Chapter 3

# Win32 Implementations: A Comparative Look

IN THIS CHAPTER

- Comparing and contrasting implementation of the Win32 API in Windows NT and Windows 95
- Aspects of implementation in both Windows NT and Windows 95

EACH OPERATING SYSTEM provides sets of services - referred to as an *application programming interface (API)* - to developers in some form or another. The developers write software applications using this API. For example, DOS provides this interface in the form of the famous INT 21h interface. Microsoft's newer 32-bit operating systems, such as Windows 95 and Windows NT, provide the interface in the form of the Win32 API.

Presently, there are four Win32 API implementations available from Microsoft:

- Windows 95/98
- Windows NT
- Win32s
- Windows CE

Of these, Win32s is very limited due to bugs and the restrictions of the underlying operating system. Presently, Win32 API implementations on Windows 95/98 and Windows NT are very popular among developers. Windows CE is meant for palmtop computers. The Win32 API was first implemented on the Windows NT operating system. Later, the same API was made available in Windows 95. Ideally, an application written using the standard Win32 API should work on any operating system that supports the Win32 API implementation. (However, this is not necessarily true due to the differences between the implementations.) The Win32 API should hide all the details of the underlying implementations and provide a consistent view to the outside world.

In this chapter, we focus on the differences between the implementations of the Win32 API under Windows NT and Windows 95. As developers, you should be aware of these differences while you develop applications that can run on both of these operating systems.

# Win32 API Implementation on Windows 95

The Win32 API is provided in the form of the famous trio of the KERNEL32, USER32, and GDI32 dynamic link libraries (DLLs). However, in most cases, these DLLs are just wrappers that use generic thunking to call the 16-bit functions.


Generic *thunking* is a way of calling 16-bit functions from a 32-bit application. (More on thunking later in this chapter.)

The major design goal for Windows 95 was backward compatibility. Hence, instead of porting all the 16-bit functions to 32-bit, Microsoft decided to reuse the existing 16-bit code (from the Windows 3.*x* operating system) by wrapping it in 32-bit code. This 32-bit code would in turn call the 16-bit functions. This was a good approach because the tried—and—true 16-bit code was already running of machines all over the world. In this Win32 API implementation, most of the functions from KERNEL32 thunk down to KRNL386, USER32 thunks down to USER.EXE, and GDI32 thunks down to GDI.EXE.

# Win32 API Implementation on Windows NT

On Windows NT also, the Win32 API is provided in the form of the famous trio of the KERNEL32, USER32, and GDI32 DLLs. However, this implementation is done completely from scratch without using any existing 16-bit code, so it is purely a 32-bit implementation of Win32 API. Even 16-bit applications end up calling this 32-bit API. Windows NT's 16-bit subsystem uses universal thunking to achieve this.


*Universal thunking* is a way of calling 32-bit functions from 16-bit applications. (More on thunking later in this chapter.)

KRNL386.EXE, USER.EXE, and GDI.EXE, which are used to support 16-bit applications, thunk up to KERNEL32, USER32, and GDI32 through the WOW (Windows on Windows) layer. Most of the functions provided by KERNEL32.DLL call one or more native system services to do the actual work. The native system services are available through a DLL called NTDLL.DLL.


All these system services are discussed in Chapter 6.

As far as USER32 and GDI32 are concerned, the implementation differs in NT versions 3.51 and later versions. Under Windows NT 3.51, a separate subsystem process implements the USER32 and GDI32 calls. The DLLs USER32 and GDI32 contain stubs, which pass the function parameters to the Win32 subsystem (CSRSS.EXE) and get the results back. The communication between the client application and the Win32 subsystem is achieved by using the local procedure call facility provided by the NT executive.


Chapter 8 covers the details of the local procedure call (LPC) mechanism.

Under Windows NT 4.0 and Windows 2000, the USER32 GDI32 calls the system services provided by a kernel-mode device driver called WIN32K.SYS. USER32 and GDI32 contain stubs that call these system services using the 2Eh interrupt. Hence, most of the functionality of the Win32 Subsystem process (CSRSS.EXE) is taken over by the kernel-mode driver (WIN32K.SYS). The CSRSS process still exists in NT 4.0 and Windows 2000 — however, its role is limited to mainly supporting Console I/O.

It is interesting to note that the Win32 API completely hides NTDLL.DLL from the developer. Actually, most of the functions provided by the Win32 API ultimately call one or more system services. This system service layer is very powerful and many times contains functions that do not have equivalent Win32 API functions. Most of the Windows NT Resource Kit utilities link to this DLL implicitly.

# Win32 Implementation Differences

Now we will consider a few aspects of the Win32 API implementation on Windows NT and Windows 95 that might affect the way developers program using this so-called standard Win32 API.

## Address Space

Both Windows 95 and Windows NT deal with flat, 32-bit linear addresses that give 4GB of virtual address space. Of this, the upper 2GB (hereafter referred to as the *shared address space)* is reserved for operating system use, and the lower 2GB (hereafter referred to as the *private address space)* is used by the running process. The private address space of each process is different for each process. Although the virtual addresses in the private address space of all processes is the same, they may point to a different physical page. The addresses in the shared address space of all the processes point to the same physical page.

