Chapter 8: Event-Driven Input

8.1: Event-Driven Input Versus “Hurry Up and Wait”

Immediately after a new Win32 programmer gets over the shock of discovering that event-driven output is nothing like the type of programming they re used to, they take a look at event-driven input and go into shock again. Most programmers are comfortable using statement like readln and gets that they can call to read whole lines of text, lists of values, etc. This has become known as the hurry up and wait model because the programmer writes code that runs as quickly as possible between two input operations (so the user doesn t notice a delay between inputs) and then the computer sits idle for a very long time waiting for theuser to enter data from the keyboard.

The event-driven input model is far more efficient from the CPU s point of view. The operating system sends your application messages whenever there is some input to process. Therefore, the system isn t just sitting around waiting for the input to arrive - it can be doing other things (including passing other messages on to your application for processing).

Unfortunately, from the application s point of view, event-driven input is far more complex. With a statement like readln(i32); in Pascal or cin >> i32; in C++, you re telling the operating system to take charge until the user enters an integer value (assuming `i32` is an integer variable). The application doesn t have to deal with collecting the individual characters together that make up the string that represents this integer. Nor does the application have to deal with the conversion of that string to an integer value. More importantly, the application doesn t have to worry about dealing with other possible inputs while waiting for the user to input this integer value. All of these simplifications go away when working with input in an event-driven system like Windows.

The input model in Windows is actually quite simple. If an input event occurs (that is associated with your application), then Windows sends your application a message specifying the type of input event. The bad news is that these are very simple events, like someone just pressed a key on the keyboard or someone just pressed a button on the mouse. It is up to your application to interpret these events in an abstract manner (e.g., the user has pressed a sequence of keystrokes whose ASCII codes form a numeric string that the application can convert to an integer value).

While the concept is simple, there is a major problem with the Windows model: the user of your application can decide to pause the input of some integer value (for example) half-way through the input operation, switch to some other input box and enter data there, then return and finish the original input string. This fact adds conceptual complexity on top of programming complexity (that is, it s hard to visualize how the end-user might behave when designing the application, thus increasing the likelihood that your program may not consider some possible input sequence by the user). In this chapter, we re going to consider how to deal with these complexities.

Although the primary focus of this chapter is going to be on the keyboard, we re also going to look at mouse input and timer events in this chapter. Mouse (or pointer device) input is probably the second-most common form of input in use under Windows. Timer events are less common, but very important. A timer event lets you wake up an application by sending it a periodic (or timed) message even in the absence of any other events or messages directed at the application. We ll consider these three forms of input in this chapter.

8.2: Focus on Focus

Because a typical PC has only a single set of input devices, applications must share the use of these input devices. In particular, the operating system will typically send all input events some device generates to a single application. For example, the operating system only sends keystrokes to a single application s window (imagine

the confusion that would ensue if the OS sent the same keystroke messages to a word processor, a spreadsheet, and a database application simultaneously). To control which application (or, more specifically, which window in an application) receives input messages, Windows uses the concept of *input focus*.

Focus is associated with the currently active Window. This is the window to which Windows will send all keyboard (and possibly other) input events. If the user switches between active windows on their desktop, then the input focus switches to the new active window and all keyboard messages are sent to the new active window rather than the original window.

When the Windows makes some window the active window, it sends that application s window procedure a `w.WM_SETFOCUS` message so the application is made aware that it might be receiving keyboard messages before too much longer. Conversely, when the user switches from one window to another, Windows first sends the window that originally had the keyboard focus a `w.WM_KILLFOCUS` message to let it know that a focus change is about to occur and should take appropriate action.

8.3: Stick and Caret

Whenever a window is expecting keyboard input, it usually displays a flashing block, underline, vertical bar, or some other symbol to indicate the position on the window where the application will place any characters input via the keyboard. Although the common term for this symbol is *cursor*, Windows actually uses the term *caret* to describe the keyboard input position[1]. In this section we ll take a look at the functions that control the display of the caret within your application s window.

Because only one window can have the input focus, that is, only one application window can receive keyboard input, there is only one system-wide caret. It wouldn t do for two open application windows to be displaying that blinking caret - the poor user wouldn t be able to tell which window would receive the next input character that they type. Therefore, your applications can display the caret when they are given the input focus and they must relinquish the input caret when they give up the input focus. As noted above, Windows sends your application s window procedure the `w.WM_SETFOCUS` and `w.WM_KILLFOCUS` messages when it gives or retracts the input focus (respectively). By handling these messages, your applications can properly obtain and release the caret as needed.

When an application is given the focus (i.e., it receives a `w.WM_SETFOCUS` message) it should call the w.CreateCaret function to create a caret specifically for the application. The prototype for this function is the following:

```
static
    CreateCaret: procedure
    (
        hWnd            :dword;
        hBitmap         :dword;
        nWidth          :dword;
        nHeight         :dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__CreateCaret@16" );
```

The `hWnd` parameter is the handle of the window that will own the caret (e.g., your application s main window). The `hBitmap` parameter is the handle of a bitmap object that Windows will use for the caret. If this param-

1. In Windows terminology, cursor describes the screen object the denotes the position of the mouse pointer.

eter is NULL, Windows will not use a bitmap for the caret and will, instead, use the nHeight and nWidth parameters to define the caret. The nHeight and nWidth parameters specify a block-style cursor that is nWidth pixels wide and nHeight pixels high.

When your program gives up the focus (i.e., when you receive a w.WM_KILLFOCUS message), you must destroy the caret you ve created with a w.DestroyCaret API call. Here s the prototype for that function:

```
static
    DestroyCaret: procedure;
        @stdcall;
        @returns( "eax" );
        @external( "__imp__DestroyCaret@0" );
```

Note that this function doesn t require any parameters. Because there is only one system caret, Windows will destroy the only one in existence (the one associated with the window handle you originally passed to w.CreateCaret).

Whenever you create a caret via the w.CreateCaret API call, Windows creates an invisible caret. In order to actually display the caret you need to call the w.ShowCaret API function. While your application is holding the focus (and the caret), you can make the caret invisible again by calling the w.HideCaret API function. Here are their prototypes:

```
static
    ShowCaret: procedure
    (
        hWnd        :dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__ShowCaret@4" );

    HideCaret: procedure
    (
        hwnd            :dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__HideCaret@4" );
```

For both procedures, the hwnd parameter is the handle of the window that currently has the caret attached to it. This must be the same handle you originally passed to the w.CreateCaret function.

Windows maintains an internal caret visible counter that w.ShowCaret increments and w.HideCaret decrements. While this counter is positive, Windows shows the caret. When this value is zero, Windows hides the cursor. Therefore, if you call w.ShowCaret several times without a comparable number of calls to w.HideCaret, Windows will keep the caret visible until you ve made the corresponding number of calls to w.HideCaret.

Once you enable the display of the caret in your application s window, you control the position of the caret via the w.SetCaretPos function call. This function has the following prototype:

```
static
    SetCaretPos: procedure
    (
        X           :dword;
        Y           :dword
    );
```

```
        @stdcall;
        @returns( "eax" );
        @external( "__imp__SetCaretPos@8" );
```

The (X,Y) parameters specify the x-coordinate and y-coordinate of the caret in the client area of your window. In particular, these coordinates specify the upper-left hand corner of the caret bitmap or block in your window.

Armed with this information about the caret and the w.WM_SETFOCUS and w.WM_KILLFOCUS messages, it s now possible to write some code that will automatically show the caret whenever your application gains the focus and hides the caret whenever it loses the focus. Here are a couple of routines, SetFocus and KillFocus, that handle these messages and take the appropriate actions:

```
// SetFocus-
//
//  This procedure gets called whenever this application gains the
// input focus.

procedure SetFocus( hwnd: dword; wParam:dword; lParam:dword );
begin SetFocus;

    w.CreateCaret( hwnd, NULL, AverageCharWidth, AverageCharHeight );
    w.SetCaretPos( 0, 0 ); // "Home" the cursor
    w.ShowCaret( hwnd );
    xor( eax, eax );  // Return success

end SetFocus;


// KillFocus-
//
//  Processes the WM_KILLFOCUS message that gets sent whenever this
// application is losing the input focus.

procedure KillFocus( hwnd: dword; wParam:dword; lParam:dword );
begin KillFocus;

    w.HideCaret( hwnd );
    w.DestroyCaret();
    xor( eax, eax );  // Return success

end KillFocus;
```

These routines are not completely general. First of all, the SetFocus function always homes the cursor to position (0, 0). Second, you might not always want to show the caret whenever you get the focus (e.g., the application may not be prepared to accept keyboard input just because it has received the focus). Nonetheless, these two message handling procedures demonstrate how to call these five different API functions.

There are a couple of additional API functions related to the caret that you might find useful. The first of these is `w.GetCaretPos` which returns the current caret position in the window. The prototype for this function is the following:

```
static
    GetCaretPos: procedure
    (
        var lpPoint     :POINT
```

```
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__GetCaretPos@4" );
```

This function returns the current caret position in the `lpPoint` parameter you pass by reference (`w.POINT` objects have an `x` and a `y` field that receive the caret coordinates).

The `w.GetCaretBlinkTime` and the `w.SetCaretBlinkTime` functions have the following prototypes:

```
static
    GetCaretBlinkTime: procedure;
        @stdcall;
        @returns( "eax" );
        @external( "__imp__GetCaretBlinkTime@0" );

    SetCaretBlinkTime: procedure
    (
        uMSeconds       :dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__SetCaretBlinkTime@4" );
```

The `w.GetCaretBlinkTime` function returns the number of milliseconds that pass between inversions of the caret on the display (the alternating inversions of the bitmap are what causes the caret to blink). The `uMSeconds` parameter you pass to `w.SetCaretBlinkTime` specifies the number of milliseconds between inversions of the caret bitmap.

8.4: Keyboard Messages

Windows sends several different keyboard related messages to your applications. In fact, a single keystroke typically winds up sending *three* different messages to your application's window procedure. Fortunately, you can ignore many of the messages Windows sends to your applications. Most of the time, there are only two types of messages to which you will normally respond. Nevertheless, it's important to understand the purpose of each of these messages in order to properly process the keyboard messages that are important to you.

