Chapter 3

The Windows Programming Model

The fundamental unit of Windows programming is—you guessed it—the window. Windows are even more central to Windows programming than they are to the Macintosh. Everything that happens in a Windows program—not only interactions with the user, but all communications with the system as well—takes place under the control of a window. Just about everything the Windows user sees on the screen is actually a window. Even things that Macintosh programmers aren't accustomed to thinking of as windows, such as icons and menus and pushbuttons and scroll bars, turn out to be just different kinds of window. If you want to understand Windows, you have to begin by understanding windows.

A window communicates with the world by receiving and sending *messages.* Windows messages are much like Macintosh events, in that they report occurrences or circumstances that the window may wish to respond to. In fact, some Windows messages correspond more or less exactly to equivalent Macintosh events: mouse clicks, keystrokes, window activations and updates, and the like. However, as we'll see, Windows also uses the message mechanism to signal higher-level conditions that a Macintosh program could obtain only through additional calls to the Toolbox. The idea of sending a message to a window is the central unifying concept on which the entire Windows programming model is based.

Each window has a *window procedure* to handle the messages it receives from the system. Every message sent to a window ultimately finds its way to the window procedure for processing. By defining how a window responds to messages, the window procedure determines everything about the window's appearance and behavior on the screen. A category of windows sharing the same window procedure is known as a *window class.* Every window must belong to some class or other, specified as a parameter when the window is created. A typical program defines at least one class, for its main window, and may define additional classes for other kinds of window that it uses on the screen.

One of the neat things about Windows programming is that there's already a complete window procedure built into the system and ready to use. This predefined window procedure, named **`DefWindowProc`** ("default window procedure"), provides a standard response for each type of message your windows can receive. This saves you from having to write your own code for every one of the hundreds of possible messages the system can throw at you: you can just "cherry-pick" those messages you care about and pass the rest on to the default window procedure, knowing that it will do something sensible with them.

Using the default window procedure gives Windows programming something of an object-oriented flavor, even if you're not actually using an object-oriented application framework like the Microsoft Foundation Classes. In effect, the default window procedure defines a universal superclass that determines the baseline appearance and behavior for all windows. Writing your own window procedure is like creating a subclass: your windows inherit all of the standard behavior except for those messages that you explicitly override with code of your own.

Each message is represented by a *message structure* of type **MSG**, shown in Listing 3–1. If the contents of this structure look familiar to you, it's because they're essentially the same as the fields of the Macintosh event record: the type of message, the time it was posted, the coordinates of the mouse at the time, and some additional parameter information that varies depending on the message type. Notice, though, that the Windows message also has to identify the window to which it is directed; this isn't necessary on the Macintosh, where all events are addressed directly to the program itself. Notice also that the Windows message contains *two* type-dependent parameters instead of only one. Originally, one of these was a 16-bit word (**wParam**) and the other a 32-bit long word (**lParam**). In the Win32 interface, however, the data types **WPARAM** and **LPARAM** are both defined as equivalent to **LONG**, so both parameters are actually 32 bits in length.

Listing 3–1. Message structure

```
typedef struct tagMSG
   {
      HWND    hwnd;                         // Handle to window receiving message
      UINT    message;                      // Message identifier
      WPARAM  wParam;                       // Type-dependent information
      LPARAM  lParam;                       // Type-dependent information
      DWORD   time;                         // Time message was posted
      POINT   pt;                           // Mouse position at time of message

   }  MSG;
```

When it comes to sheer variety of messages, Windows runs rings around the Macintosh Toolbox. The Macintosh makes do with only sixteen event types, mainly representing low-level user actions (mouse-down, mouse-up, key-down, key-up, auto-key, disk-inserted) and very basic window manipulations (activate, update, suspend, resume). Even the new-fangled Apple events (which pertain only to communicating with other programs anyway) are subsumed under one single event type, "high-level events." Windows, on the other hand, defines literally hundreds of message types—393 at last count, with more being added all the time. Luckily for Windows programmers, it isn't necessary for every program to provide an explicit response to all 393 messages. As we'll see, Windows has a built-in window procedure that already provides a reasonable way of dealing with most of them; the program needn't step in unless it wants to handle something in an unusual or nonstandard way.

Each message type is identified by its own unique *message identifier* in the **message** member of the **MSG** structure. The Win32 header files define constant names for each of the possible message identifiers, such as **WM_CREATE**, **WM_MOVE**, **WM_KEYDOWN**, and **WM_MENUSELECT**. (Many of the most common message identifiers begin with the prefix **WM_**, for "window message"; there are also other categories, such as **BM_** for "button message," **LB_** for "list box," and so on, many of which we'll be covering in later chapters.) Table 3–1 shows a few of the more common message types and their meanings.

Table 3–1. Some typical message types

Message type	Meaning
WM_CREATE	Window being created; initialize its associated data structures
WM_DESTROY	Window being destroyed; deallocate or finalize its data structures
WM_SHOWWINDOW	Show or hide window on screen
WM_CLOSE	Close window
WM_ACTIVATE	Window being activated or deactivated
WM_SIZE	Window's size has changed
WM_LBUTTONDOWN	Left mouse button pressed
WM_CHAR	Character typed on keyboard
WM_COMMAND	Command chosen from menu
WM_QUIT	Terminate program

Windows maintains a *message queue* for messages pending delivery, similar to the event queue on the Macintosh. Each running program (and in a multithreaded program, each separate thread) has its own message queue, holding messages addressed to that program's windows. The low-level device drivers for the mouse and keyboard gather the user's input actions and post them as messages to a single, system-wide queue; the Windows system then pulls them out one at a time, figures out which window they're addressed to, and reposts them into the private message queue of the program or thread that created that window. Eventually, the program's messasge loop wll retrieve each message and dispatch it to the appropriate window procedure for processing. In general, the queue is reserved for messages reporting "raw" user input like mouse clicks and keystrokes; it also receives a few other types of message such as **WM_TIMER** (a software-controlled interval timer has expired), **WM_PAINT** (the window must redraw all or part of its contents, similar to a Macintosh update event), and **WM_QUIT** (the program is about to terminate). Table 3–2 summarizes the message types that are posted to the queue. We'll be talking about a few of them in this chapter; the ones relating to mouse and keyboard input will be covered in detail in Chapter 5.

