Chapter 2

System Foundations

Before we get to the good stuff, we have some dull but necessary business to take care of. In this chapter, we'll take a quick look at some of the basic foundations of the Windows system: elementary data types, multithreaded processes, memory management, program loading and execution, resources, and the file system. After we have these preliminaries out of the way, we'll be ready to talk about the really interesting stuff: programming the Windows user interface.

The C/C++ language version of the Win32 Application Program Interface (API) is defined by a set of header files that you incorporate into your program with a C **`#include`** directive:

```
#include <windows.h>
```

The master header file, **`WINDOWS.H`**, in turn calls in other header files, such as **`WINDEF.H`** (basic type definitions), **`WINBASE.H`** (system kernel), **`WINGDI.H`** (graphics device interface—like QuickDraw on the Macintosh), **`WINUSER.H`** (user interface structures and functions), and so on. Taken together, the complete set of header files defines all of the data structures and functions that constitute the Win32 API.

Elementary Data Types

The Win32 header files define a set of elementary data types and related constants to represent things like integers, booleans, characters, pointers, floating-point numbers, and so forth. These are generally derived from the built-in standard types of the C language, but spelled in full capitals instead of lowercase: **`INT`** and **`CHAR`**, for instance, instead of **`int`** and **`char`**. The Windows versions are carefully defined with conditional-compilation flags to maintain source-code consistency between different (16- and 32-bit) versions of the Windows system. For example, type **`INT`** represents a 16-bit integer or a 32-bit integer, depending on which version of the system it's compiled for. As a rule, you should use the Windows-defined data types in your programs in place of the built-in versions.

Table 2–1 shows the most important of the elementary data types. Most of them are self-explanatory, but the last, **HANDLE**, needs a bit of further elaboration. Unlike the Macintosh-style handles that you're familiar with, a Windows handle is simply an "opaque reference" to an underlying object. The handle's internal semantics are meaningful only to the Windows system itself; it has no inherent, straightforward meaning like a Macintosh "pointer to a pointer." What's actually inside the handle is none of your program's business: it might be a simple pointer, an indirect pointer, an index into an internal system table, or who-knows-what else.

Table 2–1. Elementary data types

Type name	Meaning
VOID	Unspecified type
BYTE	Unsigned byte (8 bits)
WORD	Unsigned word (16 bits)
DWORD	Unsigned double word (32 bits)
BOOL	Boolean (**TRUE** or **FALSE**)
CHAR	Character
UCHAR	Unsigned character
INT	Signed integer
UINT	Unsigned integer
SHORT	Signed short integer (16 bits)
USHORT	Unsigned short integer (16 bits)
LONG	Signed long integer (32 bits)
ULONG	Unsigned long integer (32 bits)
FLOAT	Floating-point number
POINT	Point
RECT	Rectangle
HANDLE	Handle

This means that you can't "open up" a Windows handle and use it to put your hands directly on the underlying data structure. The only meaningful thing you can do with a handle is pass it to a Windows function and let the system manipulate the data structure for you. For instance, suppose **theWindow** is a handle to a window structure and you want to get the value of its **style** field, which contains a set of bit flags controlling the window's appearance on the screen. On the Macintosh, you would write something like this (ignoring for the moment the fact that Macintosh windows are actually referenced with simple pointers instead of handles):

```
windowStyle = (**theWindow).style;
```

That is, you'd dereference the handle twice to get at the underlying data structure and then select the structure's **style** field. You can't do this in Windows, because the handle is opaque: it isn't just a pointer to a pointer. Instead, you have to pass the window handle to a Windows function that retrieves and returns the value of the desired field. Windows provides all the functions you need to access the internal

components of its data structures. (In this case, you'd use the **GetWindowLong** function, which we'll learn about in Chapter 3.)

The Windows API header files also define a series of utility macros for manipulating the elementary data types (see Table 2–2). You can pack two bytes together into a 16-bit word with **MAKEWORD** and get them back out again with **HIBYTE** and **LOBYTE**. Similarly, **MAKELONG** packs two words into a 32-bit long word, and **HIWORD** and **LOWORD** extract them.

Table 2–2. Utility macros for elementary data types

Macro name	Meaning
MAKEWORD	Pack bytes into word
HIBYTE	Extract high-order byte from word
LOBYTE	Extract low-order byte from word
MAKELONG	Pack words into long word
HIWORD	Extract high-order wordfrom long word
LOWORD	Extract low-order wordfrom long word

Many of the types shown in Table 2–1 also have corresponding pointer types, such as **PWORD** ("pointer to word") and **LPWORD** ("long pointer to word"). In older versions of Windows, with their 16-bit addresses and segmented memory architecture, the pointer types beginning with **P** represented "near pointers" within the local memory segment, while those with **LP** stood for "far pointers" from one segment to another. As we'll see, Win32 systems no longer have segmented memory or near and far pointers. In Win32, both the **P** and **LP** pointer types represent true, 32-bit virtual-memory addresses.

Hungarian Notation

The names of pointer types such as **PWORD** and **LPWORD** are examples of a style of naming convention that is followed throughout the Windows programming interface. In this naming style, the names of data types, variables, structure members, and function parameters commonly begin with a prefix of anywhere from one to four letters identifying the general type of data they represent, followed by the main descriptive part of the name. For example, whereas a handle to a menu record in the Macintosh Toolbox is called a **MenuHandle**, in Windows it's an **HMENU** ("handle to menu").

This style of naming is called *Hungarian notation,* and it was actually invented by a real, live Hungarian. His name is Charles Simonyi (or Símonyi Károly in proper Magyar), and he's the [??? insert correct job title here ???] at Microsoft. Charles and I have known each other for a very long time, and I could tell you some amusing stories about him when he was just a teenager, newly arrived from Budapest, and the only words of English he knew were . . . but I digress. Charles is also one of the most remarkable, brilliant people I've ever met. He's the only person I know, for

instance, who can read Egyptian hieroglyphics. Unfortunately, his ideas about naming conventions can make even a simple C program read as if it were written in hieroglyphics, too.

Table 2–3. Common Hungarian prefixes

Prefix	Meaning
by	Byte (8 bits)
w	Word (16 bits, unsigned)
dw	Double word (32 bits, unsigned)
i	Integer (signed)
n	Short integer (16 bits, signed)
l	Long integer (32 bits, signed)
c	Count
f	Flag (boolean)
ch	Character
s	String (unterminated)
sz	String (zero-terminated)
pt	Point
x	Horizontal coordinate (16 bits, signed)
y	Vertical coordinate (16 bits, signed)
rgb	RGB color
fn	Function
p	Pointer
lp	Long pointer
h	Handle

Table 2–3 is your Rosetta Stone, showing some of the common Hungarian prefixes that you'll encounter in the Windows interface and what they mean. Actually, the Hungarian style is not entirely foreign to the Macintosh either. You can find traces of it in some parts of the Macintosh Toolbox (notably the Printing Manager) that were designed by programmers influenced by Charles's ideas.

I don't really mean to be too critical of those ideas, by the way—those remarks I made about Egyptian hieroglyphics were only for comic effect. Hungarian serves a very useful purpose in making an object's data type directly evident from its name. When you run across a function parameter named **dwCursorID**, for example, you know immediately that it's a double-word cursor ID; **fVisible** is a boolean visibility flag; and so on. If necessary, two or more of these prefixes can be combined, producing names such as **lpszTitle** ("long pointer to a string, zero-terminated, representing the title").