Windows will send the following keyboard messages to your application:

- w.WM_KEYDOWN
- w.WM_KEYUP
- w.WM_SYSKEYDOWN
- w.WM_SYSKEYUP
- w.WM_CHAR
- w.WM_SYSCHAR
- w.WM_DEADCHAR
- w.WM_SYSDEADCHAR

Most of these messages you can ignore. In fact, the vast majority of the time you can get by processing only `w.WM_KEYDOWN` and `w.WM_CHAR` messages.

To understand the purpose of each of these messages, a brief discussion of the keyboard s operation is necessary. The standard PC keyboard does not produce ASCII character code whenever you process a key. Instead, the keyboard sends out one of two numeric codes (known as scan codes). One scan code indicates that the user has just pressed the key (a *down code*), a second scan code indicates that the user has released the key (an *up code*). Usually, you will get one up code for each down code (i.e., the user presses and releases a key, generating the two key codes). The exception is when the user holds down a key long enough for it to being autorepeating. In this case you will get a sequence of down codes without a corresponding up code. When the user finally releases the key, you will get a single up code for that key.

Note that a user can hold a key down while pressing other keys. The system can use the down and up codes to determine if you are holding down one key (e.g., a shift or control key) while pressing and releasing other keys. This allows the system to translate a scan code for some key like A into different ASCII codes, based on whether you re holding down other keys while pressing the A . For example, A without a *modifier key* like shift, control, or alt, produces the a character; holding down shift while pressing A produces the A character; holding down the control or alt key produces CTRL-A or ALT-A, respectively.

An important thing to realize is that keyboard scan codes are not the same thing as ASCII codes. These are simply some numeric values that the hardware manufacturer chose when desiging the keyboard. Indeed, different manufacturers have been known to use different scan code sets for their computer keyboards. To eliminate problems with different scan code sets, Windows defines a virtual keycode set. People who write keyboard drivers translate their OEM scan codes into virtual keycodes and pass those keycodes on to applications. This allows applications to deal with a single set of codes rather than having to worry about differences in the underlying hardware. Table 8-1 lists the standard Windows virtual key codes.

Table 8-1: Windows Virtual Keycodes

Value	windows.hhf Constant	Description
$01	w.VK_LBUTTON	Left mouse button
$02	w.VK_RBUTTON	Right mouse button
$03	w.VK_CANCEL	Ctrl-Break
$04	w.VK_MBUTTON	Middle mouse button
$05	w.VK_XBUTTON1	Win 2K X1 mouse button
$06	w.VK_XBUTTON2	Win 2K X2 mouse button
$08	w.VK_BACK	Backspace
$09	w.VK_TAB	Tab
$0C	w.VK_CLEAR	5 on numeric keypad with Numlock off
$0D	w.VK_RETURN	Carriage Return (Enter) key
$10	w.VK_SHIFT	Either shift key
$11	w.VK_CONTROL	Either control key
$12	w.VK_MENU	Alt key
$13	w.VK_PAUSE	Pause key

Value	windows.hhf Constant	Description
$14	w.VK_CAPITAL	Capslock key
$1B	w.VK_ESCAPE	Escape key
$20	w.VK_SPACE	Space key
$21	w.VK_PGUP	PgUp key
$21	w.VK_PRIOR	PgUp key
$22	w.VK_NEXT	PgDn key
$22	w.VK_PGDN	PgDn key
$23	w.VK_END	End key
$24	w.VK_HOME	Home key
$25	w.VK_LEFT	Left arrow key
$26	w.VK_UP	Up arrow key
$27	w.VK_RIGHT	Right arrow key
$28	w.VK_DOWN	Down arrow key
$29	w.VK_SELECT	Special key on certain keyboards
$2A	w.VK_PRINT	Special key on certain keyboards (e.g., WinCE devices)
$2B	w.VK_EXECUTE	Special key on certain keyboards (e.g., WinCE devices)
$2C	w.VK_SNAPSHOT	Print Screen key
$2D	w.VK_INSERT	Ins key
$2E	w.VK_DELETE	Del key
$2F	w.VK_HELP	Special key on certain keyboards (e.g., WinCE)
$30	w.VK_0	0
$31	w.VK_1	1
$32	w.VK_2	2
$33	w.VK_3	3
$34	w.VK_4	4
$35	w.VK_5	5
$36	w.VK_6	6
$37	w.VK_7	7

Value	windows.hhf Constant	Description
$38	w.VK_8	8
$39	w.VK_9	9
$41	w.VK_A	A
$42	w.VK_B	B
$43	w.VK_C	C
$44	w.VK_D	D
$45	w.VK_E	E
$46	w.VK_F	F
$47	w.VK_G	G
$48	w.VK_H	H
$49	w.VK_I	I
$4A	w.VK_J	J
$4B	w.VK_K	K
$4C	w.VK_L	L
$4D	w.VK_M	M
$4E	w.VK_N	N
$4F	w.VK_O	O
$50	w.VK_P	P
$51	w.VK_Q	Q
$52	w.VK_R	R
$53	w.VK_S	S
$54	w.VK_T	T
$55	w.VK_U	U
$56	w.VK_V	V
$57	w.VK_W	W
$58	w.VK_X	X
$59	w.VK_Y	Y
$5A	w.VK_Z	Z
$60	w.VK_NUMPAD0	0 on the numeric keypad

Value	windows.hhf Constant	Description
$61	w.VK_NUMPAD1	1 on the numeric keypad
$62	w.VK_NUMPAD2	2 on the numeric keypad
$63	w.VK_NUMPAD3	3 on the numeric keypad
$64	w.VK_NUMPAD4	4 on the numeric keypad
$65	w.VK_NUMPAD5	5 on the numeric keypad
$66	w.VK_NUMPAD6	6 on the numeric keypad
$67	w.VK_NUMPAD7	7 on the numeric keypad
$68	w.VK_NUMPAD8	8 on the numeric keypad
$69	w.VK_NUMPAD9	9 on the numeric keypad
$6A	w.VK_MULTIPLY	* on the numeric keypad
$6B	w.VK_ADD	+ on the numeric keypad
$6C	w.VK_SEPARATOR	Special key on certain keyboards
$6D	w.VK_SUBTRACT	- on the numeric keypad
$6E	w.VK_DECIMAL	. on the numeric keypad
$6F	w.VK_DIVIDE	/ on the numeric keypad
$70	w.VK_F1	F1
$71	w.VK_F2	F2
$72	w.VK_F3	F3
$73	w.VK_F4	F4
$74	w.VK_F5	F5
$75	w.VK_F6	F6
$76	w.VK_F7	F7
$77	w.VK_F8	F8
$78	w.VK_F9	F9
$79	w.VK_F10	F10
$7A	w.VK_F11	F11
$7B	w.VK_F12	F12
$7C	w.VK_F13	F13
$7D	w.VK_F14	F14

Value	windows.hhf Constant	Description
$7E	w.VK_F15	F15
$7F	w.VK_F16	F16
$80	w.VK_F17	F17
$81	w.VK_F18	F18
$82	w.VK_F19	F19
$83	w.VK_F20	F20
$84	w.VK_F21	F21
$85	w.VK_F22	F22
$86	w.VK_F23	F23
$87	w.VK_F24	F24
$90	w.VK_NUMLOCK	Numlock key
$91	w.VK_SCROLL	Scroll lock key
$A0	w.VK_LSHIFT	Left shift key
$A1	w.VK_RSHIFT	Right shift key
$A2	w.VK_LCONTROL	Left control key
$A3	w.VK_RCONTROL	Right control key
$A4	w.VK_LMENU	Left alt key
$A5	w.VK_RMENU	Right alt key
$BA	w.VK_SEMICOLON	“;:” key (US keyboard)
$BB	w.VK_EQUAL	“=+” key (US keyboard)
$BC	w.VK_COMMA	“,<“ key (US keyboard)
$BD	w.VK_MINUS	“-_” key (US keyboard)
$BE	w.VK_PERIOD	“.>” key (US keyboard)
$BF	w.VK_SLASH	“/?” key (US keyboard)
$C0	w.VK_GRAVEACCENT	“‘~” key (US keyboard)
$DB	w.VK_LBRACKET	“[{“ key (US keyboard)
$DC	w.VK_BSLASH	“|” key (US keyboard)
$DD	w.VK_RBRACKET	“]}” key (US keyboard)
$DE	w.VK_APOST	“‘”” key (US keyboard)

The first pair of messages to look at are the `w.WM_KEYDOWN` and `w.WM_KEYUP` messages. Windows sends these two messages whenever the user presses or releases an *application* key (versus a *system* key), respectively. The `wParam` field of the message payload (i.e., the `wParam` parameter in the window procedure call) specifies the Windows virtual keycode for the message (see Table 8-1). The `lParam` parameter contains the information found in Table 8-2.

Table 8-2: lParam Data in a w.WM_KEYDOWN or w.WM_KEYUP Message

Bit Positions	Description
0..15	Repeat Count. This field specifies the number of keyboard messages (all the same) received for the current virtual keycode. Note that there is no guarantee that you won't see two consecutive keyboard messages with the same information (rather than merging them together into one message with an incremented repeat count). Windows uses this repeat count to minimize messages in the message queue during long auto-repeat operations while the application isn't reading messages. Note that the repeat count only applies to w.WM_KEYDOWN messages. The repeat count is always one for w.WM_KEYUP messages.
16-23	The is the OEM keyboard scan code. You should probably ignore this value as it may vary for keyboards from different manufacturers.
24	Specifies whether the key is an extended key (e.g., the right CTRL or ALT key). This bit contains one if the message is due to an extended key press or release.
25-28	Unused. Ignore these bits.
29	Context code. This code specifies whether the ALT key was pressed when this keyboard message was sent. This bit is always zero for up-key events.
30	This bit specifies the previous key state. It contains zero if the key was previously up and one if the key was previously down. Note that this bit will always contain one for w.WM_KEYUP messages, though it may contain zero or one for w.WM_KEYDOWN messages (because of the possibility of autorepeat).
31	This bit contains one if there was a down-to-up or up-to-down transition.

The `w.WM_KEYDOWN` and `w.WM_KEYUP` messages are useful for determining when the user presses a function key, cursor control key, or other special (non-ASCII) key on the keyboard. In fact, Windows will send these messages for every key you press on the keyboard if you are not holding the ALT key down simultaneously. If you are holding down the ALT key when you press a key on the keyboard, Windows will actually send `w.WM_SYSKEYDOWN` and `w.WM_SYSKEYUP` messages to your application. In general, however, your application should simply ignore these messages and let the default message handler process them. Windows will convert system keyboard messages into other message types and may pass those new messages on to your application for further processing.

