CHAPTER

3

The Win32 API

IN THIS CHAPTER

This chapter gives you an introduction to the Win32 API and the Win32 system in general. The chapter discusses the capabilities of the Win32 system and also points out some key differences from the 16-bit implementation of various features. The intent of this chapter is not to document the Win32 system in depth but rather to give you a basic idea of how Win32 operates. By having a basic understanding of the Win32 operation, you'll be able use advanced features provided by the Win32 system whenever the need arises.

Objects—Then and Now

The term *objects* is used for a number of reasons. When we speak of the Win32 architecture, we're not speaking of objects as they exist in object-oriented programming or the Component Object Model (COM). Objects have a totally different meaning in this context, and to make things more confusing, *object* means something different in 16-bit Windows than it does in Win32. We want to make sure you understand what objects are in Win32.

Basically two types of objects are in the Win32 environment: kernel objects and GDI/User objects.

Kernel Objects

Kernel objects are native to the Win32 system and include events, file mappings, files, mailslots, mutexes, pipes, processes, semaphores, and threads. The Win32 API includes various functions specific to each kernel object. Before discussing kernel objects in general, we want to discuss processes that are essential to understanding how objects are managed in the Win32 environment.

Processes and Threads

A *process* can be thought of as a running application or an application instance. Therefore, several processes can be active at once in the Win32 environment. Each process gets its own 4GB address space for its code and data. Within this 4GB address space, any memory allocations, threads, file mappings, and so on exist. Additionally, any dynamic link libraries (DLLs) loaded by a process are loaded into the address space of the process. We'll say more about the memory management of the Win32 system later in this chapter, in the section "Win32 Memory Management."

Processes are inert. In other words, they execute nothing. Instead, each process gets a *primary thread* that executes code within the context of the process that owns this thread. A process may contain several threads; however, it has only one main or primary thread.

NOTE

A *thread* is an operating system object that represents a path of code execution within a particular process. Every Win32 application has at least one thread—often called the *primary thread* or *default thread*—but applications are free to create other threads to perform other tasks. Threads are covered in greater depth in Chapter 11, "Writing Multithreaded Applications."

When a process is created, the system creates the main thread for it. This thread may then create additional threads, if necessary. The Win32 system allocates CPU time called *time slices* to the threads of the process.

Table 3.1 shows some common process functions of the Win32 API.

TABLE 3.1 Process Functions

Function	*Purpose*
`CreateProcess()`	Creates a new process and its primary thread. This function replaces the `WinExec()` function used in Windows 3.11.
`ExitProcess()`	Exits the current process, terminating the process and all threads related to that process.
`GetCurrentProcess()`	Returns a pseudohandle of the current process. A *pseudohandle* is a special handle that can be interpreted as the current process handle. A real handle can be obtained by using the `DuplicateHandle()` function.
`DuplicateHandle()`	Duplicates the handle of a kernel object.
`GetCurrentProcessID()`	Retrieves the current process ID, which uniquely identifies the process throughout the system until the process has terminated.
`GetExitCodeProcess()`	Retrieves the exit status of a specified process.
`GetPriorityClass()`	Retrieves the priority class for a specified process. This value and the values of each thread priority in the process determine the base priority level for each thread.
`GetStartupInfo()`	Retrieves the contents of the `TStartupInfo` structure initialized when the process was created.
`OpenProcess()`	Returns a handle of an existing process as specified by a process ID.
`SetPriorityClass()`	Sets a process's priority class.
`TerminateProcess()`	Terminates a process and kills all threads associated with that process.
`WaitForInputIdle()`	Waits until the process is waiting for input from the user.

Some Win32 API functions require an application's instance handle, whereas others require a module handle. In 16-bit Windows, there was a distinction between these two values. This is not true under Win32. Every process gets its own instance handle. Your Delphi 5 applications can refer to this instance handle by accessing the global variable, `HInstance`. Because `HInstance` and the application's module handle are the same, you can pass `HInstance` to Win32 API functions calling for a module handle, such as the `GetModuleFileName()` function, which returns the filename of a specified module. See the following Caution for when `HInstance` does not refer to the module handle of the current application.

CAUTION

`HInstance` will not be the module handle of the application for code that's compiled into packages. Use `MainInstance` to refer always to the host application module and `HInstance` to refer to the module in which your code resides.

Another difference between Win32 and 16-bit Windows has to do with the `HPrevInst` global variable. In 16-bit Windows, this variable held the handle of a previously run instance of the same application. You could use the value to prevent multiple instances of your application from running. This no longer works in Win32. Each process runs within its own 4GB address space and can't see any other processes. Therefore, `HPrevInst` is always assigned the value `0`. You must use other techniques to prevent multiple instances of your application from running, as shown in Chapter 13, "Hard-core Techniques."