Hungarian-style names are used universally throughout the Windows programming interface, and anyone who wants to do Windows programming has to know how to read them. They're just not my style, though, and I don't use them in my own code. I prefer naming my variables in English.

Like Macintosh System 7 (or earlier versions of the Macintosh system under MultiFinder), Windows is a *multitasking* operating system. This means that a user can run more than one program at a time, switching between them by clicking in their windows on the screen. The system keeps track of each program's state of execution and switches the processor's attention from one to another as needed.

The Windows approach to multitasking is different from that of the Macintosh, however. The Macintosh uses a *cooperative multitasking* model, which relies on each program to be a "good citizen" and voluntarily give up control of the processor when it isn't needed. By using the Toolbox routine **`WaitNextEvent`** to retrieve events from its event queue, a program gives the system permission to suspend its execution and give control of the processor to some other program in its place. Eventually, the system will take advantage of one of the second program's **`WaitNextEvent`** calls to switch control back to the first. Everything works smoothly as long as every program calls **`WaitNextEvent`** often enough to give the others their turn at bat.

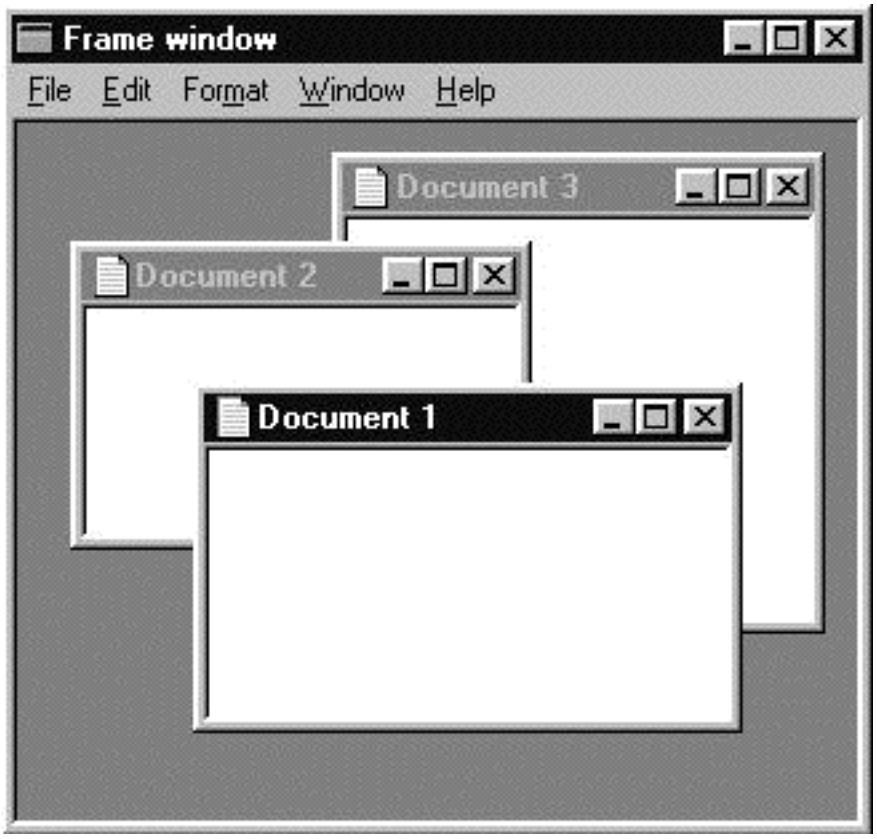
Windows, by contrast, uses a *preemptive multitasking* approach. The system itself maintains control of the processor and portions it out among the running programs in discrete "time slices." Each program behaves as if it had the computer all to itself. Programs don't have to do anything explicit to relinquish control; the system can forcibly yank the processor away from one program and turn it over to another.

Also unlike the Macintosh, Windows allows the user to run more than one *instance* of the same program simultaneously. Each separate instance of the program runs in its own *process,* independent of any others. There can be as many instances of the same program as the user wishes to start. All instances of the same program share the same copy of the program's executable code, but each has its own private set of data to work with.

This ability to create multiple instances of the same program can change the way you approach an application. My old MiniEdit program for the Macintosh, for instance, is a multiple-window text editor, allowing the user to view several text files side by side on the screen and to cut and paste text from one to another. When I ported the program to Windows, I realized that multiple windows were unnecessary. The Windows version of the program, WiniEdit, opens just one window on the screen, viewing a single file at a time. When the user asks to open a new file, it replaces the old file in the program's one window instead of opening in a separate window of its own. This is no limitation, though: a user who wants to view two or more files simultaneously can simply start up a new instance of the program for each file. Cutting and pasting from one file to another is no problem, since all such operations go through a global clipboard that's shared by all running processes.

Restricting the program to just one window and one file at a time helps simplify the code and is one reason (among others) why the Windows version of the program is significantly shorter than its Macintosh counterpart.

Figure 2–1. An MDI frame window



Programs wishing to display multiple files on the screen simultaneously can use the Win32 Multiple Document Interface (MDI). An MDI program displays a single *frame window,* which in turn hosts a variable number of subwindows ("child windows," in Windows terminology), each containing a different document. The user can open and close document subwindows as needed and can freely move, size, and arrange them within the confines of the enclosing frame window (see Figure 2–1). Many common Windows applications use the MDI model, including the Windows NT Program Manager and File Manager, Microsoft Word and Excel, and Visual C++. The Win32 API provides all the support needed to write an MDI application; see the *Win32 Programmer's Reference* for details. However, MDI also introduces a significant extra level of programming complexity. Before deciding on an MDI implementation, consider seriously whether your user might not be just as well served by multiple instances of a single-document application instead.

Windows goes the Macintosh one step further in multitasking by allowing a program to spawn multiple threads of execution. A *thread* is an independent locus of control

that shares processor time with all of the other threads in the system. (In a multiprocessor system, two or more threads might even run simultaneously on separate processors.) Your program might do background printing, for instance, by spawning a separate thread to print a document while the main thread continues to receive and process mouse and keyboard input from the user. Or you might implement your own memory allocation scheme with a background garbage collector running in a thread of its own. All of a program's threads operate within the context of a single process and share access to the same address space and other system resources. To prevent threads from interfering with each other by independently accessing the same memory or resources, Windows offers a variety of facilities for synchronizing thread operations, such as wait functions, semaphores, and mutexes (short for "*mut*ually *ex*clusive access"). We won't be discussing multithread programming and synchronization in this book, but you can learn about them in the *Win32 Programmer's Reference* if your application calls for them.

Memory has had a long and checkered history in the Intel world. Even if you've never programmed an Intel-based machine, you've probably heard horror stories from your beleaguered DOS friends about segment registers, expanded memory, extended memory, protected mode, real mode, near pointers, far pointers, huge pointers, and the infamous "640K barrier." The early versions of Windows, which functioned within the DOS environment and relied on it for their memory management, inherited all of these headaches as well.

The good news is that a new day has dawned for Windows memory management. If you're an old hand at Intel/DOS/Windows programming, you can forget everything you've ever learned about memory. (Then again, if you're such an old hand at it, why are you reading this book?) If you're a newcomer like me, you don't have to know anything about the bad old days except that they're gone. In Win32, every process (that is, every instance of every program) gets its very own, totally linear, transparently paged, 4-gigabyte virtual address space. No segments. No modes. No barriers. A pointer is a pointer is a pointer.