To demonstrate how you would use the w.WM_KEYDOWN and w.WM_KEYUP messages in your applications to process virtual scan codes, we ll modify the *System Metrics* application that originally appeared in the chapter on Windows text processing. This *kbSysmet* program extends the *sysmet* program by handling w.WM_KEYDOWN messages and translating certain cursor control keys (the arrow keys, page up, page down, home, and end) into messages that will cause the application to scroll the window in appropriate directions. The following is a typical case in a switch statement:

```
        // If they press the "HOME" key, scroll to the top of the window.
        // Do this by sending a w.WM_VSCROLL message to do the scrolling
        // routines to reposition the window to the beginning.

        case( w.VK_HOME )

            w.SendMessage( hwnd, w.WM_VSCROLL, w.SB_TOP, 0 );
```

Other cases handle the other cursor control operations by translating the key press message into a corresponding mouse event on the scroll bar.

```
// kbSysmet.hla-
//
//  System metrics display program that supports keyboard messages.

unit kbSysmets;

// Set the following to true to display interesting information
// during program operation. You must be running
// the "DebugWindow" application for this output to appear.

?debug := false;

#includeonce( "excepts.hhf" )
#includeonce( "conv.hhf" )
#includeonce( "hll.hhf" )
#includeonce( "memory.hhf" )
#includeonce( "w.hhf" )
#includeonce( "wpa.hhf" )
#includeonce( "winmain.hhf" )

?@NoDisplay := true;
?@NoStackAlign := true;

type
    // Data type for the system metrics data array:

    MetricRec_t:
        record

            MetConst    :uns32;
            MetStr      :string;
            MetDesc     :string;

        endrecord;
```

```
static
    AverageCapsWidth    :dword;         // Font metric values.
    AverageCharWidth    :dword;
    AverageCharHeight   :dword;

    ClientSizeX         :int32 := 0;    // Size of the client area
    ClientSizeY         :int32 := 0;    //  where we can paint.
    MaxWidth            :int32 := 0;    // Maximum output width
    VscrollPos          :int32 := 0;    // Tracks where we are in the document
    VscrollMax          :int32 := 0;    // Max display position (vertical).
    HscrollPos          :int32 := 0;    // Current Horz position.
    HscrollMax          :int32 := 0;    // Max Horz position.

readonly

    ClassName   :string := "kbSysmetsWinClass";     // Window Class Name
    AppCaption  :string := "kbSysmets Program";     // Caption for Window


    // The dispatch table:
    //
    //  This table is where you add new messages and message handlers
    //  to the program.  Each entry in the table must be a MsgProcPtr_t
    //  record containing two entries: the message value (a constant,
    //  typically one of the w.WM_***** constants found in windows.hhf)
    //  and a pointer to a "MsgProcPtr_t" procedure that will handle the
    //  message.


    Dispatch    :MsgProcPtr_t; @nostorage;

        MsgProcPtr_t
            MsgProcPtr_t:[  w.WM_DESTROY, &QuitApplication   ],
            MsgProcPtr_t:[  w.WM_PAINT,   &Paint             ],
            MsgProcPtr_t:[  w.WM_CREATE,  &Create            ],
            MsgProcPtr_t:[  w.WM_HSCROLL, &HScroll           ],
            MsgProcPtr_t:[  w.WM_VSCROLL, &VScroll           ],
            MsgProcPtr_t:[  w.WM_SIZE,    &Size              ],
            MsgProcPtr_t:[  w.WM_KEYDOWN, &KeyDown           ],

            // Insert new message handler records here.

            MsgProcPtr_t:[  0, NULL ];   // This marks the end of the list.



readonly

    MetricData: MetricRec_t[ ] :=
    [
        MetricRec_t:[  w.SM_CXSCREEN, "w.SM_CXSCREEN", "Screen width" ],
        MetricRec_t:[  w.SM_CYSCREEN, "w.SM_CYSCREEN", "Screen height" ],
        MetricRec_t:[  w.SM_CXVSCROLL, "w.SM_CXVSCROLL", "Vert scroll arrow width" ],
        MetricRec_t:[  w.SM_CYVSCROLL, "w.SM_CYVSCROLL", "Vert scroll arrow ht" ],
        MetricRec_t:[  w.SM_CXHSCROLL, "w.SM_CXHSCROLL", "Horz scroll arrow width" ],
        MetricRec_t:[  w.SM_CYHSCROLL, "w.SM_CYHSCROLL", "Horz scroll arrow ht" ],
        MetricRec_t:[  w.SM_CYCAPTION, "w.SM_CYCAPTION", "Caption bar ht" ],
```

```
        MetricRec_t:[  w.SM_CXBORDER, "w.SM_CXBORDER", "Window border width" ],
        MetricRec_t:[  w.SM_CYBORDER, "w.SM_CYBORDER", "Window border height" ],
        MetricRec_t:[  w.SM_CXDLGFRAME, "w.SM_CXDLGFRAME", "Dialog frame width" ],
        MetricRec_t:[  w.SM_CYDLGFRAME, "w.SM_CYDLGFRAME", "Dialog frame height" ],
        MetricRec_t:[  w.SM_CXHTHUMB, "w.SM_CXHTHUMB", "Horz scroll thumb width" ],
        MetricRec_t:[  w.SM_CYVTHUMB, "w.SM_CYVTHUMB", "Vert scroll thumb width" ],
        MetricRec_t:[  w.SM_CXICON, "w.SM_CXICON", "Icon width" ],
        MetricRec_t:[  w.SM_CYICON, "w.SM_CYICON", "Icon height" ],
        MetricRec_t:[  w.SM_CXCURSOR, "w.SM_CXCURSOR", "Cursor width" ],
        MetricRec_t:[  w.SM_CYCURSOR, "w.SM_CYCURSOR", "Cursor height" ],
        MetricRec_t:[  w.SM_CYMENU,  "w.SM_CYMENU", "Menu bar height" ],
        MetricRec_t:[  w.SM_CXFULLSCREEN, "w.SM_CXFULLSCREEN", "Largest client width" ],
        MetricRec_t:[  w.SM_CYFULLSCREEN, "w.SM_CYFULLSCREEN", "Largets client ht" ],
        MetricRec_t:[  w.SM_DEBUG, "w.SM_CDEBUG", "Debug version flag" ],
        MetricRec_t:[  w.SM_SWAPBUTTON, "w.SM_CSWAPBUTTON", "Mouse buttons swapped" ],
        MetricRec_t:[  w.SM_CXMIN, "w.SM_CXMIN", "Minimum window width" ],
        MetricRec_t:[  w.SM_CYMIN, "w.SM_CYMIN", "Minimum window height" ],
        MetricRec_t:[  w.SM_CXSIZE, "w.SM_CXSIZE", "Minimize/maximize icon width" ],
        MetricRec_t:[  w.SM_CYSIZE, "w.SM_CYSIZE", "Minimize/maximize icon height" ],
        MetricRec_t:[  w.SM_CXFRAME, "w.SM_CXFRAME", "Window frame width" ],
        MetricRec_t:[  w.SM_CYFRAME, "w.SM_CYFRAME", "Window frame height" ],
        MetricRec_t:[  w.SM_CXMINTRACK,  "w.SM_CXMINTRACK", "Minimum tracking width" ],
        MetricRec_t:[  w.SM_CXMAXTRACK,  "w.SM_CXMAXTRACK", "Maximum tracking width" ],
        MetricRec_t:[  w.SM_CYMINTRACK,  "w.SM_CYMINTRACK", "Minimum tracking ht" ],
        MetricRec_t:[  w.SM_CYMAXTRACK,  "w.SM_CYMAXTRACK", "Maximum tracking ht" ],
        MetricRec_t:[  w.SM_CXDOUBLECLK, "w.SM_CXDOUBLECLK", "Dbl-click X tolerance" ],
        MetricRec_t:[  w.SM_CYDOUBLECLK, "w.SM_CYDOUBLECLK", "Dbl-click Y tolerance" ],
        MetricRec_t:[  w.SM_CXICONSPACING, "w.SM_CXICONSPACING", "Horz icon spacing" ],
        MetricRec_t:[  w.SM_CYICONSPACING, "w.SM_CYICONSPACING", "Vert icon spacing" ],
        MetricRec_t:[  w.SM_CMOUSEBUTTONS, "w.SM_CMOUSEBUTTONS", " # of mouse btns" ]
    ];

const
    NumMetrics := @elements( MetricData );



/*****************************************************************************/
/*          W I N M A I N   S U P P O R T   C O D E                          */
/*****************************************************************************/

// initWC - We don't have any initialization to do, so just return:

procedure initWC; @noframe;
begin initWC;

    dbg.put( hwnd, nl "Bitmaps3----------------", nl );
    ret();

end initWC;

// appCreateWindow- the default window creation code is fine, so just
//                  call defaultCreateWindow.

procedure appCreateWindow; @noframe;
begin appCreateWindow;
```

```
    jmp defaultCreateWindow;

end appCreateWindow;


// appException-
//
// Gives the application the opportunity to clean up before
// aborting when an unhandled exception comes along:

procedure appException( theException:dword in eax );
begin appException;

    raise( eax );

end appException;




// This is the custom message translation procedure.
// We're not doing any custom translation, so just return EAX=0
// to tell the caller to go ahead and call the default translation
// code.

procedure LocalProcessMsg( var lpmsg:w.MSG );
begin LocalProcessMsg;

    xor( eax, eax );

end LocalProcessMsg;


/**************************************************************************/
/*          A P P L I C A T I O N   S P E C I F I C   C O D E             */
/**************************************************************************/


// QuitApplication:
//
//  This procedure handles the w.WM_DESTROY message.
//  It tells the application to terminate.  This code sends
//  the appropriate message to the main program's message loop
//  that will cause the application to terminate.

procedure QuitApplication( hwnd: dword; wParam:dword; lParam:dword );
begin QuitApplication;

    w.PostQuitMessage( 0 );

end QuitApplication;


// Create-
//
//  This procedure responds to the w.WM_CREATE message.
//  Windows sends this message once when it creates the
```

```
//  main window for the application.  We will use this
//  procedure to do any one-time initialization that
//  must take place in a message handler.

procedure Create( hwnd: dword; wParam:dword; lParam:dword );
var
    hdc:    dword;              // Handle to video display device context
    tm:     w.TEXTMETRIC;
begin Create;

    GetDC( hwnd, hdc );

        // Initialization:
        //
        //  Get the text metric information so we can compute
        //  the average character heights and widths.

        GetTextMetrics( tm );

        mov( tm.tmHeight, eax );
        add( tm.tmExternalLeading, eax );
        mov( eax, AverageCharHeight );

        mov( tm.tmAveCharWidth, eax );
        mov( eax, AverageCharWidth );

        // If bit #0 of tm.tmPitchAndFamily is set, then
        // we've got a proportional font.  In that case
        // set the average capital width value to 1.5 times
        // the average character width.  If bit #0 is clear,
        // then we've got a fixed-pitch font and the average
        // capital letter width is equal to the average
        // character width.

        mov( eax, ebx );
        shl( 1, tm.tmPitchAndFamily );
        if( @c ) then

            shl( 1, ebx );              // 2*AverageCharWidth

        endif;
        add( ebx, eax );                // Computes 2 or 3 times eax.
        shr( 1, eax );                  // Computes 1 or 1.5 times eax.
        mov( eax, AverageCapsWidth );

    ReleaseDC;
    intmul( 40, AverageCharWidth, eax );
    intmul( 25, AverageCapsWidth, ecx );
    add( ecx, eax );
    mov( eax, MaxWidth );

end Create;


// Paint:
//
//  This procedure handles the w.WM_PAINT message.
```

```
//  For this System Metrics program, the Paint procedure
//  displays three columns of text in the main window.
//  This procedure computes and displays the appropriate text.

procedure Paint( hwnd: dword; wParam:dword; lParam:dword );
var
    x           :int32;         // x-coordinate of start of output str.
    y           :int32;         // y-coordinate of start of output str.

    CurVar      :string;        // Current system metrics variable name.
    CVlen       :uns32;         // Length of CurVar string.

    CurDesc     :string;        // Current system metrics description.
    CDlen       :string;        // Length of the above.
    CDx         :int32;         // X position for CurDesc string.

    value       :string;
    valData     :char[ 32];
    CVx         :int32;         // X position for value string.
    vallen      :uns32;         // Length of value string.