Under Windows 95/98, the operating system DLLs, such as KERNEL32, USER32, and GDI32, reside in the shared address space, whereas in Windows NT these DLLs are loaded in the process's private address space. Hence, under Windows 95/98, it is possible for one application to interfere with the working of another application. For example, one application can accidentally overwrite memory areas occupied by these DLLs and affect the working of all the other processes.


Although the shared address space is protected at the page table level, a kernel-mode component (for example, a VXD) is able to write at any location in 4GB address space.

In addition, under Windows 95/98, it is possible to load a dynamic link library in the shared address space. These DLLs will have the same problem described previously if the DLL is used by multiple applications in the system.

Windows NT loads all the system DLLs, such as KERNEL32, USER32, and GDI32, in the private address space. As a result, it is never possible for one application to

interfere with the other applications in the system without intending to do so. If one application accidentally overwrites these DLLs, it will affect only that application. Other applications will continue to run without any problems.

Memory-mapped files are loaded in the shared address space under Windows 95/98, whereas they are loaded in the private address space in Windows NT. In Windows 95/98, it is possible for one application to create and map a memory-mapped file, pass its address to another application, and have the other application use this address to share memory. This is not possible under Windows NT. You have to explicitly create and map a named memory-mapped file in one application and open and map the memory-mapped file in another application in order to share it.

The address space differences have strong impacts on global API hooking. The topic of global API hooking has been covered many times in different articles and books. There is still no common API hooking solution for both Windows NT and Windows 95/98. The basic problem with global API hooking is that under Windows 95/98, it is possible to load a DLL in shared memory. Also, all the system DLLs reside in shared memory. Hooking an API call amounts to patching the few instructions at the start of function and routing them to a function in a shared DLL using a simple JMP instruction. This does not work under Windows NT because if you patch the bytes at the start of the function, they will be patched only in your address space as the function resides in the private address space.

To do any kind of global API hooking under Windows NT, you have to make sure that the hooking is performed in each of the running processes. For this, you need to play with the address space of other processes. In addition, the same hooking also needs to be done in newly started processes. Windows NT provides a way to automatically load a particular DLL in each process through the AppInit_DLL registry key.

## Process Startup

There are several differences in the way the process is started under Windows 95/98 and Windows NT. Although the same CreateProcess API call is used in Windows 95/98 and Windows NT, the implementation is quite different. In this chapter, we are looking only at an example of a CreateProcess API call. Ideally, both of the CreateProcess implementations should give the same view to the outside world. When somebody says that a particular API call is *standard,* this means that given a specific set of parameters to a function, the function should behave exactly the same on all the implementations of this API call. In addition, the function should return the same error codes based on the type of error.

Consider a simple problem such as detecting the successful start of an application. If you try to spawn a program that has some startup problem (for example, implicitly linked DLLs are missing), it should return an appropriate error code. The Windows 95/98 implementation returns an appropriate error code such as STATUS_DLL_NOT_FOUND, whereas Windows NT does not return any error. Windows

NT's implementation will return an error only if the file spawned is not present at the expected location. This happens mainly because of the way the CreateProcess call is implemented under Windows NT and Windows 95/98. When you spawn a process in Windows 95/98, the complete loading and startup of the process is performed as part of the CreateProcess call itself. That is, when the CreateProcess call returns, the spawned process is already running.

It is interesting to see Windows NT's implementation of the CreateProcess call. Windows NT's CreateProcess calls the native system service (NtCreateProcess) to create a process object. As part of this call, NTDLL.DLL is mapped in the process's address space. Then, the CreateProcess API calls the native system service to create the primary thread in the process (NtCreateThread). The implicitly linked DLL loading does not happen as part of the CreateProcess API call. Instead, the primary thread of the process starts at a function in NTDLL.DLL. This function in turn loads the implicitly loaded DLLs. As a result, there is no way for the caller to know whether the process has started properly or not. Of course, for GUI applications, you can use WaitForInputIdle to synchronize with the startup of a process. However, for non-GUI applications, there is no standard way to achieve this.

## Toolhelp Functions

Win32 implementation on Windows 95/98 provides some functions that enable you to enumerate the processes running in the system, module list, and so on. These functions are provided by KERNEL32.DLL. The functions are CreateToolHelp32Snapshot, Process32First, Process32Next, and others. These functions are not implemented under Windows NT's implementation of KERNEL32. The programs that use these functions implicitly will not start at all under Windows NT. The Windows NT 4.0 SDK comes with a new DLL called PSAPI.DLL, which provides the equivalent functionality. The header file for this PSAPI.H is also included with the Windows NT 4.0 SDK. Windows 2000 has this toolhelp functionality built into KERNEL32.DLL.


A function is *implicitly linked* if the program calls the function directly by name and includes the appropriate .LIB file in the project. That is, it does not use GetProcAddress to get the address of the function.