Table 3–2. Queued messages

Message type	Meaning
WM_LBUTTONDOWN	Left mouse button pressed in client area (like Macintosh content region)
WM_LBUTTONUP	Left mouse button released in client area
WM_LBUTTONDBLCLK	Left mouse button double-clicked in client area
WM_MBUTTONDOWN	Middle mouse button pressed in client area
WM_MBUTTONUP	Middle mouse button released in client area
WM_MBUTTONDBLCLK	Middle mouse button double-clicked in client area
WM_RBUTTONDOWN	Right mouse button pressed in client area
WM_RBUTTONUP	Right mouse button released in client area
WM_RBUTTONDBLCLK	Right mouse button double-clicked in client area
WM_NCLBUTTONDOWN	Left mouse button pressed in nonclient area (like Macintosh window frame)
WM_NCLBUTTONUP	Left mouse button released in nonclient area
WM_NCLBUTTONDBLCLK	Left mouse button double-clicked in nonclient area
WM_NCMBUTTONDOWN	Middle mouse button pressed in nonclient area
WM_NCMBUTTONUP	Middle mouse button released in nonclient area
WM_NCMBUTTONDBLCLK	Middle mouse button double-clicked in nonclient area
WM_NCRBUTTONDOWN	Right mouse button pressed in nonclient area
WM_NCRBUTTONUP	Right mouse button released in nonclient area
WM_NCRBUTTONDBLCLK	Right mouse button double-clicked in nonclient area
WM_MOUSEMOVE	Mouse position changed within client area
WM_NCMOUSEMOVE	Mouse position changed within nonclient area
WM_KEYDOWN	Key pressed on keyboard (no Alt key)
WM_KEYUP	Key released on keyboard (no Alt key)
WM_SYSKEYDOWN	Key pressed on keyboard in combination with Alt key
WM_SYSKEYUP	Key released on keyboard in combination with Alt key
WM_CHAR	Character typed on keyboard (no Alt key)
WM_DEADCHAR	Dead character typed on keyboard (no Alt key)
WM_SYSCHAR	Character typed on keyboard in combination with Alt key
WM_SYSDEADCHAR	Dead character typed on keyboard in combination with Alt key
WM_TIMER	Software-controlled interval timer expired

WM_PAINT	Redraw all or part of client area
WM_QUIT	Terminate program

Unlike Macintosh events, however, not all Windows messages actually go through the queue. Many of the messages a window receives are *unqueued* messages: the system sends them directly to the relevant window procedure instead of posting them to the queue to be retrieved by the program's message loop. Whereas queued messages generally correspond to the same kinds of low-level user input as Macintosh events, unqueued messages operate at a higher level of discourse, reporting conditions and occurrences that have been generated or interpreted within the Windows system itself. Often, the system's response to one message will trigger a whole series of further messages sent directly to the window procedure. These high-level, unqueued messages are what gives Windows programming much of its ease and power.

Consider what happens when a Macintosh program receives a mouse-down event from the Toolbox Event Manager. The decision logic needed to interpret and respond to such an event typically involves an elaborate sequence of Toolbox calls designed to narrow down the location and meaning of the event and provide the appropriate user feedback on the screen. When the user presses and drags the mouse in a window's size box, for example, it touches off a sequence of operations more or less like the following:

1. Your main event loop receives the event from **`WaitNextEvent`**.
2. You examine the event code in the **`what`** field of the event record and determine that it is a mouse-down event.
3. You pass the event to the Toolbox routine **`FindWindow`** to determine where on the screen the mouse was pressed, and learn that it was in the size box of the active window.
4. You call the Toolbox routine **`GrowWindow`** to track the mouse for as long as the user continues to hold down the button, providing visual feedback in the form of a dotted outline whose bottom-right corner follows the mouse's movements.
5. When the button is released, **`GrowWindow`** returns to you with the new dimensions of the window. You then pass this information to the Toolbox routine **`SizeWindow`** to resize the window to its new dimensions.
6. **`SizeWindow`** automatically redraws the window's frame in its new size, as well as any portions of other windows' frames that have become visible as a result of the resizing operation. It also generates update events for those portions of any window's content region that need to be redrawn.
7. On a later pass of your event loop, you will receive the update event(s) and will redraw the contents of the window(s) as needed.

Now let’s look at how Windows would handle this same event. Although the Windows mouse can have as many as three buttons, the usual convention is to use the left button for manipulating windows on the screen; so the Windows equivalent to the user’s pressing the mouse button in a Macintosh window’s size box would be to press the left button in the window’s sizing border. The sizing border is part of the window’s nonclient area—the part of the window that the system draws for you, corresponding to the window frame in Macintosh terminology—so this will generate a **WM_NCLBUTTONDOWN** message (**NC** for “nonclient,” **L** for “left”) analogous to the Macintosh mouse-down event. Your message loop will retrieve this message from the queue with the Windows function **GetMessage** (like **WaitNextEvent** on the Macintosh) and hand it off to another Windows function, **DispatchMessage**; the latter, in turn, will relay it back to your program’s window procedure.

At this point, your window procedure could (theoretically, anyway) choose to process the **WM_NCLBUTTONDOWN** message “by hand,” with a sequence of operations similar to the ones you’d need to perform on the Macintosh: hit-testing the mouse location, tracking the mouse as it drags the window’s sizing border, resizing the window, and so on. But it’s much easier to skip all that and just pass the message on to the default window procedure, **DefWindowProc**. That’s what the default window procedure is there for: to provide a standard response to each message you receive, saving you the trouble of having to write one for yourself.

So let’s assume you pass that mouse message (**WM_NCLBUTTONDOWN**) to the default window procedure. Here’s where things start to get interesting. The first thing **DefWindowProc** has to do is figure out what part of the window’s nonclient area received the mouse click, so it sends the window the message **WM_NCHITTEST**, with the mouse’s screen coordinates as a parameter. Again, you could intercept this message in your window procedure and process it yourself, but you’ll usually just pass it through to the default window procedure to perform the hit-test in the standard way.

When the result of the **WM_NCHITTEST m**essage indicates that the mouse press was in the window’s sizing border, the default procedure will track the mouse by processing **WM_NCMOUSEMOVE** messages until it receives a **WM_NCLBUTTONUP**. Each time it receives a **WM_NCMOUSEMOVE**, it calls the Windows function **MoveWindow** to provide visual feedback to the user by redrawing the window in a new size. **MoveWindow** redraws the window’s nonclient area (frame) and then sends another series of messages back to the window:

- **WM_MOVE** to tell it the new coordinates of its top-left corner
- **WM_SIZE** to tell it its new width and height
- **WM_NCCALCSIZE** to ask it to recalculate the coordinates of its client area (the Windows equivalent of the Macintosh content region)
- **WM_PAINT** to tell it to redraw the contents of its client area

There’s also a pair of “hook” messages to allow the window procedure to get control before and after redrawing, in case it wants to customize the operation in some way:

- **WM_WINDOWPOSCHANGING** to tell it its size or position is about to change
- **WM_WINDOWPOSCHANGED** to tell it its size or position *has* just changed

(If the window procedure intercepts the **WM_WINDOWPOSCHANGED** message, the later **WM_MOVE** and **WM_SIZE** messages are suppressed, since by that time the window has already had a chance to adjust to its new position and size.)