    firstMet    :int32;         // Starting metric to begin drawing
    lastMet     :int32;         // Ending metric index to draw.

    hdc         :dword;         // Handle to video display device context
    ps          :w.PAINTSTRUCT; // Used while painting text.

begin Paint;

    // Message handlers must preserve EBX, ESI, and EDI.
    // (They've also got to preserve EBP, but HLA's procedure
    // entry code already does that.)

    push( ebx );
    push( esi );
    push( edi );


    // Initialize the value->valData string object:

    mov( str.init( (type char valData), 32 ), value );

    // When Windows requests that we draw the window,
    // fill in the string in the center of the screen.
    // Note that all GDI calls (e.g., w.DrawText) must
    // appear within a BeginPaint..EndPaint pair.

    BeginPaint( hwnd, ps, hdc );

        // Figure out which metric we should start drawing
        // (firstMet =
        //      max( 0, VscrollPos + ps.rcPaint.top/AverageCharHeight - 1)):

        mov( ps.rcPaint.top, eax );
        cdq();
        idiv( AverageCharHeight );
        add( VscrollPos, eax );
```

```
dec( eax );
if( (type int32 eax) < 0 ) then

    xor( eax, eax );

endif;
mov( eax, firstMet );


// Figure out the last metric we should be drawing
// ( lastMet =
//      min( NumMetrics,
//           VscrollPos + ps.rcPaint.bottom/AverageCharHeight )):

mov( ps.rcPaint.bottom, eax );
cdq();
idiv( AverageCharHeight );
add( VscrollPos, eax );
if( (type int32 eax) > NumMetrics ) then

    mov( NumMetrics, eax );

endif;
mov( eax, lastMet );


// The following loop processes each entry in the
// MetricData array.  The loop control variable (EDI)
// also determines the Y-coordinate where this code
// will display each line of text in the window.
// Note that this loop counts on the fact that Windows
// API calls preserve the EDI register.

for( mov( firstMet, edi ); edi < lastMet; inc( edi )) do

    // Before making any Windows API calls (which have
    // a nasty habit of wiping out registers), compute
    // all the values we will need for these calls
    // and save those values in local variables.
    //
    //  A typical "high level language solution" would
    // be to compute these values as needed, immediately
    // before each Windows API calls.  By moving this
    // code here, we can take advantage of values previously
    // computed in registers without having to worry about
    // Windows wiping out the values in those registers.

    // Compute index into MetricData:

    intmul( @size( MetricRec_t ), edi, esi );

    // Grab the string from the current MetricData element:

    mov( MetricData.MetStr[ esi ], eax );
    mov( eax, CurVar );
    mov( (type str.strRec [ eax]).length, eax );
    mov( eax, CVlen );
```

```
mov( MetricData.MetDesc[  esi ], eax );
mov( eax, CurDesc );
mov( (type str.strRec [ eax]).length, eax );
mov( eax, CDlen );

// Column one begins at X-position AverageCharWidth (ACW).
// Col 2 begins at ACW + 25*AverageCapsWidth.
// Col 3 begins at ACW + 25*AverageCapsWidth + 40*ACW.
// Compute the Col 2 and Col 3 values here.

mov( 1, eax );
sub( HscrollPos, eax );
intmul( AverageCharWidth, eax );
mov( eax, x );

intmul( 25, AverageCapsWidth, eax );
add( x, eax );
mov( eax, CDx );

intmul( 40, AverageCharWidth, ecx );
add( ecx, eax );
mov( eax, CVx );

// The Y-coordinate for the line of text we're writing
// is computed as AverageCharHeight * (1-VscrollPos+edi).
// Compute that value here:

mov( 1, eax );
sub( VscrollPos, eax );
add( edi, eax );
intmul( AverageCharHeight, eax );
mov( eax, y );


// Now generate the string we're going to print
// as the value for the current metric variable:

w.GetSystemMetrics( MetricData.MetConst[  esi ] );
conv.i32ToStr( eax, 0, ' ', value );
mov( str.length( value ), vallen );


// First two columns have left-aligned text:

SetTextAlign( w.TA_LEFT | w.TA_TOP );

// Output the name of the metric variable:

TextOut( x, y, CurVar, CVlen );

// Output the description of the metric variable:

TextOut( CDx, y, CurDesc, CDlen );

// Output the metric's value in the third column.  This is
// a numeric value, so we'll right align this data.
```

```
            SetTextAlign( w.TA_RIGHT | w.TA_TOP );
            TextOut( CVx, y, value, vallen );

            // Although not strictly necessary for this program,
            // it's a good idea to always restore the alignment
            // back to the default (top/left) after you done using
            // some other alignment.

            SetTextAlign( w.TA_LEFT | w.TA_TOP );


        endfor;


    EndPaint;

    pop( edi );
    pop( esi );
    pop( ebx );

end Paint;



// Size-
//
//  This procedure handles the w.WM_SIZE message
//
//  L.O. word of lParam contains the new X Size
//  H.O. word of lParam contains the new Y Size

procedure Size( hwnd: dword; wParam:dword; lParam:dword );
begin Size;

    // Convert new X size to 32 bits and save:

    movzx( (type word lParam), eax );
    mov( eax, ClientSizeX );

    // Convert new Y size to 32 bits and save:

    movzx( (type word lParam[2]), eax );
    mov( eax, ClientSizeY );

    // VscrollMax = max( 0, NumMetrics+2 - ClientSizeY/AverageCharHeight )

    cdq();
    idiv( AverageCharHeight );
    mov( NumMetrics+2, ecx );
    sub( eax, ecx );
    if( @s ) then

        xor( ecx, ecx );

    endif;
    mov( ecx, VscrollMax );
```

```
    // VscrollPos = min( VscrollPos, VscrollMax )

    if( ecx > VscrollPos ) then

        mov( VscrollPos, ecx );

    endif;
    mov( ecx, VscrollPos );

    w.SetScrollRange( hwnd, w.SB_VERT, 0, VscrollMax, false );
    w.SetScrollPos( hwnd, w.SB_VERT, VscrollPos, true );

    // HscrollMax =
    //  max( 0, 2 + (MaxWidth - ClientSizeX) / AverageCharWidth);

    mov( MaxWidth, eax );
    sub( ClientSizeX, eax );
    cdq();
    idiv( AverageCharWidth );
    add( 2, eax );
    if( @s ) then

        xor( eax, eax );

    endif;
    mov( eax, HscrollMax );

    // HscrollPos = min( HscrollMax, HscrollPos )

    if( eax > HscrollPos ) then

        mov( HscrollPos, eax );

    endif;
    mov( eax, HscrollPos );
    w.SetScrollRange( hwnd, w.SB_HORZ, 0, HscrollMax, false );
    w.SetScrollPos( hwnd, w.SB_HORZ, HscrollPos, true );
    xor( eax, eax ); // return success.


end Size;



// HScroll-
//
//  Handles w.WM_HSCROLL messages.
//  On entry, L.O. word of wParam contains the scroll bar activity.

procedure HScroll( hwnd: dword; wParam:dword; lParam:dword );
begin HScroll;

    // Convert 16-bit command to 32 bits so we can use switch macro:

    movzx( (type word wParam), eax );
    switch( eax )
```

```
        case( w.SB_LINELEFT )

            mov( -1, eax );

        case( w.SB_LINERIGHT )

            mov( 1, eax );

        case( w.SB_PAGELEFT )

            mov( -8, eax );

        case( w.SB_PAGERIGHT )

            mov( 8, eax );

        case( w.SB_THUMBPOSITION )

            movzx( (type word wParam[2]), eax );
            sub( HscrollPos, eax );

        default

            xor( eax, eax );

    endswitch;

    // eax =
    //  max( -HscrollPos, min( eax, HscrollMax - HscrollPos ))

    mov( HscrollPos, edx );
    neg( edx );
    mov( HscrollMax, ecx );
    add( edx, ecx );
    if( eax > (type int32 ecx) ) then

        mov( ecx, eax );

    endif;
    if( eax < (type int32 edx )) then

        mov( edx, eax );

    endif;
    if( eax <> 0 ) then

        add( eax, HscrollPos );
        imul( AverageCharWidth, eax );
        neg( eax );
        w.ScrollWindow( hwnd, eax, 0, NULL, NULL );
        w.SetScrollPos( hwnd, w.SB_HORZ, HscrollPos, true );

    endif;
    xor( eax, eax ); // return success

end HScroll;
```

```
// VScroll-
//
//  Handles the w.WM_VSCROLL messages from Windows.
//  The L.O. word of wParam contains the action/command to be taken.
//  The H.O. word of wParam contains a distance for the w.SB_THUMBTRACK
//  message.

procedure VScroll( hwnd: dword; wParam:dword; lParam:dword );
begin VScroll;

    movzx( (type word wParam), eax );
    switch( eax )

        case( w.SB_TOP )

            mov( VscrollPos, eax );
            neg( eax );

        case( w.SB_BOTTOM )

            mov( VscrollMax, eax );
            sub( VscrollPos, eax );

        case( w.SB_LINEUP )

            mov( -1, eax );

        case( w.SB_LINEDOWN )

            mov( 1, eax );

        case( w.SB_PAGEUP )

            mov( ClientSizeY, eax );
            cdq();
            idiv( AverageCharHeight );
            neg( eax );
            if( (type int32 eax) > -1 ) then

                mov( -1, eax );

            endif;

        case( w.SB_PAGEDOWN )

            mov( ClientSizeY, eax );
            cdq();
            idiv( AverageCharHeight );
            if( (type int32 eax) < 1 ) then

                mov( 1, eax );

            endif;
```

```
        case( w.SB_THUMBTRACK )

            movzx( (type word wParam[2]), eax );
            sub( VscrollPos, eax );

        default

            xor( eax, eax );

    endswitch;

    // eax = max( -VscrollPos, min( eax, VscrollMax - VscrollPos ))

    mov( VscrollPos, edx );
    neg( edx );
    mov( VscrollMax, ecx );
    add( edx, ecx );
    if( eax > (type int32 ecx) ) then

        mov( ecx, eax );

    endif;
    if( eax < (type int32 edx)) then

        mov( edx, eax );

    endif;

    if( eax <> 0 ) then

        add( eax, VscrollPos );
        intmul( AverageCharHeight, eax );
        neg( eax );
        w.ScrollWindow( hwnd, 0, eax, NULL, NULL );
        w.SetScrollPos( hwnd, w.SB_VERT, VscrollPos, true );
        w.UpdateWindow( hwnd );

    endif;
    xor( eax, eax ); // return success.


end VScroll;


// KeyDown-
//
//  Handles the w.WM_KEYDOWN messages from Windows.
//  The L.O. word of wParam contains the action/command to be taken.
//  The H.O. word of wParam contains a distance for the w.SB_THUMBTRACK
//  message.

procedure KeyDown( hwnd: dword; wParam:dword; lParam:dword );
begin KeyDown;

    mov( wParam, eax );
    switch( eax )
```

```
        // If they press the "HOME" key, scroll to the top of the window:

        case( w.VK_HOME )

            w.SendMessage( hwnd, w.WM_VSCROLL, w.SB_TOP, 0 );

        // If they press the "END" key, scroll to the bottom of the window:

        case( w.VK_END )

            w.SendMessage( hwnd, w.WM_VSCROLL, w.SB_BOTTOM, 0 );

        // If they press the "PgUp" key, scroll up one page:

        case( w.VK_PRIOR )

            w.SendMessage( hwnd, w.WM_VSCROLL, w.SB_PAGEUP, 0 );

        // If they press the "PgDn" key, scroll down one page:

        case( w.VK_NEXT )

            w.SendMessage( hwnd, w.WM_VSCROLL, w.SB_PAGEDOWN, 0 );

        // If they press the "Up" key, scroll up one line:

        case( w.VK_UP )

            w.SendMessage( hwnd, w.WM_VSCROLL, w.SB_LINEUP, 0 );

        // If they press the "Down" key, scroll down one line:

        case( w.VK_DOWN )

            w.SendMessage( hwnd, w.WM_VSCROLL, w.SB_LINEDOWN, 0 );

        // If they press the "Left" key, scroll text to the right:

        case( w.VK_LEFT )

            w.SendMessage( hwnd, w.WM_HSCROLL, w.SB_PAGEUP, 0 );

        // If they press the "Right" key, scroll text to the left:

        case( w.VK_RIGHT )

            w.SendMessage( hwnd, w.WM_HSCROLL, w.SB_PAGEDOWN, 0 );

    endswitch;

end KeyDown;

end kbSysmets;
```

