Chapter 5

Mouse and Keyboard

One of the most striking things about our example program, WiniEdit, is that it never has to deal directly with the user's mouse clicks and keystrokes. This is because it relies on the system's default window procedure, **`DefWindowProc`**, and the predefined window class **`EDIT`** to provide standard responses to the user's input actions: moving and manipulating windows on the screen, choosing menu items, selecting and typing text in the window's edit control, and so on. In the real world, of course, application programs have to be able to provide new functionality beyond what's already built into the system. This means intercepting the user's input actions and responding to them in the program's own way, according to the needs of its particular application. In this chapter, we'll learn how to receive and respond to input from the mouse and keyboard.

Like so many other things in the IBM/DOS world, the mouse has never been standardized. While Macintosh programmers could always count on a standard, stable one-button mouse, their DOS colleagues have had to deal with a proliferation of possibilities. The most common configuration is the two-button Microsoft mouse, but there are also systems in use with a one-button mouse, a three-button mouse, or no mouse at all. The Windows system is designed to support all of these possibilities, and Windows programs have to be prepared to accommodate any of them.

One thing this means in practical terms is always to provide a keyboard-based alternative to anything the user can do with the mouse. We'll see later how Windows supports this idea with alternative keyboard interfaces to such things as menus, controls, and scroll bars. In planning your own user interface, you should also give some thought to how your mouseless users can access your program's features from the keyboard alone.

Another factor to consider is how you'll deal with varying numbers of buttons on the mouse. By convention, the button on a one-button mouse is considered to be the left; those on a two-button mouse, the left and right. So one approach to button uncertainty is simply to ignore the right and middle buttons and provide a one-button interface based solely on the left button. This is certainly the easiest strategy for porting an existing Macintosh program to Windows. A variation on the same idea would be to treat all three buttons interchangeably, by intercepting all messages

referring to the right and middle buttons and reposting them with **SendMessage** or **PostMessage** as equivalent left-button messages.

A more elaborate approach would be to use the right and middle buttons for convenience features that can also be accessed in other ways. For instance, many Windows programs use the right mouse button to pop up a context-sensitive floating menu offering the most common or useful operations for a particular area of the screen. As long as all of the same commands are also available via the menu bar in the usual way, this “context menu” is simply a handy extra that you can offer to those users equipped to take advantage of it, without seriously penalizing those who aren’t. Similarly, you might use the middle button as an alternative to Shift-click for extending an existing selection: again, nothing significant is lost, since users without a middle button can still perform the operation in another way. (Most Windows programs simply ignore the middle button entirely.)

If you design your programs this way, you don’t normally need to know whether there’s a mouse installed or how many buttons it has: you simply write your window procedure to handle clicks from the various mouse buttons *in case* you receive any. If you do need to find out what kind of mouse is available on the user’s machine, you can use the Windows function **GetSystemMetrics**. This is a general-purpose function that can provide a whole range of information on the current system configuration—including, but not limited to, the mouse characteristics. The function takes a single parameter, an integer selector that identifies the particular item of information you want. All of the selectors are defined as interface constants beginning with the prefix **SM_**, for “system metrics.” The call

```
hasMouse = GetSystemMetrics(SM_MOUSEPRESENT);
```

will return a boolean result telling whether the system has a mouse, and

```
nButtons = GetSystemMetrics(SM_CMOUSEBUTTONS);
```

will tell you how many buttons it has (or `0` if there is no mouse).

As a convenience to left-handed users, those with a two- or three-button mouse can use the Mouse control panel to swap the functions of the left and right buttons. There’s a Windows function, **SwapMouseButton**, for doing this, but you shouldn’t normally call it yourself, since it affects the global behavior of the system for all programs, not just your own. It’s best simply to honor whatever preference the user has set through the control panel. If necessary, though, you can call **GetSystemMetrics** with the selector **SM_SWAPBUTTON**

```
isSwapped = GetSystemMetrics(SM_SWAPBUTTON);
```

to find out whether the buttons are currently swapped. Another property the user can set with the control panel is the mouse’s tracking speed, which you can find out by calling the Windows function **SystemParametersInfo** with the selector **SPI_GETMOUSE**. The selector **SPI_SETMOUSE** lets you change the mouse speed, but again this is not something you should normally be doing from within an application program.

Mouse Messages

When the user moves the mouse or presses or releases one of its buttons, Windows reports this event to your program by sending one of the messages listed in Table 5–1. The message is normally sent to the window that contains the mouse at the time of the event. (There's an exception to this rule, which we'll learn about shortly.) So the first thing Windows has to do is figure out which window the mouse is in. It then sends the window a message of type **WN_NCHITTEST** to narrow down the specific part of the window where the event occurred (in the title bar, menu bar, sizing border, client area, or whatever). The correct handling of this very important message is critical to the proper functioning of the entire Windows user interface. It's essential for your program to pass it through to the default window procedure, **DefWindowProc**, for processing.

Table 5–1. Mouse messages

Message type	Meaning
WM_NCHITTEST	Test mouse's location on screen
WM_LBUTTONDOWN	Left mouse button pressed in client area
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
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

The **lParam** parameter to **WM_NCHITTEST** contains the current mouse position in screen-relative coordinates (that is, relative to an origin at the top-left corner of the

screen). The default window procedure responds to this message by testing this location against the various regions of the window and returning one of the result codes shown in Table 5–2. Most of the possible results refer to parts of the window's nonclient area, such as the title bar or sizing border. On receiving one of these results, Windows sends the window an appropriate nonclient-area mouse message, such as **WM_NCLBUTTONDOWN** or **WM_NCMOUSEMOVE**, with the hit-test result and the screen-relative mouse coordinates as parameters. These messages are really intended to be handled by the Windows system itself, and most programs simply pass them to the default window procedure instead of processing them explicitly.