If all this proliferation of messages strikes you as rather complex, you're right. Keep in mind, though, that you're not required to deal with all of them yourself, because

the default window procedure is always there to back you up. In fact, most of these messages are really *intended* to be handled by the default window procedure, beneath your program's level of notice. The only reason for sending the messages is to give you the *option,* at various points in the process, of stepping in and taking control for yourself. The combination of the message mechanism with the default window procedure gives you the freedom to sit back and let the Windows system handle everything for you, together with the flexibility to redefine or customize its behavior if you choose to do so.

One of the common criticisms of the Macintosh Toolbox, right from the beginning, has been the amount of coding effort needed to get even the simplest application running. The Toolbox provides all the support needed to put windows on the screen, move them, size them, scroll them, and so on; but there's still a considerable burden on the program (and the programmer) to issue the right sequence of Toolbox calls to gather the user's mouse clicks, interpret them, and carry them out. By contrast, Windows' high-level message structure and built-in default window procedure make it comparatively easy to get a simple application running. Listing 3–2, for example, shows the complete code for a "null" application that just puts a window on the screen and lets the user manipulate it with the mouse. The window has all the expected paraphernalia that we'll be discussing in Chapter 4—a title bar for dragging it around the screen, a sizing border, maximize and minimize boxes, a system menu—and they all work just the way the user expects them to. If you throw out all the comments, white space, and pretty-printing (not to mention my somewhat verbose coding style), the entire program comes to only about 50 lines of "live" code. An equivalent Macintosh program (including support for desk accessories, which don't exist in Windows) would have to be at least five times that long.

Main Function

The main entry point to our NullApp program is named **`WinMain`**. Every Windows program must begin with a function of this name, which takes four parameters:

- A handle to the current instance of the program, the one just being started up. (Recall from Chapter 2 that Windows allows more than one instance of a program to run at the same time.)
- If any other instances of the same program are already running, a handle to the previous instance most recently started before this one. If the current instance is the only one now running, this parameter is **`NULL`**.
- A pointer to a null-terminated character string containing any parameters being passed to the program via its command line. This parameter is a relic of the nasty old DOS command-line interface; in Windows, programs are normally started with a double click or a menu command from the Program Manager or File Manager, rather than by typing a command line. If the user starts the program by double-clicking one of its document files, the system will pass the document's path name in this parameter.

Listing 3–2. A null Windows application

```
//
//                                    NullApp
//                        Null Windows application program
//                      S. Chernicoff        15 January 1995
//

//         Simple program to display and manipulate a window on the screen


   #include <windows.h>


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


// Global variables


   HWND  TheWindow;                                      // Handle to main window
   BOOL  ContinueFlag = TRUE;                               // Keep running?

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


// Function prototypes


INT CALLBACK WinMain (HINSTANCE thisInstance, HINSTANCE prevInstance, CHAR *commandLine, INT showState);
      // Main function
   VOID PASCAL Initialize (HANDLE thisInstance, HANDLE prevInstance, CHAR *commandLine, INT showState);
         // Do one-time-only initialization.
      VOID InitClass (HANDLE thisInstance, HANDLE prevInstance);
            // Register the window class.
      VOID InitWindow (HANDLE thisInstance, INT showState);
            // Create the window.
   VOID MainLoop (VOID);
         // Execute one pass of main program loop.
   VOID Finalize (VOID);
         // Do one-time-only finalization.

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

Listing 3–2. A null Windows application (*continued*)

```
INT CALLBACK WinMain (HINSTANCE thisInstance, HINSTANCE prevInstance, CHAR *commandLine, INT showState)

   // Main function

   {

      Initialize (thisInstance, prevInstance, commandLine, showState);   // Do one-time-only initialization


      do
         MainLoop ();                                    // Execute one pass of main loop
      while ( ContinueFlag );                               // Continue until time to quit


      Finalize ();                                       // Do one-time-only finalization


      return NO_ERROR;                                   // Signal successful completion


   }  /* end WinMain */

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

VOID PASCAL Initialize (HANDLE thisInstance, HANDLE prevInstance, CHAR *commandLine, INT showState)

   // Do one-time-only initialization.

   {
      InitClass  (thisInstance, prevInstance);  // Register the window class
      InitWindow (thisInstance, showState);     // Create the window

   }  /* end Initialize */

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

Listing 3–2. A null Windows application (*continued*)

```
VOID InitClass (HANDLE thisInstance, HANDLE prevInstance)

   // Register the window class.

   {
      WNDCLASS  windowClass;                                 // Window class
      HICON     progIcon;                                    // Program's screen icon
      HCURSOR   arrowCursor;                                 // Default cursor
      HBRUSH    bkBrush;                                     // Brush for painting window background


      if ( prevInstance == NULL )                            // Is this the first instance of program?
         {
         windowClass.lpszClassName = "NullApp";                 // Use program name as class name

         windowClass.lpfnWndProc = DefWindowProc;               // Use default window procedure
         windowClass.hInstance   = thisInstance;                // Current instance is the owner

         windowClass.style = NULL;                              // No special style

         windowClass.lpszMenuName = NULL;                       // No menu

         progIcon = LoadIcon(NULL, IDI_APPLICATION);            // Load generic icon
         windowClass.hIcon = progIcon;                          // Set icon

         arrowCursor = LoadCursor(NULL, IDC_ARROW);             // Load stock arrow cursor
         windowClass.hCursor = arrowCursor;                     // Set cursor

         bkBrush = HBRUSH(COLOR_WINDOW + 1);                    // Create brush for system background color
         windowClass.hbrBackground = bkBrush;                // Set window background brush

         windowClass.cbClsExtra = 0;                         // No extra class data
         windowClass.cbWndExtra = 0;                         // No extra window data

         RegisterClass (&windowClass);                          // Register the class

         }  /* end if ( PrevInstance == NULL ) */

   }  /* end InitClass */

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

Listing 3–2. A null Windows application (*continued*)

```
VOID InitWindow (HANDLE thisInstance, INT showState)

   // Create the window.

   {
      TheWindow = CreateWindow ("NullApp",            // Use program name for class name
                               "This space left blank",  // Set window title
                               WS_OVERLAPPEDWINDOW,   // Use standard window style
                               CW_USEDEFAULT,         // Let Windows choose initial x
                               CW_USEDEFAULT,         //   and y position
                               CW_USEDEFAULT,         // Let Windows choose initial width
                               CW_USEDEFAULT,         //   and height
                               NULL,                  // No parent window
                               NULL,                  // Use menu from window class
                               thisInstance,          // Use current program instance
                               NULL);                 // No special creation parameters

      ShowWindow   (TheWindow, showState);            // Display window on screen
      UpdateWindow (TheWindow);                          // Force update of client area

   }  /* end InitWindow */

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

VOID MainLoop (VOID)

   // Execute one pass of main program loop.

   {
      MSG   theMessage;                               // Next message to process


      ContinueFlag = GetMessage(&theMessage, NULL, 0, 0);  // Get next message

      TranslateMessage (&theMessage);                 // Convert virtual keys to characters
      DispatchMessage  (&theMessage);                 // Send message to window procedure

   }  /* end MainLoop */

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

VOID Finalize (VOID)

   // Do one-time-only finalization.