The *kbSysmets* program demonstrates how to process raw virtual keycodes in your program. Most of the time, however, you re not interested in the virtual keycodes; what you really want are ASCII codes for whatever keys the user presses. The only time you ll really want to deal with virtual key codes is when reading keystrokes from the user that have no corresponding ASCII codes.

In theory, you could convert virtual keycodes into ASCII codes yourself. The catch is that you must maintain certain state information, such as which modifier keys (shift, control, alt, capslock, numlock, etc.) are currently active and use this information to translate a virtual key code into the corresponding ASCII code. For example, if you get the virtual keycode $41 (A), you would have to translate this code to $61 (a), $41 (A), or $01 (ctrl-A), depending on the state of the shift and control keys (note that Windows treats the alt modifier key specially, you wouldn t normally have to deal with this). There are a couple of problems with this translation - first of all it s a lot of work. Second, and more important, there is no single translation you can do. Different Windows systems in different countries do the translation differently. Trying to handle the translation of virtual key codes for the dozens of different keyboards that exist today would be overwhelming. Fortunately, you don t have to do this translation yourself (nor should you attempt it): Windows will automatically do the translation for you. Consider the main message processing loop in the *Winmain.hla* module:

```
        forever

            w.GetMessage( msg, NULL, 0, 0 );
            breakif( eax == 0 );
            if( LocalProcessMsg( msg ) == 0) then

                w.TranslateMessage( msg );

            endif;
            w.DispatchMessage( msg );

        endfor;
```

The `w.TranslateMessage` API call intercepts `w.WM_KEYDOWN` and `w.WM_KEYUP` messages and generates a new message to send to your application if the key message corresponds to some ASCII character. Note that upon return, the message processing loop still sends the `w.WM_KEYDOWN` or `w.WM_KEYUP` message to your application; the new message that `w.TranslateMessage` creates will follow shortly. Therefore, when the w.TranslateMessage function does a translation, your application will actually receive two messages.

Before looking at these new message types, let s first discuss the `LocalProcessMsg` function call that appears in the loop above. This is not a Windows API function call (note the lack of a w. prefix). Instead, this is a call to an application-supplied function that determines whether any local message processing should take place. `LocalProcessMsg` returns zero/not zero in EAX to determine whether the main message processing loop should call the `w.TranslateMessage` API function (EAX is a skip translation flag, zero indicates that translation should take place and non-zero means to skip the translation operation). The vast majority of the time, you ll want to call the Windows `w.TranslateMessage` API function and not mess around with the message. Therefore, the typical default `LocalProcessMessage` function looks like this:

```
procedure LocalProcessMsg( var lpmsg:w.MSG );
begin LocalProcessMsg;

    xor( eax, eax );

end LocalProcessMsg;
```

That is, the function simply returns with EAX zero so that the main message processing loop will call the w.TranslateMessage API function. The single parameter is a pointer to a message structure, this object takes the following form:

```
type
    MSG:
        record
            hwnd:    dword;
            message: dword;
            wParam:  dword;
            lParam:  dword;
            time:    dword;
            pt:      POINT;
        endrecord;
```

The three fields of particular interest are the message, wParam, and lParam fields. These fields contain the values that Windows will pass as parameters to your Window procedure (i.e., the parameters that wind up being passed to your message handling procedures in your application). You can peek at the message field s valueto determine if you want to do any special processing on the message. If so, you can fetch other values from the wParam and lParam fields. Any changes you make to the fields of this structure will be passed along to w.TranslateMessage (if you set EAX to zero before returning) and to the w.DispatchMessage API function (that winds up calling your window procedure). Therefore, you can do some sophisticated translation of your own, should you choose to do so, within the LocalProcessMsg function. The vast majority of the time, however, you ll not bother with doing any translations inside this code.

If the message processing loop sends w.TranslateMessage a w.WM_KEYDOWN or w.WM_KEYUP message, then the w.TranslateMessage function may inject a new message into the message queue containing an ASCII translation of the keypress. This new message is a w.WM_CHAR message. The w.WM_CHAR message contains the same information in the lParam field as the w.WM_KEYDOWN and w.WM_KEYUP messages, the wParam field contains an ASCII key code rather than a Windows virtual key code.

Whenever you press (and release) a key that has a corresponding ASCII key code, Windows will actually send three messages to your application: a w.WM_KEYDOWN message, a w.WM_CHAR message, and then a w.WM_KEYUP message, in that order. Generally, you will ignore the keydown and key up messages and process only the w.WM_CHAR messages within your application (in fact, most applications also ignore all w.WM_KEYUP messages, as well).

Windows also sends a couple of other keyboard-related messages to your applications. These messages are w.WM_DEADCHAR, w.WM_SYSCHAR, and w.WM_SYSDEADCHAR. The ...SYS... messages correspond to system keystrokes. You application can safely ignore these messages. Windows sends a w.WM_DEADCHAR message whenever you press an accent key prefix key on the keyboard that will produce an accented character (e.g., on non-U.S. keyboards). You can usually ignore all dead character messages as the following w.WM_CHAR message will incorporate the dead key information. The only reason for looking at these messages is to create your own, special, accented characters that Windows doesn t normally support. As such a need is rare, we won t consider the dead codes here any farther.