Table 5–2. Hit-test results

Name	Meaning
HTCAPTION	Title bar
HTREDUCE	Minimize box
HTZOOM	Maximize/restore box
HTMENU	Menu or menu bar
HTSYSMENU	System menu
HTBORDER	Nonsizing border
HTTOP	Sizing border, top edge
HTBOTTOM	Sizing border, bottom edge
HTLEFT	Sizing border, left edge
HTRIGHT	Sizing border, right edge
HTTOPLEFT	Sizing border, top-left corner
HTTOPRIGHT	Sizing border, top-right corner
HTBOTTOMLEFT	Sizing border, bottom-left corner
HTBOTTOMRIGHT	Sizing border, bottom-right corner
HTVSCROLL	Vertical scroll bar
HTHSCROLL	Horizontal scroll bar
HTSIZE	Size box
HTGROWBOX	Size box
HTCLIENT	Client area
HTTRANSPARENT	Window covered by another window
HTERROR	Screen background; beep
HTNOWHERE	Screen background; no beep

If the hit-test message returns a result of **HTCLIENT**, Windows generates a client-area mouse message, such as **WM_LBUTTONDOWN** or **WM_MOUSEMOVE**, instead of a nonclient-area message. In this case, before passing the mouse coordinates as a message parameter, it converts them to client-relative form, in which the origin of

the coordinate system is the top-left corner of the window's client area instead of the full screen. The other parameter, **wParam**, is a flag word giving the state of all the mouse buttons and the keyboard modifier keys at the time of the click or move; it is thus analogous to the **modifiers** field of a Macintosh event record. The constants listed in Table 5–3 are bit masks for extracting the individual flags from this parameter.

Table 5–3. Mouse-key modifiers

Name	Meaning
MK_SHIFT	Shift key down
MK_CONTROL	Control key down
MK_LBUTTON	Left mouse button down
MK_MBUTTON	Middle mouse button down
MK_RBUTTON	Right mouse button down

Mouse Movements

On the Macintosh, mere movements of the mouse don't generate events in themselves. There is a mouse-moved event, but it's used strictly for cursor management: you get one only when the mouse moves outside a region that you've specified as valid for the current cursor shape. In Windows, you receive a mouse-moved message (**WM_MOUSEMOVE** or **WM_NCMOUSEMOVE**) every time the user moves the mouse. You can use this message to track the mouse when the user drags it while holding down one of the buttons.

Listing 5–1 shows an example in which the mouse is used to draw out a rectangle between two diagonally opposite corners within a window's client area. The tracking operation begins when we receive a **WM_LBUTTONDOWN** message reporting that the left mouse button was pressed inside the client area. We retrieve the window's client rectangle with the Windows function **GetClientRect**, call another Windows function, **ClipCursor**, to confine the mouse's movements within that rectangle, and save the current mouse coordinates as the starting point of the rectangle to be drawn. Then, each time we receive a **WM_MOUSEMOVE** message with the left button still down, we extend the rectangle to the new ending point. If a tracking operation is already in progress (indicated by the global program flag **tracking**), we first erase the rectangle drawn by the previous **WM_MOUSEMOVE** message; then we save the mouse coordinates as the new ending point, redraw the rectangle, and set the **tracking** flag to force the rectangle to be erased on the next iteration. When the user finally releases the button, we will receive a **WM_LBUTTONUP** message: we call **ClipCursor** again with a **NULL** parameter to cancel the confinement of the mouse, then clear the **tracking** flag to end the tracking operation. The next time the left button is pressed, a new tracking operation will begin.

Listing 5–1. Tracking the mouse

```
BOOL   tracking = FALSE;                                  // Currently tracking mouse?
POINT  startPoint;                                        // Starting point for tracking mouse
POINT  endPoint;                                          // Ending point for tracking mouse


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

   // Get and process one message.

   {
      LONG  result = 0;                                      // Function result


      switch ( msgCode )                                  // Dispatch on message code
         {
            . . . ;

            case WM_LBUTTONDOWN:
               DoLButtonDown (thisWindow, wParam, lParam);     // Handle WM_LBUTTONDOWN message
               break;

            case WM_MOUSEMOVE:
               DoMouseMove (thisWindow, wParam, lParam);       // Handle WM_MOUSEMOVE message
               break;

            case WM_LBUTTONUP:
               DoLButtonUp (thisWindow, wParam, lParam);       // Handle WM_LBUTTONUP message
               break;

            . . . ;

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

         }  /* end switch ( msgCode ) */


      return result;


   }  /* end DoMessage */
```