Of course, the operative word in the preceding paragraph is "virtual." They haven't built a personal computer yet with 4 gigabytes of RAM, and if they did, you couldn't afford to buy it. (Anyway, I know *I* couldn't!) What really happens is that the Windows operating system, with hardware support from the Intel processor (80386 or later), gives the program the *illusion* of a 4-gigabyte playpen to throw its toys around in. In practice, a program will actually use only a fraction of this vast address space—and typically, only a fraction of *that* will be physically present in RAM at any one time.

How Virtual Memory Works

On the Macintosh, virtual memory is a luxury available only on high-end models—those equipped with a paged memory-management unit, or PMMU. Even on those

systems, it's strictly optional: the user chooses whether to activate it or not (using the Memory control panel), and the system can function perfectly well without it. In Win32, virtual memory is a necessity, woven inextricably into the fabric of the system itself.

As in any virtual memory system, the Windows operating system is responsible for maintaining the correspondence between logical addresses seen by the program and physical storage locations in RAM or on the disk. To simplify the system's internal bookkeeping and reduce memory fragmentation, both the logical address space and the physical storage media are divided into units of uniform size called *pages.* A typical page size is 4K (4096) bytes, but this can vary from one hardware platform to another: the DEC Alpha implementation of Windows, for instances, uses an 8K page.

The essence of virtual memory is that not all of a program's logical address space has to be physically present in RAM at all times. The system creates a *paging file* on the disk to hold pages that are not currently in RAM, and a *page map* to keep track of each page's current location, either in RAM or on the disk. When a program attempts to refer to a logical address on a page that is not currently in RAM, a hardware *page fault* occurs. The system then locates the required page on the disk and reads it into RAM, replacing one of the pages already there. The page being replaced may belong to the same process (that is, the same instance of the same program) as the page being read in, or to some other process. If the contents of the page are "dirty" (have been changed while in RAM), they are copied back out to the disk before being overwritten. This page swap is transparent to the running program, which has the illusion of a 4-gigabyte linear address space all to itself.

Establishing the correspondence between logical addresses and physical storage locations is a two-step process. First the program *reserves* a range of addresses in its address space, known as a *region.* Reserving a region doesn't actually allocate any physical memory resources yet; it simply informs the system that the program intends to use the logical addresses within the designated range. When the program no longer needs this block of addresses, it *releases* the region to make it available for reuse.

A reserved region is always a whole number of pages in size (the system will adjust it upward, if necessary, to the nearest multiple of the page size) and always begins at an address that is a multiple of the system's *allocation granularity.* Like the page size, the allocation granularity can vary from one platform to another, but it is typically 64K (65,536) bytes. A program can use the Windows function **`GetSystemInfo`** to learn the page size and allocation granularity, as well as a variety of other parameters of the hardware platform it's running on.

Once a region of address space has been reserved, physical storage resources must be *committed* to it before it can be used. Once again, the unit of storage commitment is the page. Committing a page of address space gives it a physical location on the disk in which to hold its contents when they're not active in RAM. In committing a page, a program can specify its level of access protection: possible protection attributes include familiar ones such as read-only, read/write, execute-

only, and so on, as well as a few other, more specialized options that we'll be discussing shortly.

The program can commit all of the pages in a region at once, in a single operation, but this isn't required: they can also be committed a page at a time, as needed. When the physical storage space is no longer needed, the page can be *decommitted.*

Reserving a region of address space and committing physical storage to it can be done separately or combined in a single operation. The Windows function **`VirtualAlloc`** performs either or both operations, depending on the values of the parameters passed to it. Similarly, the **`VirtualFree`** function can decommit physical storage, release a logical region, or both.

Heap Allocation

What "memory management" means to most Macintosh programmers is allocating and freeing individual blocks of memory from a heap. Needless to say, Windows has heaps too. They don't work identically to those on the Macintosh, but there's nothing particularly mysterious about them, either.

In the older 16-bit versions of Windows, each program had its own *local heap,* along with a system-wide *global heap* that was shared in common by all programs (sort of like the Macintosh application and system heaps). A program could create blocks in either heap with the Windows functions **`LocalAlloc`** and **`GlobalAlloc`**, and destroy them with **`LocalFree`** and **`GlobalFree`**. When creating a block, you could specify whether it should be *movable* or *fixed* (analogous to Macintosh relocatable or nonrelocatable blocks) and *discardable* or *nondiscardable* (like purgeable or nonpurgeable on the Macintosh).

These old allocation functions would return a handle to the new block, but remember that a Windows handle is merely an "opaque reference" to an object, whose internal contents are meaningful only to the system. In the case of the old heap allocation functions, the handle returned was actually a 16-bit index to an entry in a descriptor table, which in turn held the address of the block along with other information about it. To gain access to the block's contents, you had to call the Windows function **`LocalLock`** or **`GlobalLock`** (depending on which heap the block was in). These functions would lock the block in place, preventing it from being moved or discarded, and return a pointer that you could use to get at its contents. There was literally no way to touch the block without locking it at the same time. When you were finished manipulating the block, you would unlock it again with **`LocalUnlock`** or **`GlobalUnlock`**, after which your pointer to it was no longer valid:

```
blockHandle = LocalAlloc(...);          // Allocate block from local heap
. . . ;
blockPtr = LocalLock (blockHandle);        // Lock block and get pointer
   . . . *blockPtr . . . ;                 // Use pointer to access block
LocalUnlock (blockHandle);                 // Unlock block; pointer no longer valid
```

Handles still exist in Win32, but only for objects that the system creates on your behalf, such as windows or menus. (Needless to say, they're also 32 bits long now

instead of 16.) The old **LocalAlloc** and **GlobalAlloc** and their related functions have been replaced with a new series called **HeapAlloc**, **HeapFree**, and so on. The old calls are still supported, but only for the sake of backward compatibility with

existing code. New programs being developed today are supposed to use the new functions exclusively.

The biggest difference between the old and new heap functions is that with the new ones there are no movable or discardable blocks and no heap compaction. (There is a function called **HeapCompact**, but all it does is coalesce adjacent free blocks and decommit empty pages; it never moves or discards any existing blocks.) I guess the theory is that with 4 gigabytes of address space available, there's always room to grow the heap if you have to, so compaction is unnecessary. Whatever the reason, the fact is that every block in a Win32 heap is fixed and nondiscardable, and is referenced with a direct pointer instead of a handle.

Table 2–4. Win32 heap allocation functions

Function	Mac counterpart	Purpose
HeapCreate	**InitZone**	Create heap
HeapDestroy	———————	Destroy heap
GetProcessHeap	**ApplicationZone**	Get program's default heap
HeapAlloc	**NewHandle**, **NewPtr**	Allocate block from heap
HeapFree	**DisposHandle**, **DisposPtr**	Deallocate block from heap
HeapSize	**GetPtrSize**	Get size of block in heap
HeapReAlloc	**SetPtrSize**	Change size of block in heap

Table 2–4 shows the basic functions in the new Win32 series. **HeapAlloc** creates a new block of a specified size and returns a pointer (not a handle!) to it. **HeapFree** deallocates an existing block; all pointers to the block become invalid and must not be used again. **HeapSize** returns the size of an existing block in bytes; **HeapReAlloc** changes the size of an existing block.