   {
      DestroyWindow (TheWindow);                      // Destroy the window

   }  /* end Finalize */
```

- An integer code specifying how the program is to display its main window at startup. The value of this parameter is normally one of two constants defined in the Windows API header files: either **SW_SHOWNORMAL**, meaning to display the window in the normal way, or **SW_SHOWMINNOACTIVE**, meaning to show the window initially in its minimized state (as a button in the Windows 95 task bar, or as an icon on the desktop in earlier versions of Windows). After creating its main window, the program typically just passes the value of this parameter along to the Windows function **ShowWindow**, which will display the window in the requested state.

Notice in Listing 3–2 that the **WinMain** function is declared with the keyword **CALLBACK**. This keyword is defined in the Windows API header files and is required for any function (such as a main entry point or a window procedure) that will be called directly by the Windows system, rather than from within the program itself.

Looking at the logical structure of our **WinMain** function, you'll see that it's no different from that of a typical Macintosh program. In fact, if you compare it with the main function of my old MiniEdit program in Appendix A, you'll find that it's virtually identical. Just as you would expect, the program begins by doing some one-time initialization, then enters a main loop that it executes repeatedly under the control of a boolean flag variable. In a Macintosh program, each pass of the main loop processes one event; in a Windows program, each pass processes one message. When something happens to change the value of the control flag, the program falls out of its main loop, does any needed finalization, and exits back to the system shell.

One minor difference you may notice between the Macintosh and Windows versions of the program is the polarity of the control flag. The Macintosh version uses a global boolean flag named **Finished**, which is initialized to **false** and becomes **true** when the user chooses the **Quit** command from the menu; this causes the program to fall out of its main loop and terminate. In the Windows version, the name of this flag is changed to **ContinueFlag** and its polarity is reversed: it is initially **TRUE** and becomes **FALSE** to signal termination. This change of polarity is just a minor programming convenience resulting from the way the Windows system signals when it's time for the program to quit, as we'll see later in this chapter.

Registering a Window Class

One of the first orders of business for any Windows program is to *register* its window classes with the system so that it can use them to create its windows. The Windows function **RegisterClass** accepts a pointer to a data structure of type **WNDCLASS**, which the program fills in with descriptive information about the class to be registered. Listing 3–3 shows the contents of this structure, which include the following items:

- The name of the class.
- A pointer to its window procedure.
- The program instance to which it belongs.

- A word of bit flags specifying various aspects of the way its windows appear or behave.
- An icon for representing its windows on the screen when in their minimized state.
- A default cursor shape to use when the cursor is moved into its windows.
- A brush (like a QuickDraw pattern) for painting the background area inside its windows.
- The resource name or ID number of its top-level menu, which defines the contents of its windows' menu bars (actually more like a Macintosh **'MBAR'** than a **'MENU'** resource).
- The number of bytes of extra, private data the program wishes to allocate in the window class structure itself and in each window structure created from it. (We'll learn how to access this extra data in Chapter 4.)

Listing 3–3. Contents of `WNDCLASS` structure

```
typedef struct _WNDCLASS
   {
     UINT     style;             // Style options
     WNDPROC  lpfnWndProc;       // Pointer to window procedure
     int      cbClsExtra;           // Extra bytes of client data in class structure
     int      cbWndExtra;           // Extra bytes of client data in window structure
     HANDLE   hInstance;         // Program instance registering the class
     HICON    hIcon;             // Icon for minimizing windows
     HCURSOR  hCursor;              // Default cursor
     HBRUSH   hbrBackground;        // Brush for painting window background
     LPCTSTR  lpszMenuName;      // Resource name of menu
     LPCTSTR  lpszClassName;        // Name of class

   }  WNDCLASS;
```

> In earlier versions of the Windows system, multiple instances of the same program weren't allowed to reregister the same window class more than once. The normal way to prevent this was to test the previous-instance handle that the program's main function received as a parameter, and register the class only if the previous instance was **`NULL`** (that is, if no other instance of the program already existed). Although Win32 has removed this restriction, there's no harm in observing it anyway for backward compatibility, as illustrated by our NullApp program's **`InitClass`** function in Listing 3–2.

Every window class has to have a unique name, different from that of any other class in the system. To ensure uniqueness, the common convention is to use the name of your program as the name of its main window class. You can see an example of this in the **`InitClass`** function of Listing 3–2, where the name of the class is set with the statement

```
windowClass.lpszClassName = "NullApp";         // Use program name as class name
```

For the class's window procedure, NullApp simply uses the system's built-in window procedure, **DefWindowProc**, without modification:

```
windowClass.lpfnWndProc = DefWindowProc;        // Use default window procedure
```

This means, of course, that the program's windows can't do anything special beyond the standard behavior that's built into the system by default. This allows the user to use all the standard components of the window (the title bar, sizing border, and so forth) without displaying anything in the window or responding to user input in any way. We'll see later how to write our own window procedure to extend or override the standard behavior.

Table 3–3. Window class style options

Style name	Meaning
CS_NOCLOSE	Omit close box and system-menu Close command
CS_DBLCLKS	Receive double-click messages
CS_HREDRAW	Redraw window when width changes
CS_VREDRAW	Redraw window when height changes
CS_BYTEALIGNCLIENT	Align client area on horizontal byte boundaries for faster drawing
CS_BYTEALIGNWINDOW	Align nonclient area on horizontal byte boundaries for faster positioning
CS_SAVEBITS	Save obscured portion of screen for later restoration
CS_OWNDC	Allocate a private device context for each window of class
CS_PARENTDC	Inherit parent window's device context
CS_GLOBALCLASS	Make class globally accessible to all programs

The s**tyle** member of the **WNDCLASS** structure typifies a common technique that we'll find repeated again and again throughout the Windows programming interface: a word of bit flags that define or modify the way an element of the interface works. Table 3–3 shows the style options that are available for window classes. Each of the names in the table, such as **CS_NOCLOSE**, is defined in the Win32 header files as a constant bit mask corresponding to a single bit in the **WNDCLASS** structure's 16-bit **style** word. You can combine these masks with the C language's bitwise logical operators (**&**, **|**, **^**, and **~**) to form any combination of style bits you need. For instance, although our NullApp program doesn't request any of these options (it just sets **windowClass.style** to **NULL**), we'll see later that WiniEdit uses the statement