Listing 5–1. Tracking the mouse (*continued*)

```
VOID DoLButtonDown (HWND thisWindow, WPARAM wParam, LPARAM lParam)

   // Handle WM_LBUTTONDOWN message.

   {
      RECT  clientRect;                                    // Window's client rectangle


      GetClientRect (thisWindow, &clientRect);             // Get client rectangle in client coordinates
      /* Convert clientRect to screen-relative coordinates */

      ClipCursor (&clientRect);                            // Confine mouse movements to client rectangle

      startPoint.x = LOWORD(lParam);                      // Extract mouse coordinates
      startPoint.y = HIWORD(lParam);                      //    and save as starting point

   }  /* end DoLButtonDown */


VOID DoMouseMove (HWND thisWindow, WPARAM wParam, LPARAM lParam)

   // Handle WM_MOUSEMOVE message.

   {
      HDC  windowContext;                                 // Handle to device context for drawing in window


      if ( wParam & MK_LBUTTON );                         // Is left button down?
         {
            /* Get device context for drawing in window */;

            if ( tracking )                               // Have we already begun tracking?
               /* Erase previous rectangle from startPoint to endPoint */;

            endPoint.x = LOWORD(lParam);                     // Extract mouse coordinates
            endPoint.y = HIWORD(lParam);                     //    and save as ending point

            /* Draw new rectangle from startPoint to endPoint */;

            tracking = TRUE;                              // Indicate tracking in progress

            /* Release device context */;

         }  /* end if ( wParam & MK_LBUTTON ) */

   }  /* end DoMouseMove */
```

Listing 5-1. Tracking the mouse (*continued*)

```
VOID DoLButtonUp (HWND thisWindow, WPARAM wParam, LPARAM lParam)

   // Handle WM_LBUTTONUP message.

   {
      ClipCursor (NULL);                                  // Cancel mouse confinement

      tracking = FALSE;                                      // Indicate end of tracking sequence

   }  /* end DoLButtonUp */
```

Double Clicks

Windows considers a double mouse click to have occurred when the same mouse button is pressed twice in succession within a limited distance on the screen and a limited interval in time. The parameters defining these limits are set by the user with the Mouse control panel. You can find out the time interval with the Windows function **GetDoubleClickTime**, and the spatial limits by calling **GetSystemMetrics** with the selectors **SM_CXDOUBLECLK** and **SM_CYDOUBLECLK**. You can also change these values with the **SystemParametersInfo** function, using the selectors **SPI_SETDOUBLECLKWIDTH**, **SPI_SETDOUBLECLKHEIGHT**, and **SPI_SETDOUBLECLKTIME**, or with the function **SetDoubleClickTime**. However, as with the mouse speed and button swap, it's best not to disturb the user's control panel settings.

Double clicks will not be reported to a window unless you have specified the **CS_DBLCLKS** style option in the **style** field of the **WNDCLASS** parameter structure when registering the window's class. If you have, then the second click of a double-click sequence will be reported with a double-click message *in place of* the usual button-down message. Button-up messages are still reported normally. So, for instance, a double click of the left button will generate the following sequence of messages:

```
WM_LBUTTONDOWN
WM_LBUTTONUP
WM_LBUTTONDBLCLK
WM_LBUTTONUP
```

This method of reporting double clicks raises an interesting difficulty: there's no way to tell, when you receive the initial button-down message, whether it's the start of a double-click sequence or just an ordinary single click. Nor can you defer the decision until you receive the ensuing double-click message, since it may never arrive. The only reasonable way to handle the situation is to define the meaning of the first click as a self-contained operation and the second click as an extension of the same operation, in such a way that both can be processed independently when they arrive. In a text-editing program, for instance, you can use the first click to position the insertion caret between characters and the second to widen it into a full-word

selection. If the second click never arrives, the first will already have been processed completely and correctly. (Luckily, WiniEdit doesn't have to deal with this issue, since all of its text selection is handled automatically by the system's built-in window procedure for edit controls.)

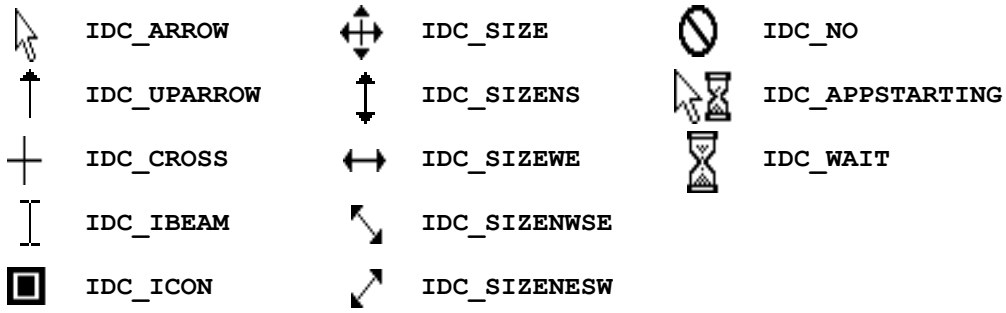
Capturing the Mouse

We mentioned earlier that there is an exception to the rule that mouse messages are always reported to the window in which they occur. In some circumstances, it may be appropriate for a window to *capture* the mouse, so that it will receive all mouse clicks and movements, even if they take place outside the window's borders. The usual reason for this is to continue tracking the mouse while the user holds down one of its buttons, even if it's dragged outside the window with the button still down. (Our example in Listing 5–1 addressed this issue in a different way, by confining the mouse within the window's client area with **`ClipCursor`**.) The Windows function **`SetCapture`** directs all mouse messages to a specified window until further notice; **`ReleaseCapture`** restores normal mouse reporting; **`GetCapture`** returns a handle to the window that has captured the mouse, if any.