The following application, *keytest.hla*, is another application inspired by a program in Petzold s book. This program processes the keyboard messages and writes their data payloads to the application s window. This short application lets you view all the keyboard messages that come along whenever you press a key on the PC s keyboard.

```
// keytest.hla-
//
```

```
//  This program reads keystroke messages from the system and displays them.

unit keytest;

// Set the following to true to display interesting information
// during program operation. You must be running
// the "DebugWindow" application for this output to appear.

?debug := false;

#includeonce( "excepts.hhf" )
#includeonce( "conv.hhf" )
#includeonce( "hll.hhf" )
#includeonce( "memory.hhf" )
#includeonce( "w.hhf" )
#includeonce( "wpa.hhf" )
#includeonce( "winmain.hhf" )

?@NoDisplay := true;
?@NoStackAlign := true;

type
    // Data type for the keyboard message:

    keymsgPtr_t     :pointer to keymsg_t;
    keymsg_t:
        record

            // Maintains list of records:

            Next        :keymsgPtr_t;

            // MsgStr points at a string specifying the
            // message type: "WM_CHAR", "WM_KEYDOWN", etc:

            MsgStr      :string;

            // Virtual key code in WM_KEYDOWN/WM_KEYUP
            // messages:

            VirtKey     :dword;

            // Repeat count in message (# of autorepeated
            // keys passed on this message):

            RepeatCnt   :uns16;

            // ASCII code from WM_CHAR message:

            ASCIIcode   :byte;

            // OEM scan code in messages:

            ScanCode    :byte;

            // 1 indicates down->up transition, 0 indicates
            // an up->down transition:
```

```
Transition  :uns8;

// 1 indicates key previously down, 0 indicates
// key was previously up:

PrevKeyState:uns8;

// 1 indicates that ALT was down when key
// was pressed:

Context     :uns8;

// 1 indicates extended (non-keypad) cursor control
// key was pressed:

ExtendedKey :uns8;

// 1 indicates left shift was down:

LshiftState :uns8;

// 1 indicates right shift was down:

RshiftState :uns8;

// 1 indicates that the left control
// key was down:

LctrlState  :uns8;

// 1 indicates that the right control
// key was down:

RctrlState  :uns8;

// 1 indicates that the left ALT key
// was down:

LaltState   :uns8;

// 1 indicates that the right ALT key
// was down:

RaltState   :uns8;

// 1 indicates that the keyboard is in the
// "capslock" mode (caps are on):

CapsLock    :uns8;

// 1 indicates that the keyboard is in the
// "numlock" mode:

NumLock     :uns8;

// 1 indicates that the keyboard is in the
```

```
            // "scroll lock" mode:

            ScrlLock    :uns8;

        endrecord;


static
    KeyPressList        :keymsgPtr_t := NULL;   // Ptr to list of key events.
    EndKeyPressList     :keymsgPtr_t := NULL;   // Ptr to last entry
    KeyPressCnt         :uns32 := 0;            // # of key event entries.

    TheFont             :dword;         // Font we'll use.

    AverageCapsWidth    :dword;         // Font metric values.
    AverageCharWidth    :dword;
    AverageCharHeight   :dword;

    ClientSizeX         :int32 := 0;    // Size of the client area
    ClientSizeY         :int32 := 0;    //  where we can paint.
    VscrollPos          :int32 := 0;    // Tracks where we are in the document
    VscrollMax          :int32 := 0;    // Max display position (vertical).
    HscrollPos          :int32 := 0;    // Current Horz position.
    HscrollMax          :int32 := 0;    // Max Horz position.
    MaxWidth            :int32 := 0;    // Max # of chars on a line.

readonly

    ClassName   :string := "keytestWinClass";       // Window Class Name
    AppCaption  :string := "keytest Program";       // Caption for Window


    // The dispatch table:
    //
    //  This table is where you add new messages and message handlers
    //  to the program.  Each entry in the table must be a MsgProcPtr_t
    //  record containing two entries: the message value (a constant,
    //  typically one of the w.WM_***** constants found in windows.hhf)
    //  and a pointer to a "MsgProcPtr_t" procedure that will handle the
    //  message.


    Dispatch    :MsgProcPtr_t; @nostorage;

        MsgProcPtr_t
            MsgProcPtr_t:[  w.WM_DESTROY,    &QuitApplication    ],
            MsgProcPtr_t:[  w.WM_PAINT,      &Paint              ],
            MsgProcPtr_t:[  w.WM_CREATE,     &Create             ],
            MsgProcPtr_t:[  w.WM_HSCROLL,    &HScroll            ],
            MsgProcPtr_t:[  w.WM_VSCROLL,    &VScroll            ],
            MsgProcPtr_t:[  w.WM_SIZE,       &Size               ],
            MsgProcPtr_t:[  w.WM_KEYDOWN,    &KeyDown            ],
            MsgProcPtr_t:[  w.WM_KEYUP,      &KeyUp              ],
            MsgProcPtr_t:[  w.WM_SYSKEYDOWN, &SysKeyDown         ],
            MsgProcPtr_t:[  w.WM_SYSKEYUP,   &SysKeyUp           ],
            MsgProcPtr_t:[  w.WM_CHAR,       &CharMsg            ],
            MsgProcPtr_t:[  w.WM_DEADCHAR,   &DeadCharMsg        ],
```

```
            MsgProcPtr_t:[  w.WM_SYSCHAR,    &SysCharMsg        ],
            MsgProcPtr_t:[  w.WM_SYSDEADCHAR,&SysDeadCharMsg    ],

            // Insert new message handler records here.

            MsgProcPtr_t:[  0, NULL ];   // This marks the end of the list.




/*****************************************************************************/
/*          W I N M A I N   S U P P O R T   C O D E                          */
/*****************************************************************************/

// initWC - We don't have any initialization to do, so just return:

procedure initWC; @noframe;
begin initWC;

    ret();

end initWC;

// appCreateWindow- the default window creation code is fine, so just
//                  call defaultCreateWindow.

procedure appCreateWindow; @noframe;
begin appCreateWindow;

    jmp defaultCreateWindow;

end appCreateWindow;


// appException-
//
// Gives the application the opportunity to clean up before
// aborting when an unhandled exception comes along:

procedure appException( theException:dword in eax );
begin appException;

    raise( eax );

end appException;



// This is the custom message translation procedure.
// We're not doing any custom translation, so just return EAX=0
// to tell the caller to go ahead and call the default translation
// code.

procedure LocalProcessMsg( var lpmsg:w.MSG );
begin LocalProcessMsg;
```

```
    xor( eax, eax );

end LocalProcessMsg;


/*****************************************************************************/
/*          A P P L I C A T I O N   S P E C I F I C   C O D E                */
/*****************************************************************************/


// QuitApplication:
//
//  This procedure handles the w.WM_DESTROY message.
//  It tells the application to terminate.  This code sends
//  the appropriate message to the main program's message loop
//  that will cause the application to terminate.

procedure QuitApplication( hwnd: dword; wParam:dword; lParam:dword );
begin QuitApplication;

    w.PostQuitMessage( 0 );

end QuitApplication;


// Create-
//
//  This procedure responds to the w.WM_CREATE message.
//  Windows sends this message once when it creates the
//  main window for the application.  We will use this
//  procedure to do any one-time initialization that
//  must take place in a message handler.

procedure Create( hwnd: dword; wParam:dword; lParam:dword );
var
    hdc:    dword;              // Handle to video display device context
    tm:     w.TEXTMETRIC;
begin Create;

    // First, create a useful font (fixed pitch):

    w.GetStockObject( w.SYSTEM_FIXED_FONT );
    mov( eax, TheFont );

    GetDC( hwnd, hdc );

        // Initialization:
        //
        //  Get the text metric information so we can compute
        //  the average character heights and widths.

        SelectObject( TheFont );

        GetTextMetrics( tm );

        mov( tm.tmHeight, eax );
```

```
        add( tm.tmExternalLeading, eax );
        mov( eax, AverageCharHeight );

        mov( tm.tmAveCharWidth, eax );
        mov( eax, AverageCharWidth );

        // If bit #0 of tm.tmPitchAndFamily is set, then
        // we've got a proportional font.  In that case
        // set the average capital width value to 1.5 times
        // the average character width.  If bit #0 is clear,
        // then we've got a fixed-pitch font and the average
        // capital letter width is equal to the average
        // character width.

        mov( eax, ebx );
        shl( 1, tm.tmPitchAndFamily );
        if( @c ) then

            shl( 1, ebx );              // 2*AverageCharWidth

        endif;
        add( ebx, eax );                // Computes 2 or 3 times eax.
        shr( 1, eax );                  // Computes 1 or 1.5 times eax.
        mov( eax, AverageCapsWidth );

    ReleaseDC;

end Create;



// Paint:
//
//  This procedure handles the w.WM_PAINT message.
//  For this keytest program, the Paint procedure
//  displays the list of key events we've saved in memory.

procedure Paint( hwnd: dword; wParam:dword; lParam:dword );
var
    value       :string;
    valData     :char[ 256] ;
    vallen      :uns32;         // Length of value string.

    hdc         :dword;         // Handle to video display device context
    ps          :w.PAINTSTRUCT; // Used while painting text.


begin Paint;

    // Message handlers must preserve EBX, ESI, and EDI.
    // (They've also got to preserve EBP, but HLA's procedure
    // entry code already does that.)

    push( ebx );
    push( esi );
    push( edi );
```

```
// Initialize the value->valData string object:

mov( str.init( (type char valData), 256 ), value );

// When Windows requests that we draw the window,
// fill in the string in the center of the screen.
// Note that all GDI calls (e.g., w.DrawText) must
// appear within a BeginPaint..EndPaint pair.

BeginPaint( hwnd, ps, hdc );

    // Select the fixed-pitch font into our context:

    SelectObject( TheFont );

    // Figure out which keypress record we should start drawing
    // Begin by computing the number of lines of text we can
    // can draw in the window. This is
    //
    //          (ClientSizeY/AverageCharHeight)
    //
    // The number of keyboard events we can display is the
    // the minimum of this value and the number of events
    // we've seen thus far:

    mov( ClientSizeY, eax );
    cdq();
    idiv( AverageCharHeight );
    if( eax > KeyPressCnt ) then

        mov( KeyPressCnt, eax );

    endif;
    mov( eax, esi );

    // VscrollPos specifies the starting event record number
    // we're supposed to display. Search for that entry in
    // the keyboard event list:

    mov( KeyPressList, ebx );
    for( mov( 1, edi ); edi < VscrollPos; inc( edi )) do

        mov( (type keymsg_t [ ebx]).Next, ebx );

    endfor;

    // Okay, draw all the records (esi currently holds the
    // number of records to draw):

    mov( 0, edi );  // Holds output y-coordinate.
    while( esi > 0 ) do

        // Create the output string for this line:

        str.put
        (
            value,
```

```
    (type keymsg_t [ebx]).MsgStr:-16,
    "VK=",
    (type byte (type keymsg_t [ebx]).VirtKey):2,
    " ASCII=",
    (type byte (type keymsg_t [ebx]).ASCIIcode):3,
    " SC=",
    (type byte (type keymsg_t [ebx]).ScanCode):3,
    " Trans=",
    (type keymsg_t [ebx]).Transition:2,
    " Prev=",
    (type keymsg_t [ebx]).PrevKeyState:2,
    " Context=",
    (type keymsg_t [ebx]).Context:2,