> The **HeapReAlloc** function can always shorten a block in place, but if you ask to lengthen the block, it may need to be moved to another location within the heap in order to find the needed space. Instead of updating a master pointer as on the Macintosh, **HeapReAlloc** simply returns a pointer to the block's new location; it's up to you to use the new pointer in place of any previous ones you may have been holding to the block. Of course, this can lead to trouble if you have lots of copies of the old pointer sitting around embedded in other data structures. If you want to reallocate a block to a bigger size, you'd better make sure you have a way of tracking down and updating all of the existing pointers to it.

A little-used feature of the Macintosh Toolbox is the ability to maintain multiple heaps (called "heap zones" on the Macintosh). All Macintosh memory allocation routines operate implicitly on the "current heap zone." This is normally the application heap, but you can switch to another zone with the Toolbox routine **SetZone**. Win32 allows you the same capability, but instead of using a "current

heap," each of the new heap allocation functions expects you to tell it explicitly what heap to operate on by supplying a handle to the heap as one of the function's parameters.

Usually, you'll just want to use your *process heap* for all of your memory allocation. Every instance of a Windows program has its own process heap, created automatically by the system when the instance is started up. Unless you're doing something fancy, you can simply obtain a handle to your process heap from the Win32 function **`GetProcessHeap`** and pass it as a parameter whenever you call one of the heap functions. Sometimes, though, you may find it useful to create additional heaps with the **`HeapCreate`** function. For instance, if your program creates and destroys lots of memory blocks of a certain uniform size (nodes in a linked list or tree structure, perhaps), you can manage memory more efficiently by giving them a heap of their own. Since all blocks in this heap are the same size, it can never become fragmented: as long as there's any free space at all, it's guaranteed to be the right size for the block you want to create. For a few other ways to use multiple heaps, see Jeffrey Richter's book, *Advanced Win32 Programming.*

To create a new heap zone on the Macintosh, you allocate a large nonrelocatable block from within your existing application heap (or even, in rare cases, from the stack) and pass it to the Toolbox function **`InitZone`** to give it the internal data structures it needs to function as an independent heap. In effect, the new zone becomes a "subheap" inside your main application heap. In Win32, the **`HeapCreate`** function creates a separate new heap in its own region of address space, rather than inside another existing heap. You specify the new heap's initial and maximum size; **`HeapCreate`** does the following:

- Reserves the specified maximum number of bytes in your address space
- Commits the specified initial amount of physical disk space
- Gives you back a handle to the new heap, which you can then pass to any of the other heap allocation functions

If your subsequent allocation requests exceed the initial size of the heap, the system will automatically commit additional pages of physical storage as needed, up to the reserved maximum. When you're finished using the new heap, you can decommit the storage it occupies with **`HeapDestroy`**. If you don't explicitly destroy the heap, its storage is automatically decommitted when your program terminates.

Memory-Mapped Files

One of the side benefits of the way Windows virtual memory works is a new way of doing file input and output. To read or write a file on the Macintosh, you have to allocate an input/output buffer in your heap and then explicitly transfer data between the buffer and the file, using either the high-level input/output routines **`FSRead`** and **`FSWrite`** or their low-level, parameter-block-based counterparts **`PBRead`** and **`PBWrite`**. You can do your I/O the same way in Windows, using the file transfer functions **`ReadFile`** and **`WriteFile`**—my WiniEdit example program does it this way,

for instance—but often it’s more convenient to use a *memory-mapped file* instead.

The idea of memory-mapped files follows from the way the Windows virtual memory system is structured. Although the disk space backing up a page of memory normally comes from the system's paging file, an entry in the page map can actually point anywhere at all on the disk. In particular, it can point to a disk page that's part of an ordinary data file. This makes it possible to access the contents of a file by simply mapping them directly into a region of your program's virtual address space. You can then fetch or store into any byte of the file just as if it were an ordinary memory location—because, in effect, that's just what it is. The system transparently handles all the details of physically transferring the data in and out to the disk.

The first step in memory-mapping a file is to create a *file mapping,* using a Win32 function named—guess what?—**`CreateFileMapping`**. The mapping is a system-created object that identifies the file along with some other security and protection attributes to control the type of access allowed. Once you have a file mapping, you can use it to open a *view* of all or part of the file within your virtual address space. The **`MapViewOfFile`** function returns a pointer to the region of address space containing the file; you can then access the file's contents by indexing from this pointer:

```
fileHandle = CreateFile (...);                  // Open the file
fileMap    = CreateFileMapping (fileHandle,...); // Create mapping
fileBase   = MapViewOfFile (fileMap,...);         // Open a view

. . . fileBase[i] . . . ;                         // Access file as an array
. . . *(fileBase + i) . . . ;                     //    or by pointer arithmetic

UnmapViewOfFile (fileBase);                     // Close view when finished
CloseHandle (fileMap);                          // Destroy mapping
```

Memory-mapped files are useful in their own right, just for the convenient access they offer to the contents of a file. In addition, they provide a way for separate processes or threads to communicate with one another, by sharing information through independent views of the same file. This is perfectly safe as long as only one of the processes can write to the file, with the rest restricted to read-only access. It can get tricky, though, if two or more processes try to write to the file independently. One necessary precaution is to make sure all of the processes' views of the file are based on the same mapping object. This ensures that the views remain *coherent,* meaning that any changes made via one view will be seen by the others as well. In addition, the processes must use some sort of synchronization mechanism, such as a mutex, to ensure that only one process at a time has permission to write to the file. Unfortunately, this whole topic of interprocess synchronization is beyond the scope of this book; see the *Win32 Programmer's Reference* or the Jeffrey Richter book, *Advanced Win32 Programming,* for more details.

Now comes the really cool part. If any file on the disk can be mapped into a program's virtual address space, how about the file containing the executable code of the program itself? In fact, this is just the way the Windows system loads a program

into memory for execution: it simply opens a memory-mapped view of the code from the program's executable file (which normally carries the file extension **`.exe`**, a vestige of the old DOS file system) within the program's virtual address space.

Opening a view of the executable file does not, in itself, cause any code to be loaded from the disk; but when the system attempts to transfer control to the program's main entry point, a page fault will occur that will load in the first page of code and begin executing. Thereafter, any time the program transfers control to an address that is not already in RAM, the virtual memory system will bring in the missing page automatically. There's no need to break the code up into explicitly defined segments, as on the Macintosh; code swapping takes place transparently as part of the routine workings of virtual memory.

Notice that this scheme makes it easy for multiple instances of a program to share the same copy of the code: the system simply maps the same executable file into each instance's address space, allowing them to page in portions of the code as needed. If two instances happen to be executing in the same page of code, they can share the same physical copy of that page in RAM.

Of course, each instance has to have its own private region of address space to hold its copy of the program's global variables. The way Windows manages this is also instructive. Space for the global variables is reserved within the program's executable file. When these pages are mapped into a process's address space, they are given a special protection attribute called *copy-on-write.* This allows the process to read the page's contents freely, so long as it doesn't try to modify them. The first time the process attempts to write to such a page, the system automatically creates a new copy of the page in the system paging file and maps the new copy into the process's address space in place of the original copy from the program's executable file. Multiple instances of the same program can thus share the identical copy of the data page as long as they're only reading from it, while getting their own private copies to work with if they have to write into it.