Capturing the mouse is always a *temporary* state of affairs, to be used only for a limited time and a specific purpose. For instance, if you capture the mouse in order to track it during a drag, be sure to release it on receiving the mouse-up message ending the tracking operation: otherwise, no other window will be able to receive mouse input and the user will be unable to activate another window by clicking in it. A thread can capture the mouse only for one of its own windows, never for a window belonging to another thread. The capturing window should always be the active (frontmost) window on the screen. If it isn't, the capture will not behave as expected: mouse events will be clipped to the window's visible region, with those outside the region going unreported.

Windows cursors are pretty much the same as their Macintosh counterparts: they have a color or monochrome bit image, a monochrome bit mask, and a hot spot. The image defines the cursor's appearance on the screen, the mask works like a cookie-cutter to define its shape, and the hot spot defines the point within the image that marks the actual mouse location. However, because Windows has to work with a variety of display devices at different resolutions and aspect ratios, the dimensions of the cursor aren't standardized, as they are on the Macintosh at 16 pixels by 16. The cursor's width and height aren't even necessarily equal, since some devices use square pixels and others use rectangular ones with different horizontal and vertical resolutions. The cursor dimensions for the currently configured display device are available via the **`GetSystemMetrics`** function, using the selectors **`SM_CXCURSOR`** and **`SM_CYCURSOR`**.

Figure 5–1. Stock cursors



You can create a cursor dynamically at run time with the Windows function **CreateCursor**, but the usual practice is to read it in as a resource with **LoadCursor**. Both functions return a cursor handle of type **HCURSOR**. Like all resource-loading functions, **LoadCursor** takes two parameters: a handle to the program instance in whose executable file the resource resides, and a second parameter identifying the individual resource within the file. As we learned in Chapter 2, this second parameter is nominally a string pointer to the resource name, but in reality it can instead be an integer resource ID converted to string form with the **MAKEINTRESOURCE** macro. A null instance handle refers to one of the standard or "stock" cursors shown in Figure 5–1; the second parameter must then be one of the constants in Table 5–4. (The prefix **IDC_** stands for "identifier, cursor.") These constants are already typecast into string pointers, so it isn't necessary to use **MAKEINTRESOURCE** on them. For example, the statement

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

loads the standard arrow cursor. The stock cursors actually exist in a variety of dimensions and color depths for use on different display devices; **LoadCursor** automatically selects the one best suited to the current screen.

Each time the user moves the mouse within a window's client area, Windows sends the window the message **WM_SETCURSOR**, with the hit-test code returned by **WM_NCHITTEST** as one of its parameters. By processing this message, you can test the cursor's location and set its shape accordingly with the Windows function **SetCursor**. The standard window procedure does this automatically for the window's non-client area, depending on the hit-test code: for instance, if the hit-test value is **HTTOPLEFT**, denoting the top-left corner area of the window's sizing border, Windows will set the cursor to **IDC_SIZENWSE**, the double-headed arrow pointing northwest and southeast.

The easiest way to manage the cursor, though, is to specify a *class cursor* when you register your window class. Windows will automatically set the cursor to this shape whenever it moves into the client area of a window belonging to the class. If you need to divide your window into distinct functional areas with different cursor shapes, you can do it by creating a "transparent" child window (one with no visible border) for each area, and giving each such child window a different class cursor. When the child window receives the **WM_SETCURSOR** message, it will relay it back to the parent window. If it wishes, the parent window itself can set the cursor and

return a result of **TRUE** to indicate that the cursor is already taken care of; or it can return **FALSE** and let the child window do the job.

> If you decide to do your cursor management by hand, be sure to set the class cursor to **NULL** when you register your window class; otherwise, the cursor will flicker on the screen as your program and Windows fight a tug-of-war over its shape every time the user moves the mouse. Also, remember that the cursor is a system-wide resource that you share with other programs. Don't change its shape unless it's within the client area of one of your windows or unless you've captured the mouse; and if you do capture the mouse, remember to restore the cursor to its previous shape before releasing it.

Table 5–4. Stock cursors

Name	Description	Purpose
IDC_ARROW	Northwest arrow	General-purpose
IDC_UPARROW	Up arrow	General-purpose
IDC_CROSS	Crosshairs	Precise location
IDC_IBEAM	I-beam	Text selection
IDC_ICON	Empty icon	Program selection
IDC_SIZE	Four-way arrow	Move or size window
IDC_SIZEALL	Four-way arrow	Move or size window
IDC_SIZENS	North–south double arrow	Size window, top or bottom edge
IDC_SIZEWE	West–east double arrow	Size window, left or right edge
IDC_SIZENWSE	Northwest–southeast double arrow	Size window, top-left or bottom-right corner
IDC_SIZENESW	Northeast–southwest double arrow	Size window, top-right or bottom-left corner
IDC_NO	Circle with diagonal slash	Prohibited operation
IDC_APPSTARTING	Arrow with hourglass	Waiting for program to load
IDC_WAIT	Hourglass	General delay

You can get a handle to the current cursor with the Windows function **GetCursor** and retrieve or change its coordinates on the screen with **GetCursorPos** or **SetCursorPos**. On a system without a mouse, **SetCursorPos** is handy for moving the cursor around on the screen with the keyboard arrow keys. The **ClipCursor** function confines the cursor within a specified rectangle, pinning it at the rectangle's edges and preventing it from traveling outside. (We've already seen an example of this function in action in Listing 5–1.) You can find out the current confinement rectangle, if any, with **GetClipCursor**; this is useful for saving and restoring the previous confinement state.