    " Ext=",
    (type keymsg_t [ebx]).ExtendedKey:2,
    " LS=",
    (type keymsg_t [ebx]).LshiftState:2,
    " RS=",
    (type keymsg_t [ebx]).RshiftState:2,
    " LC=",
    (type keymsg_t [ebx]).LctrlState:2,
    " RC=",
    (type keymsg_t [ebx]).RctrlState:2,
    " LA=",
    (type keymsg_t [ebx]).LaltState:2,
    " RA=",
    (type keymsg_t [ebx]).RaltState:2,
    " CL=",
    (type keymsg_t [ebx]).CapsLock:2,
    " NL=",
    (type keymsg_t [ebx]).NumLock:2,
    " SL=",
    (type keymsg_t [ebx]).ScrlLock:2
);

mov( value, eax );
mov( (type str.strRec [eax]).length, ecx );

// Activate the horizontal scroll bars if we emit a line
// that is wider than the current window:

if( ecx > MaxWidth ) then

    mov( ecx, MaxWidth );
    push( eax );
    mov( MaxWidth, eax );
    intmul( AverageCharWidth, eax );
    sub( ClientSizeX, eax );
    cdq();
    idiv( AverageCharWidth );
    add( 2, eax );
    if( @s ) then

        xor( eax, eax );

    endif;
    mov( eax, HscrollMax );
```

```
                w.SetScrollRange( hwnd, w.SB_HORZ, 0, HscrollMax, false );
                pop( eax );

            endif;

            // Emit the current line to the display:

            add( HscrollPos, eax );     // Add in offset to 1st char in line
            sub( HscrollPos, ecx );     // Decrease line length by like amount.
            TextOut
            (
                5,
                edi,
                eax,
                ecx
            );

            // Move on to the next line of text to display:

            add( AverageCharHeight, edi );
            dec( esi );
            mov( (type keymsg_t [ebx]).Next, ebx );

        endwhile;

    EndPaint;

    pop( edi );
    pop( esi );
    pop( ebx );

end Paint;



// Size-
//
//  This procedure handles the w.WM_SIZE message
//
//  L.O. word of lParam contains the new X Size
//  H.O. word of lParam contains the new Y Size

procedure Size( hwnd: dword; wParam:dword; lParam:dword );
begin Size;

    // Convert new X size to 32 bits and save:

    movzx( (type word lParam), eax );
    mov( eax, ClientSizeX );

    // Convert new Y size to 32 bits and save:

    movzx( (type word lParam[2]), eax );
    mov( eax, ClientSizeY );
```

```
// Compute new bounds values for VscrollMax and VscrollPos based
// on the new size of the window:
//
// VscrollMax = max( 0, KeyPressCnt - ClientSizeY/AverageCharHeight )

mov( ClientSizeY, eax );
cdq();
idiv( AverageCharHeight );
neg( eax );
add( KeyPressCnt, eax );
if( @s ) then

    xor( eax, eax );

endif;
mov( eax, VscrollMax );

// VscrollPos = min( VscrollPos, VscrollMax )

if( eax > VscrollPos ) then

    mov( VscrollPos, eax );

endif;
mov( eax, VscrollPos );

// Set the new scrolling range and position based on the
// new VscrollMax and VscrollPos values:

w.SetScrollRange( hwnd, w.SB_VERT, 0, VscrollMax, false );
w.SetScrollPos( hwnd, w.SB_VERT, VscrollPos, true );


// Repeat the above for the horizontal scroll items:
//
// HscrollMax =
//  max( 0, 2 + (MaxWidth - ClientSizeX) / AverageCharWidth);


mov( MaxWidth, eax );
intmul( AverageCharWidth, eax );
sub( ClientSizeX, eax );
cdq();
idiv( AverageCharWidth );
add( 2, eax );
if( @s ) then

    xor( eax, eax );

endif;
mov( eax, HscrollMax );

// HscrollPos = min( HscrollMax, HscrollPos )

if( eax > HscrollPos ) then

    mov( HscrollPos, eax );
```

```
    endif;
    mov( eax, HscrollPos );
    w.SetScrollRange( hwnd, w.SB_HORZ, 0, HscrollMax, false );
    w.SetScrollPos( hwnd, w.SB_HORZ, HscrollPos, true );


    xor( eax, eax ); // return success.


end Size;



// HScroll-
//
//  Handles w.WM_HSCROLL messages.
//  On entry, L.O. word of wParam contains the scroll bar activity.

procedure HScroll( hwnd: dword; wParam:dword; lParam:dword );
begin HScroll;

    // Convert 16-bit command to 32 bits so we can use switch macro:

    movzx( (type word wParam), eax );
    switch( eax )

        case( w.SB_LINELEFT )

            mov( -1, eax );

        case( w.SB_LINERIGHT )

            mov( 1, eax );

        case( w.SB_PAGELEFT )

            mov( -8, eax );

        case( w.SB_PAGERIGHT )

            mov( 8, eax );

        case( w.SB_THUMBPOSITION )

            movzx( (type word wParam[2]), eax );
            sub( HscrollPos, eax );

        default

            xor( eax, eax );

    endswitch;

    // eax =
    //  max( -HscrollPos, min( eax, HscrollMax - HscrollPos ))
```

```
    mov( HscrollPos, edx );
    neg( edx );
    mov( HscrollMax, ecx );
    add( edx, ecx );
    if( eax > (type int32 ecx) ) then

        mov( ecx, eax );

    endif;
    if( eax < (type int32 edx )) then

        mov( edx, eax );

    endif;
    if( eax <> 0 ) then

        add( eax, HscrollPos );
        neg( eax );
        w.SetScrollPos( hwnd, w.SB_HORZ, HscrollPos, true );

    endif;
    w.InvalidateRect( hwnd, NULL, false );
    xor( eax, eax ); // return success

end HScroll;



// VScroll-
//
//  Handles the w.WM_VSCROLL messages from Windows.
//  The L.O. word of wParam contains the action/command to be taken.
//  The H.O. word of wParam contains a distance for the w.SB_THUMBTRACK
//  message.

procedure VScroll( hwnd: dword; wParam:dword; lParam:dword );
begin VScroll;

    movzx( (type word wParam), eax );
    switch( eax )

        case( w.SB_TOP )

            mov( VscrollPos, eax );
            neg( eax );

        case( w.SB_BOTTOM )

            mov( ClientSizeY, eax );
            cdq();
            idiv( AverageCharHeight );
            neg( eax );
            add( VscrollMax, eax );
            sub( VscrollPos, eax );

        case( w.SB_LINEUP )
```

```
        mov( -1, eax );

    case( w.SB_LINEDOWN )

        mov( 1, eax );

    case( w.SB_PAGEUP )

        mov( ClientSizeY, eax );
        cdq();
        idiv( AverageCharHeight );
        neg( eax );
        if( (type int32 eax) > -1 ) then

            mov( -1, eax );

        endif;

    case( w.SB_PAGEDOWN )

        mov( ClientSizeY, eax );
        cdq();
        idiv( AverageCharHeight );
        if( (type int32 eax) < 1 ) then

            mov( 1, eax );

        endif;

    case( w.SB_THUMBTRACK )

        movzx( (type word wParam[2]), eax );
        sub( VscrollPos, eax );

    default

        xor( eax, eax );

endswitch;

// VscrollPos += min( eax, VscrollMax - VscrollPos )

mov( VscrollMax, ecx );
sub( VscrollPos, ecx );
if( eax > (type int32 ecx)) then

    mov( ecx, eax );

endif;
add( VscrollPos, eax );
if( @s ) then

    xor( eax, eax );

endif;
mov( eax, VscrollPos );
w.SetScrollPos( hwnd, w.SB_VERT, eax, true );
```

```
    w.InvalidateRect( hwnd, NULL, false );
    xor( eax, eax ); // return success.

end VScroll;


// KeyMsg-
//
//  Handles the keyboard messages from Windows.
//  This routine creates a keyboard message record and adds it to the
//  list we're building in memory.

procedure KeyMsg
(
    hwnd:dword;
    msg:string;
    wParam:dword;
    lParam:dword;
    ASCII:dword
);
const
    msgPtr :text := "(type keymsg_t [ ebx] )";

var
    ThisMsg :keymsgPtr_t;

begin KeyMsg;

    push( ebx );

    // Allocate storage for a new node in our
    // keymsg list:

    malloc( @size( keymsg_t ) );
    mov( eax, ThisMsg );
    mov( eax, ebx );

    // Build the current key message record that we've
    // just allocated:

    mov( NULL, msgPtr.Next );

    mov( msg, eax );
    mov( eax, msgPtr.MsgStr );
    mov( wParam, eax );
    mov( eax, msgPtr.VirtKey );

    // The repeat count is bits 0..16 of lParam:

    mov( lParam, eax );
    mov( ax, msgPtr.RepeatCnt );

    // The transition flag is in bit 31:

    test( eax, eax );
    sets( cl );
    mov( cl, msgPtr.Transition );
```

```
// The PrevKeyState flag is in bit 30:

bt( 30, eax );
setc( cl );
mov( cl, msgPtr.PrevKeyState );

// The Context flag is in bit 29:

bt( 29, eax );
setc( cl );
mov( cl, msgPtr.Context );

// The extended key flag is in bit 24:

bt( 24, eax );
setc( cl );
mov( cl, msgPtr.ExtendedKey );

// The eight-bit scan code is in bits 16..23:

shr( 16, eax );
mov( al, msgPtr.ScanCode );

// Save the ASCII code in the record entry:

mov( (type byte ASCII), al );
mov( al, msgPtr.ASCIIcode );

// Call GetKeyState to get the remaining
// flags. Most of these set bit 31 to 1 if
// the current state is true:

w.GetKeyState( w.VK_LSHIFT );
shr( 31, eax );
mov( al, msgPtr.LshiftState );

w.GetKeyState( w.VK_RSHIFT );
shr( 31, eax );
mov( al, msgPtr.RshiftState );

w.GetKeyState( w.VK_LCONTROL );
shr( 31, eax );
mov( al, msgPtr.LctrlState );

w.GetKeyState( w.VK_RCONTROL );
shr( 31, eax );
mov( al, msgPtr.RctrlState );

w.GetKeyState( w.VK_LMENU );
shr( 31, eax );
mov( al, msgPtr.LaltState );

w.GetKeyState( w.VK_RMENU );
shr( 31, eax );
mov( al, msgPtr.RaltState );
```

```
// For the "lock" keys, bit zero
// determines if the keyboard is
// in the "locked" state:

w.GetKeyState( w.VK_CAPITAL );
and( 1, eax );
mov( al, msgPtr.CapsLock );

w.GetKeyState( w.VK_NUMLOCK );
and( 1, eax );
mov( al, msgPtr.NumLock );

w.GetKeyState( w.VK_SCROLL );
and( 1, eax );
mov( al, msgPtr.ScrlLock );

// Bump out key event counter and
// add the current entry to our list
// of key events.

inc( KeyPressCnt );
if( EndKeyPressList = NULL ) then

    mov( ebx, KeyPressList );
    mov( ebx, EndKeyPressList );

else

    mov( EndKeyPressList, eax );
    mov( ebx, (type keymsg_t [ eax]).Next );
    mov( ebx, EndKeyPressList );

endif;

// Whenever they press a key, automatically move to the end of the
// display list so that the new line is displayed:

mov( KeyPressCnt, ebx );
mov( ClientSizeY, eax );    // Compute size of window.
cdq();
idiv( AverageCharHeight );
sub( eax, ebx );
if( @s ) then

    xor( ebx, ebx );

endif;
mov( ebx, VscrollMax );     // VscrollMax = max( 0, KeyPressCnt-scrnsize)

push( eax );
w.SetScrollRange( hwnd, w.SB_VERT, 0, ebx, false );
pop( ebx );

// Move the thumb on the scroll bar to the appropriate
// position:

mov( KeyPressCnt, eax );
```

```
    sub( ebx, eax );
    if( @s ) then

        xor( eax, eax );

    endif;
    mov( eax, VscrollPos );

    w.SetScrollPos( hwnd, w.SB_VERT, eax, true );

    // Force Windows to redraw this window without erasing
    // it so that we get feedback in the window:

    w.InvalidateRect( hwnd, NULL, false );

    pop( ebx );

end KeyMsg;


// KeyDown, KeyUp, SysKeyDown, SysKeyUp-
// These procedures handle the actual Windows' keyboard messages
// and pass them on to KeyMsg to do the actual work.

procedure KeyDown( hwnd: dword; wParam:dword; lParam:dword );
begin KeyDown;

    KeyMsg( hwnd, "WM_KEYDOWN", wParam, lParam, 0 );

end KeyDown;

procedure KeyUp( hwnd: dword; wParam:dword; lParam:dword );
begin KeyUp;

    KeyMsg( hwnd, "WM_KEYUP", wParam, lParam, 0 );

end KeyUp;

procedure SysKeyDown( hwnd: dword; wParam:dword; lParam:dword );
begin SysKeyDown;

    KeyMsg( hwnd, "WM_SYSKEYDOWN", wParam, lParam, 0 );

end SysKeyDown;

procedure SysKeyUp( hwnd: dword; wParam:dword; lParam:dword );
begin SysKeyUp;

    KeyMsg( hwnd, "WM_SYSKEYUP", wParam, lParam, 0 );

end SysKeyUp;


procedure CharMsg( hwnd: dword; wParam:dword; lParam:dword );
begin CharMsg;

    KeyMsg( hwnd, "WM_CHAR", 0, lParam, wParam );
```

```
end CharMsg;


procedure DeadCharMsg( hwnd: dword; wParam:dword; lParam:dword );
begin DeadCharMsg;

    KeyMsg( hwnd, "WM_DEADCHAR", 0, lParam, wParam );

end DeadCharMsg;


procedure SysCharMsg( hwnd: dword; wParam:dword; lParam:dword );
begin SysCharMsg;

    KeyMsg( hwnd, "WM_SYSCHAR", 0, lParam, wParam );

end SysCharMsg;


procedure SysDeadCharMsg( hwnd: dword; wParam:dword; lParam:dword );
begin SysDeadCharMsg;

    KeyMsg( hwnd, "WM_SYSDEADCHAR", 0, lParam, wParam );

end SysDeadCharMsg;

end keytest;
```