Types of Kernel Objects

There are several kinds of kernel objects. When a kernel object is created, it exists in the address space of the process, and that process gets a handle to that object. This handle can't be passed to other processes or reused by the next process to access the same kernel object. However, a second process can obtain its own handle to a kernel object that already exists by using the appropriate Win32 API function. For example, the `CreateMutex()` Win32 API function creates a named or unnamed mutex object and returns its handle. The `OpenMutex()` Win32 API returns the handle to an existing named mutex object. `OpenMutex()` passes the name of the mutex whose handle is being requested.

NOTE

Named kernel objects are optionally assigned a null-terminated string name when created with their respective `Create`*`XXXX`*`()` functions. This name is registered in the Win32 system. Other processes can access the same kernel object by opening it, using the Open*XXXX*() function, and passing the specified object name. A demonstration of this technique is used in Chapter 13, "Hard-core Techniques," where we show you how to prevent multiple instances of an application from running.

If you want to share a mutex across processes, you can have the first process create the mutex by using the `CreateMutex()` function. This process must pass a name that will be associated with this new mutex. Other processes must use the `OpenMutex()` function, to which they pass the same name of the mutex used by the first process. `OpenMutex()` will return a handle to the mutex object of the given name. Various security constraints may be imposed on other processes accessing existing kernel objects. Such security constraints are specified when the mutex is initially created with `CreateMutex()`. Look to the online help for these constraints as they apply to each kernel object.

Because multiple processes can access kernel objects, kernel objects are maintained by a usage count. As a second application accesses the object, the usage count is incremented. When it's done with the object, the application should call the `CloseHandle()` function, which decrements the object's usage count.

GDI and User Objects

Objects in 16-bit Windows referred to entities that could be referenced by a handle. This didn't include kernel objects because they didn't exist under 16-bit Windows.

In 16-bit Windows, there are two types of objects: those stored in the GDI and User local heaps, and those allocated from the global heap. Examples of GDI objects are brushes, pens, fonts, palettes, bitmaps, and regions. Examples of User objects are windows, window classes, atoms, and menus.

A direct relationship exists between an object and its handle. An object's handle is a selector that, when converted into a pointer, points to a data structure describing an object. This structure exists in either the GDI or User default data segment, depending on the type of object to which the handle refers. Additionally, a handle for an object referring to the global heap is a selector to the global memory segment; therefore, when converted to a pointer, it points to that memory block.

A result of this particular design is that objects in 16-bit Windows are sharable. The globally accessible Local Descriptor Table (LDT) stores the handles to these objects. The GDI and User default data segments are also globally accessible to all applications and DLLs under 16-bit Windows. Therefore, any application or DLL can get to an object used by another application. Do note that objects such as the LDT are only sharable in Windows 3.1 (16-bit Windows). Many applications use this arrangement for different purposes. One example is to enable applications to share memory.

Win32 deals with GDI and User objects a bit differently, and the same techniques you used in 16-bit Windows might not be applicable to the Win32 environment.

To begin with, Win32 introduces kernel objects, which we've already discussed. Also, the implementation of GDI and User objects is different under Win32 than under 16-bit Windows.

Under Win32, GDI objects are not shared like their 16-bit counterparts. GDI objects are stored in the address space of the process rather than in a globally accessible memory block (each process gets its own 4GB address space). Additionally, each process gets its own handle table, which stores handles to GDI objects within the process. This is an important point to remember, because you don't want to be passing GDI object handles to other processes.

Earlier, we mentioned that LDTs are accessible from other applications. In Win32, each process address space is defined by its own LDT. Therefore, Win32 uses LDTs as they were intended: as process-local tables.

CAUTION

Although it's possible that a process could call `SelectObject()` on a handle from another process and successfully use that handle, this would be entirely coincidental. GDI objects have different meanings in different processes, so you don't want to practice this method.

The managing of GDI handles happens in the Win32 GDI subsystem, which includes the validation of GDI objects and the recycling of handles.

User objects work similarly to GDI objects and are managed by the Win32 User subsystem. However, any handle tables are also maintained by User—not in the address space of the process, as with the GDI handle tables. Therefore, objects such as windows, window classes, atoms, and so on are sharable across processes.

Multitasking and Multithreading

Multitasking is a term used to describe an operating system's capability of running multiple applications concurrently. The system does this by issuing time slices to each application. In this sense, multitasking is not true multitasking but rather *task switching*. In other words, the operating system isn't really running multiple applications at the same time. Instead, it's running one application for a certain amount of time and then switching to another application and running it for a certain amount of time. It does this for each application. To the user it appears as though all applications are running simultaneously because the time slices are small.