```
windowClass.style = CS_HREDRAW | CS_VREDRAW;
```

to request that its windows be redrawn whenever their size changes in either the horizontal or vertical dimension. Other options allow you to specify whether the window has a close box, whether it receives special messages for double mouse clicks, whether it has its own private device context (the Windows equivalent of a QuickDraw graphics port) or inherits one from its parent, and so on. Some of these options are rather obscure, but they're available in case you need them.

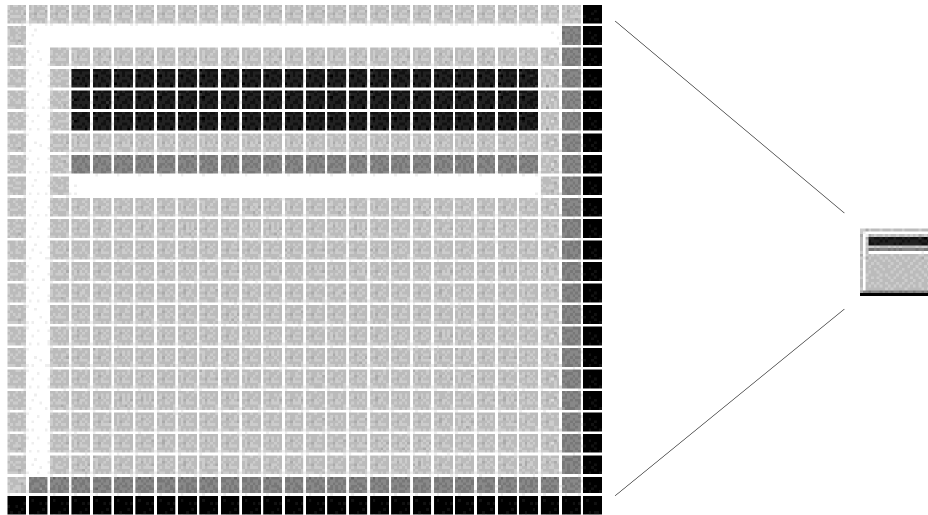
Every window class has three graphical items associated with it: an icon, a cursor,

and a background brush. The icon defines how windows belonging to the class appear when minimized. In older versions of Windows, it would be used to represent the window directly on the screen, like a file or folder on the desktop in the

Macintosh Finder; in Windows 95, it is displayed in reduced form as part of the button representing the window in the task bar. NullApp just uses the system's standard, "generic" application icon (Figure 3–1). This icon is available as a system resource under the name **IDI_APPLICATION**:

```
progIcon = LoadIcon(NULL, IDI_APPLICATION);       // Load generic icon
windowClass.hIcon = progIcon;                     // Set icon
```

Figure 3–1. Generic application icon



(The prefix **IDI_** stands for "identifier of an icon.") WiniEdit, on the other hand, provides an icon of its own, created and placed in the program's executable file with the Visual C++ onscreen icon editor. The icon's resource ID is defined in the program's resource header file, **WiniEdit Resources.h**, as a constant named **ProgIcon_ID**. The following statements load the resource and associate it with the program's window class:

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

The cursor associated with a window class tells the Windows system what cursor shape to use by default when the user moves the mouse into one of the class's windows. (We'll learn in Chapter 5 how a program can use messages from the system to do its own cursor management.) Both NullApp and WiniEdit use the "stock" arrow cursor, retrieved as a system resource:

```
arrowCursor = LoadCursor(NULL, IDC_ARROW);        // Load stock arrow cursor
windowClass.hCursor = arrowCursor;                // Set cursor
```

A *brush* is the Windows equivalent of a Macintosh pattern: a small pixel image that can be repeated indefinitely to fill an area, like identical floor tiles laid end to end. We'll save our discussion of brushes for Chapter 6, when we talk about the Windows Graphical Device Interface; all we need to know for now is that each window class includes a *background brush* for filling in the background of a window's client area. Just as with icons and cursors, the system defines a set of stock brushes as system

resources, in solid black, white, and various shades of gray, as well as a transparent or “hollow” brush.

There’s also a set of 21 brushes corresponding to *system colors* that the user is free to set with the Color control panel. These represent the user’s preferences for the colors of various user interface elements, such as **COLOR_MENU** for the background color of menus, **COLOR_MENUTEXT** for the text of menu items, **COLOR_SCROLLBAR** for the shafts of scroll bars, and so forth. In particular, the system color **COLOR_WINDOW** is the color the user has chosen for painting the background area of a window. You can get a brush for any of the system colors by typecasting the color’s ID number directly into a brush handle (type **HBRUSH**). However, for reasons unknown to mere mortals, the Windows gods have decreed that the number of the brush is *one more than* the number of the corresponding system color. Thus both NullApp and WiniEdit use the following statements to set the background brush for their window classes:

```
bkBrush = HBRUSH(COLOR_WINDOW + 1);    // Create brush for system background color
windowClass.hbrBackground = bkBrush;// Set window background brush
```

NullApp’s windows don’t need any menus, since all they do is let themselves be dragged around or resized on the screen. So the code specifying the menu for the program’s window class is simply

```
windowClass.lpszMenuName = NULL;                    // No menu
```

In WiniEdit’s case, the contents of the menu bar are defined as a resource in the program’s executable file under the resource ID **Main_Menu**. Because the **WNDCLASS** structure expects a string defining the name of the menu resource, the integer resource ID must be converted to the proper form with the **MAKEINTRESOURCE** macro that we discussed in Chapter 2:

```
resourceID = MAKEINTRESOURCE(Main_Menu);            // Convert resource ID
windowClass.lpszMenuName = resourceID;              // Set menu
```

Creating and Displaying a Window

Once you’ve registered your window class, the next step is to use it to create your main window and display it on the screen. The Windows functions for doing this are **CreateWindow** and **ShowWindow**. We’ll just look at them briefly here and save the details for our discussion of windows in Chapter 4.

NullApp uses the following statement to create its window:

```
TheWindow = CreateWindow ("NullApp",         // Use program name for class name
                         "This space left blank",     // Set window title
                         WS_OVERLAPPEDWINDOW,// Use standard window style
                         CW_USEDEFAULT,      // Let Windows choose initial x
                         CW_USEDEFAULT,      //   and y position
                         CW_USEDEFAULT,      // Let Windows choose initial width
                         CW_USEDEFAULT,      //   and height
                         NULL,               // No parent window
                         NULL,               // Use menu from window class
                         thisInstance,       // Use current program instance
                         NULL);              // No special creation parameters
```

Notice that the window's class is identified by name, rather than directly by means of a handle to the class object itself. This is why every window class must have a name that is unique in the entire system. As we've already mentioned, the usual convention is to use the name of the program itself (in this case, **"NullApp"**) as the name of its main window class. The **CreateWindow** function creates a window of the specified class and returns a window handle of type **HWND**.

Recall that one of the parameters your **WinMain** function receives from the system when you first start up is an integer code specifying how to display your main window initially on the screen. This value is known as a *show state,* and is passed as a parameter to the Windows function **ShowWindow** to tell it how to display the window. Normally, the show state you receive from the system will be the constant **SW_SHOWNORMAL**, meaning to open your window in its normal, fully displayed state. However, if the user has held down the Shift key while starting up the program, or has checked the **Run Minimized** option to the Program Manager's or File Manager's **Run...** command, you'll get a show-state parameter of **SW_SHOWMINNOACTIVE** instead, telling you to display the window in a minimized, inactive state. You don't have to inspect this parameter for yourself, though—just pass it along to **ShowWindow** after creating your main window, as our NullApp program does with the statement