The Windows function **ShowCursor** hides or shows the cursor, depending on the

value of a boolean parameter. Like its Macintosh counterpart, the Windows cursor maintains an integer visibility level that records how many times it has been hidden and not yet reshown. Hiding the cursor decrements the visibility level by one;

showing it increments the level. A negative visibility level makes the cursor invisible on the screen. The visibility level is ordinarily initialized to **0**, but on systems without a mouse, it's initialized to **-1** instead, making the cursor invisible by default.

If you've created a cursor from scratch with **CreateCursor**, you must destroy it with **DestroyCursor** before exiting from your program. It isn't necessary to destroy a cursor that you've loaded as a resource with **LoadCursor**.

The other primary input device on all Windows systems is the keyboard. The keyboard is designed mainly for entering text, but it can also be used in place of the mouse for manipulating menus, windows, and controls on the screen. Although it's possible, as WiniEdit demonstrates, to write a program that never interacts explicitly with the keyboard, most real-world programs will need to receive and process keyboard input in some form. Table 5–5 lists the main message types related to keyboard input; we'll be discussing them in the remainder of this section.

Table 5–5. Keyboard messages

Message type	Meaning
WM_KEYDOWN	Key pressed
WM_KEYUP	Key released
WM_SYSKEYDOWN	Key pressed with Alt key
WM_SYSKEYUP	Key released with Alt key
WM_CHAR	Character typed
WM_DEADCHAR	Dead character typed
WM_SYSCHAR	Character typed with Alt key
WM_SYSDEADCHAR	Dead character typed with Alt key
WM_SETFOCUS	Window acquiring input focus
WM_KILLFOCUS	Window losing input focus

Input Focus

No more than one window at a time can be the target of the user's keyboard input. The window so designated is said to have the *input focus.* The window with the focus is always either the active window or one of its children. In a dialog box, for instance, the focus normally belongs to one of the dialog's controls, identified by a dotted *focus rectangle* around the text of a button control or an insertion caret or selection highlight in an edit control. The standard window procedure responds to the keystrokes Tab and Shift-Tab by cycling the focus forward and backward among the dialog's controls, and to the space bar by activating or toggling the button with the focus as if it had been clicked with the mouse.

A window normally acquires the input focus when it becomes the active window. You can also set the focus explicitly with the Windows function **SetFocus**, or find out which window has it with **GetFocus**. Windows signals when the focus shifts from one window to another by sending the message **WM_KILLFOCUS** to the window losing the focus and **WM_SETFOCUS** to the one gaining it. WiniEdit's document window, for instance, responds to this message by relaying the focus to its child, the edit control (see Listing 5–2). The edit control's built-in window procedure will then receive and handle all keyboard input for as long as the document window remains active.

A window in the minimized state (reduced to a button in the Windows 95 task bar) can't respond to keyboard input. In this case, no window is considered to have the input focus: all keyboard input is then directed to the system itself, rather than to any individual window.

Listing 5–2. Handle WM_SETFOCUS message

```
VOID DoSetFocus (HWND thisWindow, WPARAM wParam, LPARAM lParam)

   // Handle WM_SETFOCUS message.

   {
      SetFocus (TheEditor);                    // Pass focus to the edit control

   }  /* end DoSetFocus */
```

Keystroke Messages

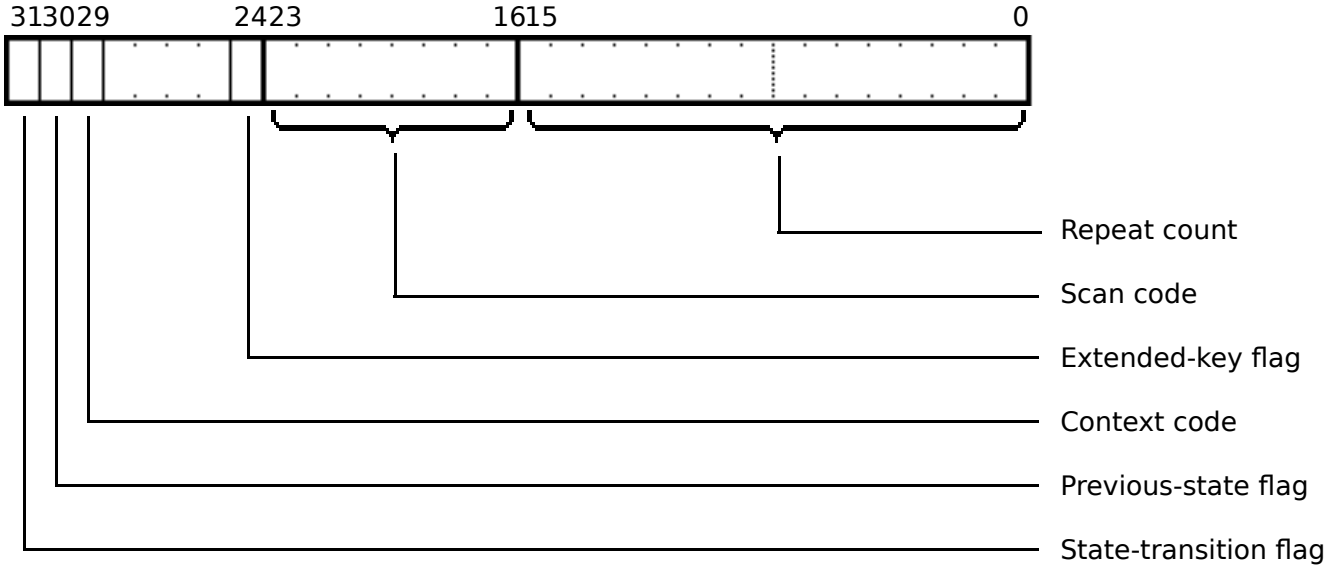
Every key on the keyboard is identified by a device-dependent *scan code* that identifies the key to the keyboard driver. The keyboard driver converts this scan code into a device-independent *virtual-key code,* using a keyboard layout configured by the user with the Keyboard control panel. The virtual-key code identifies the key by its purpose instead of its physical location on the keyboard, such as **VK_A** for the letter **A**, **VK_ENTER** for the Enter key, or **VK_SPACE** for the space bar. (See the *Win32 Programmer's Reference* for a complete table of virtual-key codes.)