One of the key early design decisions on the Macintosh was to have the executable code of the Toolbox reside permanently in ROM. Calls from a running application program to a Toolbox routine are implemented via the Motorola processor's trap mechanism. Each Toolbox call generates a *trap word* that looks just like an ordinary machine-language instruction, but actually triggers the processor's *unimplemented instruction* trap. This activates the Macintosh system's Trap Dispatcher, which decodes the trap word to determine what Toolbox routine it refers to, locates the routine's entry point in ROM, and transfers control to it. (All this is different, of course, on the new-fangled Power Macs.)

As you might expect, Windows system calls work differently. The Intel-based machines that Windows typically runs on have no built-in ROM code and no processor emulation mechanism similar to the Motorola unimplemented instruction trap. All of the code for the Windows system resides on the hard disk in a collection

of *dynamic link libraries,* or *DLLs.* DLLs are as fundamental to the operation of Windows as the trap mechanism is to the Macintosh Toolbox.

A dynamic link library is simply a collection of machine-language routines that are available to be called by any program that needs them. There are many DLLs that make up the Windows system, but the most important of them are **KERNEL.DLL**, which contains the code for low-level system functions such as memory and process management; **GDI.DLL**, which contains the Graphics Device Interface (general-purpose graphics routines); and **USER.DLL**, which implements the high-level Windows user interface. Notice that these three libraries correspond to the three main divisions of the Macintosh ROM code: the Macintosh Operating System, QuickDraw, and the User Interface Toolbox.

The reason they're called *dynamic* link libraries is that the program's references to routines in the library are resolved dynamically, when the program is loaded and run, rather than statically, at build time. Instead of linking the library code directly into the program's executable file, the Windows linker simply compiles a table of system routines the program calls and the DLLs in which they reside. When the program is run, the system locates the needed DLLs on the disk, maps them into the program's virtual address space, and patches each table entry to the correct virtual address. This allows multiple running programs to share the same copy of the system code, leaving it to the virtual memory system to page the code in from the disk only when it's actually needed.

I've always considered resources the key to Macintosh programming. Although they were first invented to serve a very limited purpose—translating onscreen messages and menu commands for use in foreign countries—they turned out to be such a useful device that they have grown to take over the entire Macintosh software architecture. Many of the familiar software conveniences that Macintosh users know and love—desk accessories, **INIT** extensions, control panels, Chooser devices—are made possible by resources. I suspect that if Apple were to redesign the Macintosh Toolbox from scratch today, they would scrap the idea of a file's data fork entirely and just make everything a resource and every file a resource file.

Windows has resources too, but they're not as pervasive a part of the overall software architecture as they are on the Macintosh. Unlike the Macintosh, with its hundreds of resource types serving every conceivable purpose, Windows has only about a dozen built-in resource types, shown in Table 2–5. Their only purpose is to isolate certain aspects of a program's operation so that they can be easily modified without affecting the code itself.

Most of the resource types shown in the table are self-explanatory, but a few of them bear further discussion. As we'll learn in Chapter 7, Windows uses the term *menu* to refer, not only to the individual lists of items that we think of as menus on the Macintosh, but also to what the Macintosh calls a *menu bar:* the complete set of individual menus that a program offers to the user. So the Windows **RT_MENU**

resource is actually a combination of the Macintosh resource types **'MBAR'** and **'MENU'**. (The prefix **RT_** stands for “resource type.”) Also, a Macintosh **'MENU'** resource includes information about Command-key shortcuts that the user can use to invoke menu items directly from the keyboard. In Windows, these shortcuts are called *keyboard accelerators* and are defined in a separate *accelerator table* obtained from a resource of type **RT_ACCELERATOR**. The Windows resource type **RT_STRING** is not a single text string like a Macintosh **'STR '** resource, but rather a *string table,* like **'STR#'** on the Macintosh. (Individual strings within the table have their own resource IDs, however, rather than a single ID for the whole table and a separate index for each string, as on the Macintosh.) Finally, the **RT_GROUP_CURSOR** and **RT_GROUP_ICON** resources are bundles of alternative versions of a cursor or icon, for use on display screens of different depths and resolutions. When your program asks to load the cursor or icon, Windows checks the properties of the current display device on the user’s machine and automatically chooses the version most suitable for that device.

Table 2-5. Standard resource types

Type name	Access function	Meaning
RT_MENU	**LoadMenu**	Menu
RT_ACCELERATOR	**LoadAccelerators**	Accelerator table
RT_DIALOG	——————	Dialog box
RT_FONT	——————	Font
RT_FONTDIR	——————	Font directory
RT_CURSOR	**LoadCursor**	Cursor
RT_GROUP_CURSOR	——————	Cursor
RT_ICON	**LoadIcon**	Icon
RT_GROUP_ICON	——————	Icon
RT_STRING	**LoadString**	String table
RT_BITMAP	**LoadBitmap**	Bitmap
RT_MESSAGETABLE	**FormatMessage**	Message table entry
RT_VERSION	——————	Version data
RT_RCDATA	——————	Application-defined resource

Resource Identification

Since Windows files don’t have separate resource forks, resources have to reside in the main body of a file (what the Macintosh would call its data fork). In the context of resource management, a file containing resources is called a *resource module.* Three types of file can serve as resource modules:

- Executable program files (file extension **.exe**)
- Dynamic link libraries (file extension **.dll**)
- Font files (file extension **.fon**)

A program's own private resources typically reside in its executable file. There are also some system-wide standard resources that live in the Windows DLLs. Windows has no equivalent to the Macintosh notion of a "current resource file": all Windows functions for retrieving resources accept an explicit parameter to identify the module in which to look for the desired resource. Typically, this is a handle to a program instance, designating the program's executable file as the resource module; a **NULL** value denotes one of the standard system-defined resources instead.

As on the Macintosh, a Windows resource is designated by its resource type and an individual identifier within that type. In Windows, each of these two items can be given as either a character string or an integer (somewhat like identifying a Macintosh resource either by name or by ID number). Although either method of resource identification is acceptable, integer identifiers are considered preferable to strings because they use less memory space. Formally, all Windows functions that operate on resources take string parameters for both the resource type and the individual ID; but as we'll see, there are a variety of ways to pass them the identifiers in integer form instead.

Resource Access

In the general case, accessing a resource is a two-stage process: first you have to locate the resource within its resource module, then load its contents into memory. The general-purpose Windows functions for these two operations are **FindResource** and **LoadResource**. **FindResource** takes three parameters, specifying the resource's module, identifier, and type. It returns a handle, not to the resource itself, but to the information block describing it in the resource module. (If the requested resource can't be found, the function returns **NULL**.) You must then pass this handle, in turn, to **LoadResource** (along with the parameter identifying the resource module again) to get a handle to the data of the resource itself. For example the following statements find and load the system's standard arrow cursor:

```
rsrcHandle = FindResource(NULL, "IDC_ARROW", "RT_CURSOR");// Find resource info
datahandle = LoadResource(NULL, rsrcHandle);              // Load resource data
```

(The string **"IDC_ARROW"** is the name of the resource containing the standard arrow cursor; the prefix **IDC_** stands for "identifier of a cursor.")