```
ShowWindow (TheWindow, showState);               // Display window on screen
```

in its **InitWindow** routine, and Windows will do the right thing. After displaying the window, you should also call the Windows function **UpdateWindow**

```
UpdateWindow (TheWindow);                        // Force update of client area
```

to make sure its initial contents get drawn properly. As we'll see in Chapter 4, the result is to generate a **WM_PAINT** message to your window procedure, telling it to draw the contents of the client area—just like a Macintosh update event.

Main Loop

After doing its preliminary initialization, our program's main function, **WinMain**, enters its message loop, in which it repeatedly calls the routine **MainLoop** under the control of a global flag, **ContinueFlag**. As long as this flag remains **TRUE**, the program will continue iterating the loop; when the flag becomes **FALSE**, the loop will terminate and control will fall through into the **Finalize** routine.

Our **MainLoop** routine issues calls to three Windows functions: **GetMessage**, **TranslateMessage**, and **DispatchMessage**. **GetMessage** is the basic message-retrieval function, analogous to **GetNextEvent** or **WaitNextEvent** on the Macintosh. It accepts a pointer to a message structure as its first parameter, fills it in with descriptive information about the message, and removes the message from the queue. The remaining parameters allow you to place restrictions on the kind of message you're requesting. The second parameter is a window handle, limiting the request to messges addressed specifically to that window or one of its children. A **NULL** handle, as in the current example, denotes any window belonging to the current thread (the one issuing the **GetMessage** call). The last two parameters narrow the request to a limited range of (minimum and maximum) message types; they thus serve a purpose similar to (though less flexible than) the event mask parameter to **GetNextEvent** or **WaitNextEvent** on the Macintosh. For example, you can limit your request to mouse messages only by setting these two parameters to the constants **WM_MOUSEFIRST** and **WM_MOUSELAST**, or to keyboard messages by using **WM_KEYFIRST** and **WM_KEYLAST**. Setting both parameters to **0** asks for the next pending message in the queue, regardless of type.

The **GetMessage** function returns a boolean result that's always **TRUE** unless the message being returned is **WM_QUIT**. You can use this boolean value to control the termination of your main message loop, as NullApp does by setting the global **ContinueFlag**. How does Windows know when to send you a **WM_QUIT** message? You tell it when, by calling the Windows function **PostQuitMessage**. (Our WiniEdit program, for instance, calls this function in response to the user's choosing the **Exit** menu command.) The **WM_QUIT** message gives your window procedure a chance to do any last-minute cleanup it might need before falling out of the message loop.

> You may be wondering what happens if you ask for a message and there aren't any waiting in the queue. Windows has no equivalent to the Macintosh concept of a null event. Instead, the **GetMessage** function simply suspends your thread's execution until a message arrives, allowing other processes or threads to run in your place. When your call to **GetMessage** does eventually return, it's guaranteed to come back with a bona-fide message for you to respond to.
>
> There's also an alternative retrieval function, **PeekMessage**, which always returns immediately, with or without a message to report. Like **GetMessage**, **PeekMessage** returns a boolean result; but whereas **GetMessage** uses its result to signal the arrival of a **WM_QUIT** message, **PeekMessage** uses it to report whether there was a message in the queue to retrieve. If there is a message, **PeekMessage** copies it to the message structure you supply as a parameter and returns a **TRUE** result; if the queue is empty, it returns **FALSE** and leaves the message structure untouched. **PeekMessage** also takes an additional parameter telling it whether to remove any message it returns or leave it in the queue for later processing; the latter option makes it analogous to the Macintosh **EventAvail** function.
>
> Finally, if **PeekMessage** reports that the queue is empty, you can use the Windows function **WaitMessage** to suspend execution of your thread pending

the arrival of a message, similarly to what **`GetMessage`** does when it finds the queue empty.

After receiving a message from **GetMessage**, the next step is to pass it to another Windows function, **TranslateMessage**. This is a rather specialized function that maps raw keystroke messages such as **WM_KEYDOWN** and **WM_SYSKEYDOWN** into corresponding character-based messages like **WM_CHAR** and **WM_SYSCHAR**, under the control of the currently installed keyboard layout; it has no effect on non-keyboard messages. We'll have more to say about **TranslateMessage** when we discuss keyboard input processing in Chapter 5; for now, all we need to know is that every message we receive from **GetMessage** should be passed to **TranslateMessage** before further processing.

The key step after retrieving a message from the queue is to pass it to the Windows function **DispatchMessage**. This examines the message to see which window it's addressed to, looks up the window class to which the window belongs, and relays the message to the window procedure registered for that class. What happens next is up to the window procedure, which is, of course, part of your program that you write yourself. Our NullApp program doesn't have a window procedure of its own: it simply uses the system's built-in window procedure, **DefWindowProc**, for all of its message processing. For an example of a real, live window procedure, we'll have to turn to our fully developed example program, WiniEdit.

The real heart of a Windows program is its *window procedure* (commonly known as a "winproc"), which analyzes and responds to the messages the program receives from the system. Just to review some of what we've already learned:

- Windows sends messages to a window.
- The window belongs to a window class.
- The window class has a window procedure, identified as a parameter when the class was registered.
- All messages sent to the window are passed to its class's window procedure for processing.

Theoretically, a program could register more than one window class and thus define more than one window procedure; but in practice, the great majority of Windows programs have just one window procedure that receives all messages addressed to any of their windows.

Table 3–4. Macintosh window messages

Message code	Meaning
wNew	Initialize new window
wCalcRgns	Calculate structure and content regions
wDraw	Draw window frame
wDrawGIcon	Draw size region
wGrow	Draw feedback image for resizing window
wHit	Find where mouse was pressed
wDispose	Prepare to dispose of window

One way to think about the window procedure is to compare it to the Macintosh window definition function, or **WDEF**. Both accept messages sent by the system at strategic times to allow you to customize some aspects of a window's operations. But the Macintosh version accepts only a very limited set of messages and no others

(Table 3–4). This means the Macintosh **WDEF** is only screen-deep: it lets you change the window's superficial appearance, but not its underlying behavior. On the other hand, since it's a resource, it can be incorporated into any program's application file with a tool like ResEdit, without touching the code of the program itself. In Volume Three of my *Macintosh Revealed* series, for instance, I developed a sample window definition function named SideWindow that displays a window with its title bar running vertically down the side instead of horizontally across the top (Figure 3–2). Simply by compiling this function into a **'WDEF'** resource and copying it into my program's application file with ResEdit, I was able to produce a version of MiniEdit that displayed its windows in this form.

Figure 3–2. A Macintosh side window

By contrast, the Windows window procedure is part of the program itself, so it can't be changed without recompiling the entire program. It can have any name the program chooses to give it (in WiniEdit, it's named **DoMessage**), but must always have the following standard function signature:

```
LONG CALLBACK DoMessage (HWND   thisWindow,  // Handle to window receiving message
                         UINT   messageCode, // Message identifier
                         WPARAM wParam,      // Type-dependent information
                         LPARAM lParam)      // Type-dependent information
```

The keyword **CALLBACK** is required because the window procedure is a *callback function:* one that's designed to be called by the Windows system, rather than from within the program itself. Notice that the procedure's four parameters are the same as the first four fields of the message structure, which we looked at earlier (Listing 3–1). The parameters **wParam** and **lParam**, you'll recall, contain message-dependent information that varies from one type of message to another. (The names are a holdover from earlier versions of the Windows system, in which one of the parameters was a 16-bit word and the other a 32-bit long word; in Win32, both the

WPARAM and **LPARAM** data types are equated to **LONG**, so both parameters are

actually 32 bits.) The procedure returns a result of type **LONG**, whose significance is also message-dependent. The meanings of the parameters and result for any particular message type are given in the *Win32 Programmer's Reference.*

Listing 3–4 shows WiniEdit's window procedure. It's essentially just a big **switch** statement that examines the message type and calls the appropriate routine within the program to respond to it. The **switch** has a branch for each type of message that the program chooses to deal with explicitly. In WiniEdit's case, there are only ten of these: **WM_CREATE**, **WM_SIZE**, **WM_CTLCOLOREDIT**, **WM_SYSCOLORCHANGE**, **WM_SETFOCUS**, **WM_INITMENUPOPUP**, **WM_COMMAND**, **WM_QUERYENDSESSION**, **WM_CLOSE**, and **WM_DESTROY**. We'll be discussing a few of these in the rest of this chapter and the others later in the book. All other messages fall through into the **switch** statement's **default** branch, which simply relays them to the system's default window procedure, **DefWindowProc**, to be dealt with in the standard way. Since many of the messages directed to a window are really designed to be handled by Windows itself, this last step is crucial to allow the system to receive these messages.

Listing 3–4. `WiniEdit`'s window procedure

```
LONG CALLBACK DoMessage (HWND thisWindow, UINT msgCode, WPARAM wParam, LPARAM lParam)

   // Get and process one message.

   {
      LONG  result = 0;                                          // Function result

      ErrorFlag = FALSE;                                      // Clear I/O error flag

      switch ( msgCode )                                      // Dispatch on message code
         {
            case WM_CREATE:
               DoCreate (thisWindow, wParam, lParam);          // Handle WM_CREATE message
               break;

            case WM_SIZE:
               DoSize (thisWindow, wParam, lParam);         // Handle WM_SIZE message
               break;

            case WM_CTLCOLOREDIT:
               result = DoCtlColorEdit (thisWindow, wParam, lParam);   // Handle WM_CTLCOLOREDIT message
               break;

            case WM_SYSCOLORCHANGE:
               DoSysColorChange (thisWindow, wParam, lParam); // Handle WM_SYSCOLORCHANGE message
               break;

            case WM_SETFOCUS:
               DoSetFocus (thisWindow, wParam, lParam);        // Handle WM_SETFOCUS message
               break;
```

Listing 3–4. `WiniEdit`'s window procedure (*continued*)

```
      case WM_INITMENUPOPUP:
        DoInitMenuPopup (thisWindow, wParam, lParam);  // Handle WM_INITMENUPOPUP message
        break;

      case WM_COMMAND:
        DoCommand (thisWindow, wParam, lParam);        // Handle WM_COMMAND message
        break;

      case WM_QUERYENDSESSION:
        result = DoQuery (thisWindow, wParam, lParam); // Handle WM_QUERYENDSESSION message
        break;

      case WM_CLOSE:
        DoClose ();                                   // Handle WM_CLOSE message
        break;

      case WM_DESTROY:
        DoDestroy (thisWindow, wParam, lParam);        // Handle WM_DESTROY message
        break;
      default:
        result = DefWindowProc (thisWindow, msgCode,   // Pass message to Windows
                                wParam, lParam);      //    for default processing
        break;

    }  /* end switch ( msgCode ) */


  return result;


}  /* end DoMessage */
```

Table 3–5 lists a few messages related to overall program control. We've already mentioned the most important of these, **`WM_QUIT`**, several times. Windows sends the **`WM_QUIT`** message when a program asks for it by calling the Windows function **`PostQuitMessage`**. This is the only message that causes the **`GetMessage`** function to return a **`FALSE`** result, signaling the program to fall out of its message loop and terminate.

The other two messages in the table are invoked when the user attempts to end the Windows session by exiting from Windows or shutting down the computer. Before shutting down, Windows sends a **WM_QUERYENDSESSION** message to each running program, asking the program's permission to end the session. If a program is in an inconvenient state, such as in the middle of an operation that shouldn't be interrupted, it can deny permission by returning a **FALSE** result to this message. Listing 3–5, for instance, shows how WiniEdit handles the **WM_QUERYENDSESSION** message. (Portions of the code are shown in skeleton form because they involve parts of the Windows programming interface that we haven't discussed yet.) The WiniEdit function for responding to this message, **DoQuery**, in turn calls another program function, **CloseDoc**, to close the document displayed in its document window. If the contents of the doucment are "dirty" (have been changed since being read from or written to the disk), **CloseDoc** displays a message box asking the user whether to save the document before shutting down. If the user selects the "Yes" button, the function saves the window's contents to the the disk and returns a boolean result of **TRUE**; **DoQuery** in turn relays this value back to the system, indicating permission to proceed with the system shutdown. If the user selects "No," or if the document isn't dirty, it returns the same result, but without saving the document. (In each of these cases, **CloseDoc** also takes care of some other housekeeping, closing the document file and clearing the text displayed in the window to empty.) However, if the user selects the third choice in the message box, "Cancel" (or if an error occurs in saving the document to the disk), the function returns a **FALSE** result, denying permission for the system to shut down.

Table 3–5. Messages relating to program control

Message type	Meaning
WM_QUIT	Terminate program
WM_QUERYENDSESSION	OK to end Windows session?
WM_ENDSESSION	Windows session ending

As soon as any program returns a **FALSE** response to the **WM_QUERYENDSESSION** message, Windows immediately stops sending such messages to the remaining programs and cancels the system shutdown; if, on the other hand, every running program responds **TRUE**, the system proceeds to shut down as planned. In either case, it sends each program a **WM_ENDSESSION** message with a boolean parameter indicating whether the shutdown is proceeding. If the parameter value is **TRUE**, this message constitutes a "goodbye kiss," notifying the program that the system is definitely about to shut down and giving it one last chance to do any final housekeeping it may require. If the parameter is **FALSE**, the message informs the program that the intended system shutdown, announced by the previous **WM_QUERYENDSESSION** message, has been canceled.

Listing 3–5. Handle `WM_QUERYENDSESSION` message

```
BOOL DoQuery (HWND thisWindow, UINT wParam, LONG lParam)

   // Handle WM_QUERYENDSESSION message.