This concept of multitasking isn't really a new feature of Windows and has existed in previous versions. The key difference between the Win32 implementation of multitasking and that of earlier versions of Windows is that Win32 uses *preemptive multitasking*, whereas earlier versions use *nonpreemptive multitasking* (which means that the Windows system doesn't schedule time to applications based on the system timer). Applications have to tell Windows that they're finished processing code before Windows can grant time to other applications. This is a problem because a single application can tie up the system with a lengthy process. Therefore, unless the programmers of the application make sure that the application gives up time to other applications, problems can arise for the user.

Under Win32, the system grants CPU time to the threads for each process. The Win32 system manages the time allotted to each thread based on thread priorities. This concept is discussed in greater depth in Chapter 11, "Writing Multithreaded Applications."

NOTE

The Windows NT/2000 implementation of Win32 offers the capacity to perform true multitasking on machines with multiple processors. Under these conditions, each application can be granted time on its own processor. Actually, each individual thread can be given CPU time on any available CPU in a multiprocessor machine.

Multithreading is the capability of an application to multitask within itself. This means that your application can perform different types of processing simultaneously. A process can have several threads, and each thread contains its own distinct code to execute. Threads may have dependencies on one another and therefore must be synchronized. For example, it wouldn't be a good idea to assume that a particular thread will finish processing its code when its result will be used by another thread. Thread-synchronization techniques are used to coordinate multiple-thread execution. Threads are discussed in greater depth in Chapter 11, "Writing Multithreaded Applications."

Win32 Memory Management

The Win32 environment introduces you to the 32-bit flat memory model. Finally, Pascal programmers can declare that big array without running into a compile error:

```
BigArray = array[1..100000] of integer;
```

The following sections discuss the Win32 memory model and how the Win32 system lets you manipulate memory.

Just What Is the Flat Memory Model?

The 16-bit world uses a segmented memory model. Under that model, addresses are represented with a *segment*:*offset* pair. The *segment* refers to a base address, and the *offset* represents a number of bytes from that base. The problem with this scheme is that it's confusing to the average programmer, especially when dealing with large memory requirements. It's also limiting—data structures larger than 64KB are extremely painful to manage and are therefore avoided.

Under the flat-memory model, these limitations are gone. Each process has its own 4GB address space to use for allocating large data structures. Additionally, an address actually represents a unique memory location.

How Does the Win32 System Manage Memory?

It's not likely that your computer has 4GB installed. How does the Win32 system make more memory available to your processes than the amount of physical memory installed on the computer? Addresses that are 32 bit don't actually represent a memory location in physical memory. Instead, Win32 uses *virtual addresses*.

By using virtual memory, each process can get its own 4GB virtual address space. The upper 2MB area of this address space belongs to Windows, and the bottom 2MB is where your applications reside and where you can allocate memory. One advantage to this scheme is that the thread for one process can't access the memory in another process. The address `$54545454` in one process points to a completely different location than the same address in another process.

It's important to note that a process doesn't actually have 4GB of memory but rather has the capability to access a range of addresses up to 4GB. The amount of memory available to a process really depends on how much physical RAM is installed on the machine and how much space is available on disk for a paging file. The physical RAM and the paging file are used by the system to break the memory available to a process into pages. The size of a page depends on the type of system on which Win32 is installed. These page sizes are 4KB for Intel platforms and 8KB for Alpha platforms. The defunct PowerPC and MIPS platforms used 4KB pages as well. The system then moves pages from the paging file to physical memory and back as needed. The system maintains a page map to translate the virtual addresses of a process to a physical address. We won't get into the hairy details of how all this happens; we just want to familiarize you with the general scheme of things at this point.

A developer can manipulate memory in the Win32 environment in essentially three ways: using virtual memory, file-mapping objects, and heaps.

Virtual Memory

Win32 provides you with a set of low-level functions that enable you to manipulate the virtual memory of a process. This memory exists in one of the following three states:

- *Free*. Memory that's available to be reserved and/or committed.
- *Reserved*. Memory within an address range that's reserved for future use. Memory within this address is protected from other allocation requests. However, this memory cannot be accessed by the process because no *physical* memory is associated with it until it's committed. The `VirtualAlloc()` function is used to reserve memory.
- *Committed*. Memory that has been allocated and associated with physical memory. Committed memory can be accessed by the process. The `VirtualAlloc()` function is used to commit virtual memory.

As stated earlier, Win32 provides various `Virtual`*`XXXX`*`()` functions for manipulating virtual memory, as shown in Table 3.2. These functions are also documented in detail in the online help.