In practice, however, it usually isn't necessary to perform these two separate steps explicitly. Many of the standard resource types have their own special-purpose access functions (listed in Table 2–5), which combine the two steps into a single operation. To load the standard arrow cursor, for instance, you'll normally use the **LoadCursor** function rather than **FindResource** and **LoadResource**:

```
arrowHandle = LoadCursor(NULL, "IDC_ARROW");
```

This function already knows what resource type to look for, so you don't have to spell it out with a parameter. It performs the operations of both **FindResource** and **LoadResource** in a single call, and gives you back a specifically-typed handle of type **HCURSOR**. The other functions listed in the table work similarly: **LoadMenu** returns a handle of type **HMENU**, **LoadIcon** a handle of type **HICON**, and so on.

Although the resource ID parameter to each of these functions is nominally defined as a string (that is, as a pointer to a character), you can actually pass an integer value instead. If the high-order word of the 32-bit parameter is zero, the resource-access functions will recognize it as an integer resource identifier instead of a string pointer. If you happened to know, for instance, that the numerical identifier for the **IDC_ARROW** cursor is **32512**, you could pass that value as the second parameter to **LoadCursor**. Of course, you would first have to pad the 16-bit integer value to 32 bits and typecast it into a string pointer. As a convenience, the Win32 interface defines a C macro named **MAKEINTRESOURCE** that does just that:

```
#define MAKEINTRESOURCE(i) (LPSTR)((DWORD)((WORD)(i)))
```

(The actual definition is a bit more complex than this, but conceptually this is the way it works.) So to load the arrow cursor, you could use the following code:

```
rsrcID      = MAKEINTRESOURCE(32512);
arrowHandle = LoadCursor(NULL, rsrcID);
```

If the first character of the parameter string is a number sign (**#**), the resource-access functions interpret the string as a decimal integer representing the resource ID—so you could also load the arrow cursor this way:

```
arrowHandle = LoadCursor(NULL, "#32512");
```

Of course, it isn't very good coding style to hard-wire that resource number directly into your code. The Win32 headers define the names of all the standard resources as constants, already padded and typecast with **MAKEINTRESOURCE** and ready to use:

```
#define IDC_ARROW  MAKEINTRESOURCE(32512)
```

So instead of using the integer value directly, you can just use the resource name as a parameter to the resource-access function:

```
arrowHandle = LoadCursor(NULL, IDC_ARROW);
```

Notice that this is not the same as the first example we looked at: there are no string quotes around the name this time. In the first example, the parameter we passed was a string containing the name of the desired resource; here, it's a constant representing the integer resource ID, padded to 32 bits and typecast to an appropriate string pointer.

When you're working with the standard resource types, you usually aren't interested in opening up the resource and looking at the data inside; you simply get a handle to the resource and pass it off to some other Windows function to operate on. If you define your own custom resource types, however, only you know what's inside them—so the only way to make use of them is to reach in and put your hands directly on the internal data. To do this, you have to convert the opaque handle you receive from **LoadResource** into an actual pointer to the beginning of the resource's data. The Windows function that does this conversion for you is **LockResource**. In older versions of the Windows system, **LockResource** did what its name implies: it locked the resource in place in the heap, so that it couldn't slide out from under your pointer in the event of a heap compaction. As we've seen, the Win32 memory manager no longer moves blocks around in the heap; but you still use the **LockResource** function to get a pointer to a resource's data:

```
dataPtr = LockResource(dataHandle);       // Convert handle to pointer
. . . dataPtr[i] . . . ;                 // Access data as an array
. . . *(dataPtr + i) . . . ;                //    or by pointer arithmetic
UnlockResource(dataHandle);              // Unlock when finished
```

The closing call to **UnlockResource** is no longer needed (and in fact has no effect at all in Win32), but you can still retain it for the sake of symmetry and good form.

Defining Your Own Resources

Of course, many if not most Windows programs will need to define some resources of their own, in addition to the standard ones built into the Windows system. The Visual C++ development system includes a resource compiler similar to Rez in the Macintosh Programmer's Workshop. The resource compiler accepts a text file (file extension **.rc**, for "resource compiler") containing a description of the program's resources and compiles the resources themselves into the program's executable file. For illustration, Listing 2–1 shows the contents of **WiniEdit.rc**, the resource description file for my WiniEdit example application program. In all, the file defines six resources: an icon for representing the program on the screen, a menu, an accelerator table (which defines keyboard shortcuts for the program's menu commands), a dialog box for the **About WiniEdit...** command, and two string tables containing various bits and pieces of text that the program uses.

Luckily, it isn't necessary to learn the syntax of the resource description language and compose your own resource descriptions in text form. The Visual C++ software development environment includes a set of interactive resource editors that allow you to create and manipulate resources directly onscreen with your mouse and keyboard, much like the Macintosh utility program ResEdit. There's a resource editor for menus, one for dialog boxes, and so on. The resource editors are fully integrated into the development environment. As you build your resources onscreen, the editors automatically generate the corresponding text descriptions and place them in the resource description file for you. Later, when you build your program, your Visual C++ make file (also generated for you automatically) will call the resource compiler to compile the resource descriptions into resources and place them in your executable file. In the normal course of events, you never have to deal directly with the contents of the resource description file yourself. All of the text in Listing 2–1 was generated for me by the Visual C++ development software, untouched by human hands. I don't fully understand everything that's in this file, and that's just fine with me.

Listing 2–1. WiniEdit resource description file

```
//Microsoft Visual C++ generated resource script.
//
#include "WiniEdit Resources.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

#ifdef APSTUDIO_INVOKED
////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "WiniEdit Resources.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

////////////////////////////////////////////////////////////////////////////
#endif    // APSTUDIO_INVOKED
```

Listing 2–1. WiniEdit resource description file (*continued*)

```
//////////////////////////////////////////////////////////////////////////
//
// Dialog
//

About_Dialog DIALOG DISCARDABLE  32, 32, 163, 96
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE
FONT 8, "MS Sans Serif"
BEGIN
    ICON            ProgIcon_ID,About_Dialog,8,8,18,20
    CTEXT           "WiniEdit 1.0",IDC_STATIC,61,12,41,8
    CTEXT           "Example Windows application",IDC_STATIC,32,32,98,8
    LTEXT           "S. Chernicoff",IDC_STATIC,8,52,44,8
    RTEXT           "15 January 1995",IDC_STATIC,100,52,55,8
    DEFPUSHBUTTON   "Continue",IDOK,60,68,43,20
END

//////////////////////////////////////////////////////////////////////////
//
// Menu
//

Main_Menu MENU PRELOAD DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&New\tCtrl+N",                New_Item
        MENUITEM "&Open...\tCtrl+O",            Open_Item
        MENUITEM "&Close\tCtrl+W",              Close_Item
        MENUITEM SEPARATOR
        MENUITEM "&Save\tCtrl+S",               Save_Item
        MENUITEM "Save &As...\tCtrl+Alt+S",     SaveAs_Item
        MENUITEM "&Revert to Saved...\tCtrl+R", Revert_Item
        MENUITEM SEPARATOR
        MENUITEM "Page Set&up...\tCtrl+Alt+P",  Setup_Item
        MENUITEM "&Print...\tCtrl+P",           Print_Item
        MENUITEM SEPARATOR
        MENUITEM "E&xit\tCtrl+Q",               Exit_Item
    END
    POPUP "&Edit"
    BEGIN
        MENUITEM "&Undo\tCtrl+Z",               Undo_Item
        MENUITEM SEPARATOR
        MENUITEM "Cu&t\tCtrl+X",                Cut_Item
        MENUITEM "&Copy\tCtrl+C",               Copy_Item
        MENUITEM "&Paste\tCtrl+V",              Paste_Item
        MENUITEM "&Delete\tDelete",             Delete_Item
        MENUITEM SEPARATOR
        MENUITEM "Select &All\tCtrl+A",         SelectAll_Item
    END
```

