recently published Native API bible, *Windows NT/2000 Native API Reference* (Nebbett 2000), doesn't contain much sample code, but it does feature complete coverage of all Native API functions available in Windows NT 4.0 and Windows 2000, including the data structures and other definitions they require. It is an ideal complement to the above articles.

The `w2k_call.dll` sample library, introduced in Chapter 6, demonstrates the typical usage of `w2k_def.h`. Chapter 6 also discusses an alternative method to call into the Windows 2000 kernel from user-mode that isn't restricted to the Native API function set. Actually, this trick is not restricted to `ntoskrnl.exe`—it is applicable to *any* module loaded into kernel memory that either exports API functions or comes with matching `.dbg` or `.pdb` symbol files. As you see, there is plenty of interesting material waiting for you in the remaining chapters of this book. But, before we get there, we'll discuss some fundamental concepts and techniques.