## Multitasking

Both Windows 95 and Windows NT use time slice-based preemptive multitasking. However, because the Windows 95 implementation of the WIN32 API depends largely on 16-bit code, it has a few inherent drawbacks. The major one is the Win16Mutex. Because the existing 16-bit code is not well suited for multitasking, the easiest choice

for Microsoft was to ensure that the 16-bit code is not entered from multiple tasks. To achieve this, Microsoft came up with the Winl6Mutex solution.

Before entering the 16-bit code, the operating system acquires the Winl6Mutex, and it leaves the Winl6Mutex while returning from 16-bit code. The Winl6Mutex is always acquired when a 16-bit application is running, which results in reduced multitasking. Windows NT does not have this problem because the entire code is 32-bit and is well suited for time slice-based preemptive multitasking. Also, the 16-bit code thunks up to 32-bit code in the case of Windows NT.

# Thunking

Thunking enables 16-bit applications to run in a 32-bit environment and vice versa. It is a way of calling a function written in one bitness from the code running at a different bitness. Bitness is a property of the processor, and you can program the processor to adjust the bitness. Bitness decides the way instructions are decoded by the processor. There are two different types of thunking available:

+ Universal thunking
+ Generic thunking

Universal thunking enables you to call a 32-bit function from 16-bit code, whereas generic thunking enables you to call a 16-bit function from 32-bit code. Windows 95/98 supports both generic and universal thunking, but Windows NT supports only universal thunking. As you saw earlier in this chapter, generic thunking is used extensively in WIN32 API implementation of Windows 95/98. For example, a 32-bit USER32.DLL calls functions from a 16-bit USER.EXE, and a 32-bit GDI32.DLL calls functions from a 16-bit GDI.EXE. Various issues are involved in thunking, such as converting 16:16 far pointers in 16-bit code to flat 32-bit address and manipulating a stack for making a proper call from code running at one bitness to code running at a different bitness. Microsoft provides tools such as thunk compilers to automate most of these tasks.

Many vendors who write code for Windows 95/98 use generic thunking to avoid a major redesign of their applications. For example, say a particular vendor has a product for Windows 3.1 and would like to port it to Windows 95. Instead of rewriting the code for Windows 95, an easier solution is to use the majority of the existing 16-bit code and use generic thunking as a way of calling this code from 32-bit applications. However, these applications need to be rewritten for Windows NT as Windows NT does not support generic thunking.

# Device Drivers

Device drivers are trusted components of the operating system that have full access to the entire hardware. There are no restrictions on what device drivers can do. Each operating system provides some way of adding new device drivers to the sys-

tern. The device drivers need to be written according to the semantics imposed by the operating system. The device drivers are called *virtual device drivers (VXD)* in Windows 95/98, and they are called as *kernel-mode device drivers* in Windows NT. Windows 95 uses LE file format for virtual device drivers, whereas Windows NT uses the PE format. As a result, the applications that use VXDs cannot be run on Windows NT. They need to be ported to a Windows NT (kernel-mode) device driver.


Chapter 2 explains how to write device drivers.

Microsoft has come up with a Common Driver Model in Windows 98 and Windows 2000. At this point, however, you need to port all the applications that use VXDs to Windows NT by writing an equivalent kernel-mode driver.

## Security

The major WIN32 API implementation difference between Windows 95/98 and Windows NT is security. Windows 95/98's implementation does not have any support for security. In all the Win32 API functions that have SECURITY ATTRIBUTES as one of the parameters, Windows 95/98's implementation just ignores these parameters. This has some impact on the way a developer programs. Registry APIs such as RegSaveKey and RegRestoreKey work fine under Windows 95/98. However, under Windows NT, you need to do a few things before you can use these functions. In Windows NT, there is a concept of privileges. There are different kinds of privileges, such as Shutdown, Backup, and Restore. Before using a function such as RegSaveKey, you need to acquire the Backup privilege. To use RegRestoreKey, you need to acquire the Restore privilege, and to use the InitiateSystemShutdown function, you need to acquire the Shutdown privilege.

Under Windows 95/98, anybody can install a VXD. To install a kernel-mode device driver under Windows NT, you need administrator privilege for security purposes. As mentioned previously, device drivers are trusted components of the operating system and have access to the entire hardware. By requiring privileges to install a device driver, Windows NT restricts the possibility that a guest account holder will install a device driver, which could potentially bring the whole system down to its knees.

## Newly Added **API** Calls

With each version of Windows NT, new APIs are being added to the WIN32 API set. Most of these APIs do not have an equivalent API under Windows 95/98. Also, there are a few APIs, such as CreateRemoteThread, that do not have the real imple-

mentation under Windows 95/98. Under Windows 95/98, this function returns ERROR_CALL_NOT_IMPLEMENTED. As a result, there will always be a few API calls that are not available on Windows 95/98 or are not implemented on Windows 95/98. At this point, one can only hope that Microsoft will implement the API in Windows 95/98 when they add a new API to Windows NT unless the API is architecture dependent.

# Summary

This chapter covered the WIN32 API implementation on Windows 95/98 and Windows NT. We discussed the differences between these two implementations with respect to address space, process startup, toolhelp functions, multitasking, thunking, device drivers, security, and newly added API calls.