Listing 2–1. WiniEdit resource description file (*continued*)

```
   POPUP "&Format"
    BEGIN
        MENUITEM "Text &Format...\tCtrl+F",     Format_Item
        MENUITEM "&Default Format\tCtrl+D",     Default_Item
        MENUITEM SEPARATOR
        MENUITEM "&Background Color...\tCtrl+B", Background_Item
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&Help\tCtrl+?",               Help_Item
        MENUITEM SEPARATOR
        MENUITEM "&About WiniEdit...",          About_Item
    END
END

////////////////////////////////////////////////////////////////////////////
//
// Icon
//

ProgIcon_ID             ICON    DISCARDABLE     "WiniEdit.ico"

////////////////////////////////////////////////////////////////////////////
//
// Accelerator
//

Accel_ID ACCELERATORS PRELOAD DISCARDABLE
BEGIN
    "A",            SelectAll_Item,         VIRTKEY, CONTROL, NOINVERT
    "B",            Background_Item,        VIRTKEY, CONTROL, NOINVERT
    "C",            Copy_Item,              VIRTKEY, CONTROL, NOINVERT
    "D",            Default_Item,           VIRTKEY, CONTROL, NOINVERT
    "F",            Format_Item,            VIRTKEY, CONTROL, NOINVERT
    "N",            New_Item,               VIRTKEY, CONTROL, NOINVERT
    "O",            Open_Item,              VIRTKEY, CONTROL, NOINVERT
    "P",            Print_Item,             VIRTKEY, CONTROL, NOINVERT
    "P",            Setup_Item,             VIRTKEY, CONTROL, ALT, NOINVERT
    "Q",            Exit_Item,              VIRTKEY, CONTROL, NOINVERT
    "R",            Revert_Item,            VIRTKEY, CONTROL, NOINVERT
    "S",            Save_Item,              VIRTKEY, CONTROL, NOINVERT
    "S",            SaveAs_Item,            VIRTKEY, CONTROL, ALT, NOINVERT
    "V",            Paste_Item,             VIRTKEY, CONTROL, NOINVERT
    VK_BACK,        Undo_Item,              VIRTKEY, ALT, NOINVERT
    VK_DELETE,      Cut_Item,               VIRTKEY, SHIFT, NOINVERT
    VK_F1,          Help_Item,              VIRTKEY, NOINVERT
    VK_F2,          Cut_Item,               VIRTKEY, NOINVERT
    VK_F3,          Copy_Item,              VIRTKEY, NOINVERT
    VK_F4,          Paste_Item,             VIRTKEY, NOINVERT
    VK_HELP,        Help_Item,              VIRTKEY, CONTROL, NOINVERT
```

Listing 2–1. WiniEdit resource description file (*continued*)

```
   VK_HELP,        Help_Item,              VIRTKEY, SHIFT, CONTROL,
                                                    NOINVERT
    VK_INSERT,      Copy_Item,              VIRTKEY, CONTROL, NOINVERT
    VK_INSERT,      Paste_Item,             VIRTKEY, SHIFT, NOINVERT
    "W",            Close_Item,             VIRTKEY, CONTROL, NOINVERT
    "X",            Cut_Item,               VIRTKEY, CONTROL, NOINVERT
    "Z",            Undo_Item,              VIRTKEY, CONTROL, NOINVERT
END

////////////////////////////////////////////////////////////////////////////
//
// String Table
//

STRINGTABLE DISCARDABLE
BEGIN
    ProgName_Str            "WiniEdit"
    NoTitle_Str             "Untitled"
    FileFilter_Str          "WiniEdit files (*.wed)|*.wed|Plain text files (*.txt)|*.txt|ASCII files (*.asc)|
*.asc*|All text files (*.wed, *.txt, *.asc)|*.wed;*.txt;*.asc|All files (*.*)|*.*||"
    FileExt_Str             "wed"
END

STRINGTABLE DISCARDABLE
BEGIN
    Save_Msg                "Save document ""%s"" before closing?"
    Revert_Msg              "Revert to most recently saved version of document ""%s""?"
    DefaultFormat_Msg       "Revert document ""%s"" to standard text format?"
    WrongType_Msg           "Sorry, WiniEdit works with text documents only. Can't read or write document
""%s""."
    TooLong_Msg             "Sorry, document ""%s"" is too long for WiniEdit to read."
    OutOfMem_Msg            "Out of memory!"
    IOError_Msg             "Unanticipated input/output error #%s."
END

#ifndef APSTUDIO_INVOKED
////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//

////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

As you compose your resources with the onscreen editors, you can assign them any names and ID numbers you choose. In addition to the main resource description file, the editors also generate a resource header file containing C definitions for the resource names and IDs you've chosen, which you can incorporate into your program's source code with a C **#include** directive. Listing 2–2 shows WiniEdit's resource header file. In this case (unlike the resource description file in Listing 2–1), I've done some editing on the raw output produced by the development software, just to pretty the file up a bit for the benefit of my human readers—things like grouping the resource IDs by type instead of in strict numerical order the way the development software produces them. But again, the essential contents of the file were generated automatically by the development system, and again, they include a few arcane incantations that I don't really understand and don't care to—as long as they work (which they do). After including the contents of this file in my source code with the directive

```
#include "WiniEdit Resources.h"
```

I can use the resource IDs it defines to load my resources with statements like

```
resourceID = MAKEINTRESOURCE(ProgIcon_ID);              // Convert resource ID
progIcon   = LoadIcon(ThisInstance, resourceID);        // Load icon
```

and

```
resourceID = MAKEINTRESOURCE(Accel_ID);                 // Convert resource ID
AccelTable = LoadAccelerators(ThisInstance, resourceID);  // Load accelerator table
```

and

```
LoadString (ThisInstance, NoTitle_Str,                  // Get default title
            NoNameTitle, TitleMax);
```

Notice that each of these examples uses the variable **ThisInstance** to identify the resource module in which to look for the desired resource. This is a global program variable that WiniEdit uses to hold a handle to the currently running instance of the program. Supplying this handle as the resource module parameter tells the Windows resource functions to look for the requested resource in the program's executable file, **WiniEdit.exe**.

Listing 2–2. WiniEdit resource header file

```
//
//                              WiniEdit Resources.h
//             Resource header for example Windows application program
//                       S. Chernicoff         15 January 1995
//

//       Global resource header file for WiniEdit example application program

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

// Icon

#define ProgIcon_ID                     1000

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

// Menus

#define Main_Menu                       1000

#define File_Menu                   0
#define    New_Item                     1001
#define    Open_Item                    1002
#define    Close_Item                   1003
#define    Save_Item                    1004
#define    SaveAs_Item                  1005
#define    Revert_Item                  1006
#define    Setup_Item                   1007
#define    Print_Item                   1008
#define    Exit_Item                    1009

#define Edit_Menu                   1
#define    Undo_Item                    1101
#define    Cut_Item                     1102
#define    Copy_Item                    1103
#define    Paste_Item                   1104
#define    Delete_Item                  1105
#define    SelectAll_Item               1106

#define Format_Menu                 2
#define    Format_Item                  1201
#define    Default_Item                 1202
#define    Background_Item              1203