   {
      BOOL  confirmed;                 // Did user confirm operation?


      confirmed = CloseDoc ();         // Allow user to save document if necessary
      return confirmed;                   // Report confirmation or cancellation

   }  /* end DoQuery */
```

Listing 3–5. Handle `WM_QUERYENDSESSION` message (*continued*)

```
BOOL CloseDoc (VOID)

   // Close document displayed in window.

   {
      BOOL  dirty;                        // Window contents changed since last save?
      INT   msgResult;                    // Result value returned by message box
      BOOL  confirmed;                    // Did user confirm operation?


      dirty = /* Has text been edited? */;
      if ( dirty )
         {
            msgResult = /* Display message box on screen */;

            switch ( msgResult )                // Dispatch on message result
               {
                  case IDYES:
                     /* Save window contents to disk */;
                     confirmed = !ErrorFlag;     // Confirm if no error
                     break;

                  case IDNO:
                     confirmed = TRUE;           // Confirm without saving
                     break;

                  case IDCANCEL:
                     confirmed = FALSE;       // Cancel operation
                     break;

               }  /* end switch ( msgResult ) */

         }  /* end if ( dirty ) */

      else
         confirmed = TRUE;                      // Confirm if not dirty


      if ( confirmed )                       // Did user confirm operation?
         {
            if ( (TheFile != NULL) )         // Is window associated with a file?
               /* Close file */

            /* Clear edit control's text */

         }  /* end if ( confirmed ) */


      return confirmed;                         // Report confirmation or cancellation


   }  /* end CloseDoc */
```

Sometimes, in the course of your program’s operations, it’s convenient to be able to send a message to a window yourself—either to one of your own windows or to one belonging to another program or process. There are two ways of doing this, depending on whether you want the message sent in queued or unqueued form. The Windows function **PostMessage** is analogous to **PostEvent** on the Macintosh; it places a specified message in a window’s message queue (more precisely, in the queue for the thread that created the window). It then returns control immediately to the point of call, without waiting for the message to be retrieved and processed by the thread’s message loop. Note that you can post any message you like this way: it doesn’t have to be one of the normal queued messages listed earlier in Table 3–2.

The second function, **SendMessage**, dispatches a message directly to the relevant window procedure. In this case, the processing of the message takes place synchronously: by the time control returns from the **SendMessage** call, the window procedure will already have processed the message to completion.

As on the Macintosh, you can define message types of your own and use **PostMessage** or **SendMessage** for internal coordination within your program itself. The Macintosh Event Manager originally reserved four of the sixteen possible event types for application use, though one of them has since been reclaimed by the system. In Windows, you can use any message type from **0x0400** to **0x7FFF** for your own purposes. The Windows API header files define a constant named **WM_USER** denoting the lower limit of this range (**0X0400**). Such messages are valid only for sending private messages from one window to another within the same window class (that is, under the control of the same window procedure). For communication between windows of different classes (those belonging to different application programs, for instance), you have to obtain an officially registered message type from the Windows function **RegisterWindowMessage**. Message types in the range **0xC000** to **0xFFFF** are reserved for messages in this category.

Table 3–6 summarizes some common Windows functions relating to messages and message queues. We've already discussed some of them in this chapter; you can learn about the rest in the *Win32 Programmer's Reference.*

Table 3–6. Common message functions

Function	Mac counterpart	Purpose
GetMessage	**WaitNextEvent**	Retrieve next message from queue or yield control until one arrives
PeekMessage	**GetNextEvent**	Retrieve next message from queue, if any
WaitMessage	——————	Yield control until message arrives in queue
GetQueueStatus	**EventAvail**	Check queue for presence of selected messages
GetInputState	——————	Any mouse or keyboard messages pending in queue?
TranslateMessage	——————	Translate raw keyboard message to character equivalent
DispatchMessage	——————	Dispatch message retrieve from queue to window procedure
GetMessagePos	**EventRecord.where**	Mouse position at time of message
GetMessageTime	**EventRecord.when**	Time message was posted
PostMessage	**PostEvent**	Place message in queue
SendMessage	**PostEvent**	Send unqueued message directly to window procedure
RegisterWindowMessage	——————	Define message type for interapplication communication
PostQuitMessage	——————	Signal program completion

- The Macintosh Toolbox communicates with programs by sending them events.
- Windows communicates with programs by sending them messages.

- Macintosh programs are based on an event loop that retrieves and processes events one at a time.
- Windows programs are based on a message loop that retrieves and processes messages one at a time.

- A Macintosh program receives the user's mouse clicks and keystrokes in the form of events.
- A Windows program receives the user's mouse clicks and keystrokes in the form of messages.

- A Macintosh event is represented by an event record.
- A Windows message is represented by a message structure.

- A Macintosh event record has fields giving the type of event, the time it was posted, and the coordinates of the mouse at the time.
- A Windows message structure has fields giving the type of message, the time it was posted, and the coordinates of the mouse at the time.

- Each Macintosh program has an event queue in which the system posts events and the program retrieves them with **`GetNextEvent`** or **`WaitNextEvent`**.
- Each Windows program (or each thread within a multithreaded program) has a message queue in which the system posts messages and the program retrieves them with **`GetMessage`**.

...Only Different

- Macintosh events are directed to the program as a whole.
- Windows messages are directed to a particular window within the program.

- A Macintosh program's event loop processes each event directly by examining its type and transferring control to the appropriate part of the program to respond to it.
- A Windows program's message loop passes each message to the Windows function **`DispatchMessage`**, which in turn dispatches the message back to the program's window procedure; the window procedure then examines the message type and transfers control to the appropriate part of the program to respond to it.

- The Macintosh has sixteen event types.
- Windows has hundreds of message types.

Macintosh	Windows
• Most Macintosh events report user-level actions such as mouse clicks and keystrokes.	• Most Windows messages report higher-level actions such as moving, resizing, or scrolling a window or sending it a menu command.
• Macintosh programs must explicitly respond to every event themselves.	• Windows provides a default window procedure that defines a standard response to each message; programs needn’t handle an event explicitly unless their response differs from the default.
• Macintosh events have a single type-dependent parameter.	• Windows messages have two type-dependent parameters, one nominally word-length and the other a long word; in Win32, both are actually long words.
• Conceptually, all Macintosh events are posted to the event queue and retrieved with **`GetNextEvent`** or **`WaitNextEvent`**.	• Many Windows messages are not posted to the message queue, but sent directly to the relevant window procedure.
• A Macintosh program has to hit-test and decode its own mouse clicks (with help from the Toolbox) and call the appropriate Toolbox routines to respond to them.	• A Windows program can let the default window procedure hit-test and decode its mouse clicks, converting them into high-level messages representing the meaning of the click.