When the user presses or releases a key, the keyboard driver posts a message, normally a **WM_KEYDOWN** or **WM_KEYUP**, to the global system message queue. (If the Alt key was being held down at the time, or if no window currently has the input focus, the message is **WM_SYSKEYDOWN** or **WM_SYSKEYUP** instead.) The message identifies the key by both its physical scan code and the corresponding virtual-key code, along with some other information that we'll talk about in a minute. The Windows system then moves the message from the system message queue into the private queue of the currently active thread. Eventually, the thread's message loop will retrieve and process the message.

All of these *keystroke messages* (**WM_KEYDOWN**, **WM_KEYUP**, **WM_SYSKEYDOWN**, **WM_SYSKEYUP**) receive the virtual-key code as their **wParam** parameter; the **lParam**

parameter contains a variety of information packed into a 32-bit long word (see Figure 5–2):

Figure 5–2. `lparam` parameter for keystroke messages



- The 8-bit raw scan code generated by the keyboard hardware.
- A 16-bit *repeat count.* Before posting a keyboard message to the queue, Windows looks first to see if there's one already there for the same keystroke. If so, it just increments the existing message's repeat count instead of posting another separate message. This typically happens when the user presses and holds a key, generating auto-repeat keystrokes faster than the program can retrieve them from the queue.
- A 1-bit *previous-state flag* showing whether the key was previously up or down. If this flag is set, it shows that the current message is part of a continuing series generated by an auto-repeating key.
- A 1-bit *state-transition flag* that tells whether the message was generated by the user's pressing a key (**`WM_KEYDOWN`** or **`WM_SYSKEYDOWN`**) or releasing one (**`WM_KEYUP`** or **`WM_SYSKEYUP`**).
- A 1-bit *extended-key flag* indicating whether the keystroke was generated by one of the additional keys on the extended keyboard. Table 5–6 lists the extended keys that cause this flag to be set. In general, they are those in the extra clusters to the right of the main keyboard layout, such as the arrow keys and the keypad. For some reason, however, not all of the keypad keys are considered extended keys: only the Enter, Divide (**`/`**), and Num Lock keys are included. Most programs don't pay much attention to the extended-key flag, but it can sometimes be useful for distinguishing between otherwise functionally equivalent keys, such as the left and right Ctrl or Alt keys or the keyboard and keypad Enter keys.

- A 1-bit *context code* that tells whether the Alt key was down at the time of the keystroke. In the case of a system keystroke (**WM_SYSKEYDOWN** or **WM_SYSKEYUP**), this flag distinguishes whether the keystroke is directed to the system because of the Alt key or because no window has the input focus.

Table 5-6. Extended keys

Virtual-key code	Key
VK_MENU	Right-hand Alt key
VK_CONTROL	Right-hand Ctrl key
VK_INSERT	Insert
VK_DELETE	Delete
VK_HOME	Home
VK_END	End
VK_PRIOR	Page Up
VK_NEXT	Page Down
VK_LEFT	Left arrow
VK_RIGHT	Right arrow
VK_UP	Up arrow
VK_DOWN	Down arrow
VK_RETURN	Keypad Enter key
VK_DIVIDE	Keypad Divide (**/**)
VK_NUMLOCK	Num Lock
VK_SNAPSHOT	Print Screen (**F13**)
VK_SCROLL	Scroll Lock (**F14**)
VK_PAUSE	Pause/Break (**F15**)

In general, raw keystroke messages (as opposed to the character-oriented messaages discussed in the next section) are most useful for handling non-character input such as the arrow and function keys. If you do need to process raw keystrokes, you should normally confine yourself to the non-system messages, **WM_KEYDOWN** and **WM_KEYUP**. The default window procedure, **DefWindowProc**, ignores these messages, but it uses the system versions, **WM_SYSKEYDOWN** and **WM_SYSKEYUP**, to implement many of the standard keyboard conventions that Windows users expect, such as the Alt-key interface for menu selection that we'll be discussing in Chapter 9. If the default window procedure doesn't receive system keystrokes, your user will lose access to these standard Windows features. So if you absolutely must intercept system keystrokes yourself, be sure you pass them on to **DefWindowProc** *in addition to* any processing you may be doing on your own. Your users will thank you.