#define Help_Menu                   3
#define    Help_Item                    1301
#define    About_Item                   1302

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

Listing 2–2. WiniEdit resource header file (*continued*)

```
// Accelerators

#define Accel_ID                        1000

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

// Control ID

#define Edit_Control                    1000

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

// Dialog

#define About_Dialog                    1000

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

// Strings

#define ProgName_Str                    1001
#define NoTitle_Str                     1002
#define FileFilter_Str                  1003
#define FileExt_Str                     1004

#define Save_Msg                        2001
#define Revert_Msg                      2002
#define DefaultFormat_Msg               2003
#define WrongType_Msg                   2004
#define TooLong_Msg                     2005
#define OutOfMem_Msg                    2006
#define IOError_Msg                     2007

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

// Next default values for new objects

#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        102
#define _APS_NEXT_COMMAND_VALUE         40004
#define _APS_NEXT_CONTROL_VALUE         1000
#define _APS_NEXT_SYMED_VALUE           102
#endif
#endif
```

Processes and Threads

• The Macintosh supports multitasking under System 7 (or earlier system versions with MultiFinder).	• Win32 supports multitasking.

Memory

• The Macintosh has a linear, unsegmented address space.	• Win32 has a linear, unsegmented address space.
• A Macintosh program can allocate blocks of memory from the heap as needed.	• A Win32 program can allocate blocks of memory from the heap as needed.
• A Macintosh program can have multiple independent heap zones.	• A Win32 program can have multiple independent heaps.

File Input/Output

• Macintosh programs can read or write files sequentially with the Toolbox routines **`FSRead`** and **`FSWRite`** (or **`PBRead`** and **`PBWrite`**).	• Win32 programs can read or write files sequentially with the Windows functions **`ReadFile`** and **`WriteFile`**.

Resources

• Macintosh programs use resources to allow aspects of the program's operation to be modified without changing its executable code.	• Win32 programs use resources to allow aspects of the program's operation to be modified without changing its executable code.
• A Macintosh resource is identified by a resource type and either a name or an integer ID number.	• A Win32 resource is identified by a resource type and either a name or an integer ID number.
• On the Macintosh, commonly used system resources are globally available in the system resource file.	• In Win32, commonly used system resources are globally available in the system resource modules.
• On the Macintosh, you can compile resources from a text description with the resource compiler Rez, or create them interactively on-screen with the resource editor ResEdit.	• In Win32, you can compile resources from a text description with the Visual C++ resource compiler, or create them interactively on-screen with specialized resource editors available

in the Visual C++ software development environment.

...Only Different

Processes and Threads

• The Macintosh uses cooperative multitasking.	• Win32 uses preemptive multitasking.
• No more than one instance of a Macintosh program can be active at a time.	• Win32 can run multiple instances of a program at the same time, each in its own process.
• To allow the user to view multiple documents at the same time, a Macintosh program must maintain multiple windows on the screen, one for each document.	• A Windows program needn't be able to open more than one document window at a time; the user can view multiple documents by running multiple instances of the program.
• A Macintosh program has a single thread of execution.	• A Win32 program can spawn multiple threads of execution.

Memory

• On the Macintosh, virtual memory is an option available only on some hardware models.	• In Win32, virtual memory is an integral part of the system.
• On the Macintosh, the user chooses whether to enable virtual memory, using the Memory control panel.	• In Win32, virtual memory is always enabled and cannot be turned off.
• On the Macintosh, all virtual memory allocation is handled transparently by the system.	• In Win32, a running program can explicitly reserve and release regions of its address space and commit and decommit physical pages to them.
• The Macintosh Memory Manager can compact the heap by relocating or purging blocks to create more usable free space.	• The Win32 Memory Manager cannot compact the heap; it can only expand it by committing additional pages of physical storage to it.
• Blocks in the Macintosh heap can be either relocatable or nonrelocatable, purgeable or nonpurgeable.	• Blocks in the Win32 heap are always immovable and nondiscardable.

• A block in the Macintosh heap may be identified by either a handle or a simple pointer, depending on whether it is relocatable or nonrelocatable.	• A block in the Win32 heap is always identified by a simple pointer.
• A Macintosh handle is an indirect reference to the underlying data structure: a pointer to the object's master pointer.	• A Win32 handle is an opaque reference to the underlying data structure: its internal contents are meaningful only to the Windows system itself.
• By dereferencing a Macintosh handle twice, you can gain access to the underlying data structure and manipulate its internal fields directly.	• The only way to manipulate the internal fields of a Win32 data structure is indirectly, via a Windows function provided for the purpose.
• On the Macintosh, alternate heap zones are created from within the existing application heap.	• In Win32, alternate heaps occupy separate regions of the process's address space.
• Blocks in the Macintosh heap are always allocated implicitly from the current heap zone. To use a different heap zone, you have to make the desired zone current.	• Win32 heap allocation functions take a handle to a heap as an explicit parameter.
• On the Macintosh, every page of virtual memory is backed by the system's paging file on the hard disk.	• In Win32, a page of virtual memory may be backed by any file on the hard disk.

File Input/Output

• Macintosh programs can only read or write a file sequentially, using **`FSRead`** and **`FSWrite`** (or **`PBRead`** and **`PBWrite`**).	• Win32 programs can read or write a file either sequentially, using **`ReadFile`** and **`WriteFile`**, or randomly, by mapping the file directly into a region of the program's virtual address space.

Program Loading and Execution

• On the Macintosh, if you want to swap parts of your program's code in and out of memory during execution, you must explicitly break up the program into separate code segments, which reside on the disk as separate **`'CODE'`** resources.	• A Win32 program has no explicit code segments, just a linear stream of code in the program's executable (**`.exe`**) file.
• The Macintosh system must physically load each of a program's code segments into memory from the disk in order to execute it.	• The Windows system simply maps the code from a program's executable file into its virtual address space, allowing the virtual memory system to load individual pages as needed.
• The executable code of the Macintosh Toolbox resides permanently in read-only memory.	• The executable code of the Windows system resides on the disk in a set of dynamic link libraries.
• A Macintosh program calls a Toolbox routine by using the processor's emulator trap mechanism.	• A Windows program calls a Windows function by mapping the appropriate dynamic link library into its virtual address space and then jumping to the function as an ordinary subroutine.

Resources

• The Macintosh Toolbox has hundreds of built-in resource types.	• Win32 has only about a dozen built-in resource types.
• A Macintosh file consists of two separate parts, a resource fork and a data fork.	• A Win32 file has no resource fork; resources reside in the main body of the file.
• The Macintosh Toolbox searches for a requested resource in a chain of open resource files,	• Win32 searches for a requested resource only in a single designated file, called a resource

beginning with the current resource file.	module. There is no concept of a current resource file.
• A Macintosh resource type is identified by a four-character name.	• A Win32 resource type is identified by either a character-string name or an integer ID number.
• On the Macintosh, finding a resource in its resource file and loading it into memory are combined into a single operation.	• In Win32, finding a resource in its resource module and loading it into memory are two separate operations, though often they are both performed by a single Windows function.
• On the Macintosh, after creating a resource interactively with the resource editor ResEdit, you can convert it into an equivalent text description with the resource decompiler DeRez.	• In Win32, the Visual C++ interactive resource editors automatically generate a resource description in text form, ready to be compiled with the resource compiler.