TABLE 3.2 Virtual Memory Functions

Function	*Purpose*
`VirtualAlloc()`	Reserves and/or commits pages in a process's virtual address space.
`VirtualFree()`	Releases and/or decommits pages in a process's virtual address space.
`VirtualLock()`	Locks a region of a process's virtual address to prevent it from being swapped to a page file. This prevents page faults with subsequent accesses to that region.
`VirtualUnLock()`	Unlocks a specified region of memory in a process's address space so that it can be swapped to a page file if necessary.
`VirtualQuery()`	Returns information about a range of pages in the calling process's virtual address space.
`VirtualQueryEx()`	Returns the same information as `VirtualQuery()` except that it allows you to specify the process.
`VirtualProtect()`	Changes access protection for a region of committed pages in the calling process's virtual address space.
`VirtualProtectEx()`	Same as `VirtualProtect()` except that it makes changes to a specified process.

NOTE

The *`xxx`*`Ex()` routines listed in this table can only be used by a process that has debugging privileges on the other process. It's complicated and rare for anything but a debugger to use these routines.

Memory-Mapped Files

Memory-mapped files (file-mapping objects) allow you to access disk files in the same way you would access dynamically allocated memory. This is done by mapping all or part of the file to the calling process's address range. After this is done, you can access the file's data by using a simple pointer. Memory-mapped files are discussed in greater detail in Chapter 12, "Working with Files."

Heaps

Heaps are contiguous blocks of memory in which smaller blocks can be allocated. Heaps efficiently manage the allocation and manipulation of dynamic memory. Heap memory is manipulated using various `Heap`*`XXXX`*`()` Win32 API functions. These functions are listed in Table 3.3 and are also documented in detail in Delphi's online help.

TABLE 3.3 Heap Functions

Function	*Purpose*
`HeapCreate()`	Reserves a contiguous block in the virtual address space of the calling process and allocates physical storage for a specified initial portion of this block
`HeapAlloc()`	Allocates a block of nonmovable memory from a heap
`HeapReAlloc()`	Reallocates a block of memory from the heap, thus allowing you to resize or change the heap's properties
`HeapFree()`	Frees a memory block allocated from the heap with `HeapAlloc()`
`HeapDestroy()`	Destroys a heap object created with `HeapCreate()`

NOTE

It's important to note that there are several differences in the Win32 implementation of Windows NT/2000 and Windows 95/98. Generally, these differences have to do with security and speed. The Windows 95/98 memory manager, for instance, is leaner than that of Windows NT/2000 (NT maintains more internal tracking information on heap blocks). However, the NT virtual memory manager is generally regarded as much faster than Windows 95/98.

Be aware of such differences when using the various functions associated with these Windows objects. The online help will point out platform-specific variations of such a function's usage. Be sure to refer to the help whenever using these functions.

Error Handling in Win32

Most Win32 API functions return either `True` or `False`, indicating that the function was either successful or unsuccessful, respectively. If the function is unsuccessful (the function returns `False`), you must use the `GetLastError()` Win32 API function to obtain the error code value for the thread in which the error occurred.

NOTE

Not all Win32 system API functions set error codes that are accessible by `GetLastError()`. For example, many GDI routines don't set error codes.

This error code is maintained on a per-thread basis, so `GetLastError()` must be called in the context of the thread causing the error. The following is an example of this function's usage:

```
if not CreateProcess(CommandLine, nil, nil, nil, False,
  NORMAL_PRIORITY_CLASS, nil, nil, StartupInfo, ProcessInfo) then
    raise Exception.Create('Error creating process: '+
    IntToStr(GetLastError));
```

TIP

The Delphi 5 `SysUtils.pas` unit has a standard exception class and utility function to convert system errors into exceptions. These functions are `Win32Check()` and `RaiseLastWin32Error()`, which raises an `EWin32Error` exception. Use these helper routines instead of writing your own result checks.

This code attempts to create a process specified by the null-terminated string `CommandLine`. We'll defer discussing the `CreateProcess()` method for a later chapter since we're focusing on the `GetLastError()` function. If `CreateProcess()` fails, an exception is raised. This exception displays the last error code that resulted from the function call by getting it from the `GetLastError()` function. You might use a similar approach in your application.

TIP

Error codes returned by `GetLastError()` are typically documented in the online help under the functions that cause the error to occur. Therefore, the error code for `CreateMutex()` would be documented under `CreateMutex()` in the Win32 online help.

Summary

This chapter introduced you to the Win32 API. You should now have an idea of the new kernel objects available as well as how Win32 manages memory. You should also be familiar with the different memory-management features available to you. As a Delphi developer, it isn't necessary that you know all the ins and outs of the Win32 system. However, you should possess a basic understanding of the Win32 system, its functions, and how you can use these functions to maximize your development effort. This chapter provides you with a starting point.