Character Messages

On the Macintosh, a single keyboard event (key-down, key-up, or auto-key) contains codes identifying both the physical key that was pressed and the logical character it represents. In Windows, the low-level keystroke messages that we've discussed so far contain no reference to specific text characters. While it's true that many virtual-key codes refer to character keys, such as **VK_M** or **VK_8** or **VK_SPACE**, that doesn't mean that every keystroke generated by such a key necessarily represents that particular character. The **VK_M** key, for instance, may produce an uppercase **M** or a lowercase **m**, depending on the state of the Shift and Caps Lock keys; **VK_8** can produce either a numeral **8** or an asterisk (*****); or a keystroke may have been typed with the Alt or Ctrl key down and not represent any character at all. The task of mapping raw keystrokes into characters is performed by the Windows function **TranslateMessage**.

We've already encountered **TranslateMessage** in Chapter 3 as part of our NullApp program's main message loop routine (reproduced for reference in Listing 5–3). After retrieving a message with **GetMessage**, we pass it to **TranslateMessage** to see if it's a keystroke representing a text character. If so, **TranslateMessage** will generate a corresponding *character message* (**WM_CHAR** or **WM_SYSCHAR**) and post it to the message queue. The character message goes straight to the front of the queue instead of the end, so it doesn't get out of order with other possible keyboard messages that may be pending in the queue. **TranslateMessage** simply ignores any message that isn't a keystroke.

Listing 5–3. NullApp main program loop

```
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 */
```

All character messages identify the character typed via a character code in the message's **wParam** parameter. The **lParam** parameter holds the same information as in a raw keystroke message, as shown earlier in Figure 5–2. The contents of **lParam** are simply copied directly from the keystroke message (**WM_KEYDOWN** or **WM_SYSKEYDOWN**) that gave rise to the character message. Most of this information is normally of no interest, but don't forget to check the repeat count in case several strokes of the same key have been combined into a single message.

Polling the Keyboard

In determining what character a keystroke represents, **TranslateMessage** must take into account the state of the Shift and Caps Lock keys. If you look carefully, however, you'll find that this information is not included anywhere in the keyboard message's parameters: there is no equivalent to the **modifiers** field of a Macintosh event record. Instead, **TranslateMessage** has to poll the state of the modifier keys directly, using the Windows function **GetKeyState**. This function accepts a virtual-key code as a parameter and reports whether the key is up or down. For keys that toggle a persistent state, such as Caps Lock, Num Lock, and Scroll Lock, it also reports whether the state is currently toggled on or off. Note that this function applies to just a single designated key; a related function, **GetKeyboardState**, returns an array of state information for the entire keyboard.

You might think that polling the keyboard this way is an unreliable way of checking the modifier keys for a keyboard event, since the state of the keys may have changed between the time the event was posted and the time it's processed. It turns out, however, that the **GetKeyState** and **GetKeyboardState** functions keep track of such changes and report the state of the keyboard *at the time the current keyboard message was posted,* not its current state at the time it's actually polled. This makes these functions ideal for processing keystrokes one by one out of the queue, but it also means you can't rely on them for polling the *current* state of the keyboard, right this instant. Luckily, there's yet another Windows function, **GetAsyncKeyState**, that reports the state of a key *right now,* without reference to any previously posted message.

Dead Keys

In some keyboard layouts, some of the keys may generate accents or diacritical marks for use in languages other than English, such as an acute accent (´), umlaut (¨), or circumflex (ˆ). Such characters are intended to combine with the next character typed to produce an accented character, such as **é**, **ü**, or **ô**. The accented letter is represented by a character code of its own, different from those of both the accent itself and the base letter to which it attaches. (That is, the character code for **é** is different from those for ´ and **e**.) Since the accent key by itself doesn't generate a text character until after the next keystroke, it is known as a *dead key;* the accent character it represents, which combines with the next character typed to form a single character, is called a *dead character.*

When the Windows **TranslateMessage** function determines that a keystroke it has received represents a dead character, it generates a message of type **WM_DEADCHAR** (or **WM_SYSDEADCHAR** for a system keystroke) instead of the usual **WM_CHAR** message. Then, when it receives the next keystroke, it combines that character with the dead character and generates a **WM_CHAR** message carrying the character code for the resulting combination character (**é** in the previous example). Thus the overall sequence of messages generated goes like this:

WM_KEYDOWN		**WM_SYSKEYDOWN**
WM_DEADCHAR (´)		**WM_SYSDEADCHAR (´)**
WM_KEYUP	**or, for a system keystroke,**	**WM_SYSKEYUP**
WM_KEYDOWN		**WM_SYSKEYDOWN**
WM_CHAR (é)		**WM_SYSCHAR (é)**
WM_KEYUP		**WM_SYSKEYUP**

If the next character typed after the dead character is not one that it can combine with (such as a consonant instead of a vowel following an acute accent), **TranslateMessage** treats them as separate characters and generates *two* **WM_CHAR** messages. For example, typing an acute accent followed by a lowercase **m** would result in the following sequence of messages:

WM_KEYDOWN		**WM_SYSKEYDOWN**
WM_DEADCHAR (´)		**WM_SYSDEADCHAR (´)**
WM_KEYUP		**WM_SYSKEYUP**
WM_KEYDOWN	**or**	**WM_SYSKEYDOWN**
WM_CHAR (´)		**WM_SYSCHAR (´)**
WM_CHAR (m)		**WM_SYSCHAR (m)**
WM_KEYUP		**WM_SYSKEYUP**

Unless you have some special reason for caring about them, it’s usually safe to ignore dead-character messages: the dead character will always turn up eventually in an ordinary character message, either in its own right or as part of a combination character like **é**.

Mouse and Keyboard Operations

Table 5–7 summarizes some common Windows functions relating to mouse and keyboard input. We've already discussed some of them in this chapter; you can learn about the rest in the *Win32 Programmer's Reference.*

Table 5–7. Common mouse and keyboard functions

Function	Mac counterpart	Purpose
SetCapture	———	Capture mouse
GetCapture	———	Get window that has captured the mouse
ReleaseCapture	———	Release captured mouse
GetDoubleClickTime	**GetDblTime**	Get double-click interval
SetDoubleClickTime	———	Set double-click interval
SwapMouseButton	———	Reverse left and right mouse buttons
CreateCursor	———	Create new cursor
LoadCursor	**GetCursor**	Load cursor from template resource
CopyCursor	———	Make copy of cursor
DestroyCursor	———	Destroy cursor
GetCursor	———	Get current cursor
SetCursor	**SetCursor**	Set current cursor
SetSystemCursor	———	Change appearance of stock cursor
GetCursorPos	**GetMouse**	Get current cursor coordinates
SetCursorPos	———	Set cursor coordinates
ShowCursor	**ShowCursor**, **HideCursor**	Make cursor visible or invisible
ClipCursor	———	Confine cursor within a rectangle
GetClipCursor	———	Get current confinement rectangle
GetKeyState	———	Get state of key at time of message
GetKeyboardState	**GetKeys**	Get state of full keyboard at time of message
GetAsyncKeyState	———	Get current state of key
GetKeyNameText	———	Get name of key in string form
GetFocus	———	Get window with input focus

SetFocus	—————	Set window with input focus

Mouse

• A Macintosh program receives a mouse-down or mouse-up event when the user presses or releases the mouse button.	• A Windows program receives a button-down or button-up message when the user presses or releases a mouse button.

Cursor

• The Macintosh cursor has an image, a mask, and a hot spot.	• The Windows cursor has an image, a mask, and a hot spot.
• A Macintosh cursor can be created dynamically or read in as a resource.	• A Windows cursor can be created dynamically or read in as a resource.
• The Macintosh Toolbox has certain standard cursors built in as system resources and available to all programs.	• Windows has certain standard (“stock”) cursors built in as system resources and available to all programs.
• The Macintosh cursor maintains an integer visibility level to count how many times the cursor has been hidden and not yet reshown.	• The Windows cursor maintains an integer visibility level to count how many times the cursor has been hidden and not yet reshown.

Keyboard

• A Macintosh program receives a key-down or key-up event when the user presses or releases a key on the keyboard.	• A Windows program receives a key-down or key-up event when the user presses or releases a key on the keyboard.

...Only Different

Mouse

• The mouse (or other pointing device) is an integral part of every Macintosh system.	• The Windows mouse is an optional accessory; not every system has one.
• The Macintosh mouse always has exactly one button.	• The Windows mouse can have one, two, or three buttons.
• Macintosh mouse events are directed globally to the program as a whole.	• Windows mouse messages are directed specifically to the window in which they occur.

• Because mouse clicks are global to the entire program, the Macintosh has no concept of capturing the mouse.	• In Windows, a window can capture the mouse, receiving all subsequent mouse messages (even those occurring outside the given window) until it releases the mouse.
• Macintosh programs receive the same type of mouse-down event for any mouse click, no matter where it occurs on the screen.	• Windows programs receive different types of button-down messages, depending on whether the mouse was clicked in a window's client or nonclient area.
• Macintosh programs must respond explicitly to all mouse clicks, including those in a window's frame (title bar, size box, close box, zoom box) or in the menu bar.	• Windows programs need only respond to mouse clicks in a window's client area; those in the nonclient area (title bar, sizing border, menu bar, close box, minimize and maximize boxes) are handled automatically by the default window procedure.
• On the Macintosh, mere movements of the mouse don't generate events in themselves.	• Windows programs receive a mouse-moved message whenever the user moves the mouse.
• Macintosh programs must poll the mouse position in order to track its movements.	• Windows programs can use mouse-moved messages to track the mouse's movements.
• Macintosh programs must decide for themselves when a double click has occurred, by comparing the times and locations of consecutive mouse clicks.	• Windows programs receive a double-click message to tell them when a double click has occurred.

Cursor

• The Macintosh cursor is always the same size, 16 pixels wide by 16 high.	• The Windows cursor can vary in dimensions according to the characteristics (resolution and aspect ratio) of the screen on which it is displayed.
• A Macintosh program must manage the appearance of the cursor in all parts of the screen.	• A Windows program must manage the appearance of the cursor only in a window's client area; the Windows system manages it automatically in the nonclient area or outside the window.
• Macintosh programs manage the cursor either by polling its location on every pass of the event loop or by passing a valid region for the current cursor as a parameter to **`WaitNextEvent`**.	• Windows programs manage the cursor by receiving a **`WM_SETCURSOR`** message every time the user moves the mouse, or by specifying a default cursor for an entire class of windows.

Keyboard

• The Macintosh Toolbox reports both a physical key code and a logical character code in a single event.	• Windows sends two different types of keyboard messages: keystroke messages containing a physical key code and character messages containing a logical character code.
• Macintosh keyboard events are directed globally to the program as a whole.	• Windows keyboard messages are directed specifically to the window that has the input focus.
• Every Macintosh keyboard event represents a single keystroke, whether key-down, key-up, or auto-key.	• A Windows keyboard message has a repeat count, allowing it to represent several successive strokes of the same key.