Before concluding the discussion of keyboard messages and events, we re going to take a look at a more practical application - one that lets you enter text from the keyboard into a window. The following application is a very simple typewriter application that displays the characters you type on the keyboard in the application s window. In addition to the standard ASCII characters (that this program writes to the display), this application also handles the following special keystrokes:

- Left arrow: moves the caret one position to the left unless the caret is already in the left-most column.
- Right arrow: moves the caret one position to the right unless the caret is in the right-most column.
- Up arrow: moves the caret up one line (preserving the horizontal position) unless the caret is already on the top line of the screen.
- Down arrow: moves the caret down on line on the screen unless the caret is on the bottom line.
- End: moves the caret to the first position beyond the last non-blank character on the current line.
- Home: moves the caret to the beginning of the current line.
- PgUp: positions the caret on the top line of the window.
- PgDn: positions the caret on the bottom line of the window.
- Del: deletes the character under the caret, sliding all characters to the right of the caret one position to the left (filling the last character position on the line with a space).
- Backspace: deletes the character to the left of the caret by moving all the characters on the current line, from the caret position to the end of the line, one position to the left.

- Tab: writes spaces to fill out to the start of the next tab position on the line. Tabstops occur every eight character positions.
- Line feed: behaves just like the down arrow key.
- Carriage return: positions the caret to the beginning of the next line.
- Escape: clears the window and homes the cursor.

If the keyboard message is not one of the above special codes and it is an ASCII character code (i.e., the application receives a `w.WM_CHAR` message), then this program writes the arriving character to the window at the current caret position.

In order to significantly reduce the effort needed to write this program, the *typewriter.hla* application uses a *fixed system font*. With a fixed pitch system font, each character consumes the same amount of horizontal space on a line. This signficantly reduces the effort needed to write this application because we can use the caret s physical position on the Window (divided by an appropriate value) as an index into an in-memory buffer. Were we to have used a proportional font, computing an index into each new line (that would preserve the caret s physical position on the screen) would have been quite a bit more work. Therefore, one of the first things this application does in the `Create` procedure (that handles `w.WM_CREATE` messages) is to get a handle to the fixed system font. You could create a new fixed system font via a call to a function like w.GetFontIndirect but an easier solution is to grab a handle to the system s fixed font with a call to the `w.GetStockObject` API function:

```
// Create-
//
//  This procedure responds to the w.WM_CREATE message.
//  Windows sends this message once when it creates the
//  main window for the application.  We will use this
//  procedure to do any one-time initialization that
//  must take place in a message handler.

procedure Create( hwnd: dword; wParam:dword; lParam:dword );
var
    hdc:    dword;              // Handle to video display device context
    tm:     w.TEXTMETRIC;
begin Create;

    // First, create a useful font (fixed pitch):

    w.GetStockObject( w.SYSTEM_FIXED_FONT );
    mov( eax, TheFont );
      .
      .
      .
```

Using the fixed system font stock object is a good idea as this font is available on all systems and generally looks very good.

Once the `Create` procedure obtains a handle to the fixed system font, the next step is to get the font metric information for that font. This is necessary because we ll position the caret at pixel offsets on the screen and we ll need to convert those pixel offsets to character offsets. To do that, we ll need to divide the caret s (pixel) location by the width and height of the character. A quick call to GetTextMetrics supplies this information for us. The following code completes the `Create` procedure:

```
    .
```

```
    .
    .
GetDC( hwnd, hdc );

        // Initialization:
        //
        //  Get the text metric information so we can compute
        //  the average character heights and widths.

        SelectObject( TheFont );

        GetTextMetrics( tm );

        mov( tm.tmHeight, eax );
        add( tm.tmExternalLeading, eax );
        mov( eax, AverageCharHeight );

        mov( tm.tmAveCharWidth, eax );
        mov( eax, AverageCharWidth );

        // If bit #0 of tm.tmPitchAndFamily is set, then
        // we've got a proportional font.  In that case
        // set the average capital width value to 1.5 times
        // the average character width.  If bit #0 is clear,
        // then we've got a fixed-pitch font and the average
        // capital letter width is equal to the average
        // character width.

        mov( eax, ebx );
        shl( 1, tm.tmPitchAndFamily );
        if( @c ) then

            shl( 1, ebx );              // 2*AverageCharWidth

        endif;
        add( ebx, eax );                // Computes 2 or 3 times eax.
        shr( 1, eax );                  // Computes 1 or 1.5 times eax.
        mov( eax, AverageCapsWidth );

    ReleaseDC;

end Create;
```

The `Paint` procedure in the *typewriter.hla* application is fairly simple. Whenever Windows requests a screen redraw, this program simply redraws the text from an internal buffer where the keyboard handling procedures stuff all the arriving characters. See the program listing a little bit later for the details.

The typewriter application maintains a buffer large enough to hold all the characters that will fit on the window (i.e., one screen buffer full of data). Should the user decide to resize the window, we ve got a small problem - if the user makes the window larger the current data buffer will not be large enough to hold the data; if the user makes the window smaller, then the buffer will be too large (which isn t as bad as being too small, but an issue nonetheless). To solve this problem, the typewriter.hla application clears the window, deallocates the existing buffer (if there is one), and then reallocates storage for a new buffer whenever the user resizes the window. This code also initializes the buffer with spaces and homes the caret to the (0,0) position. See the `Size` procedure in the full source code a little later for the details of this procedure.

One big difference between the *typewriter.hla* application and the previous two applications in this chapter is that the *typewriter.hla* application has to maintain the caret. As you ll recall, there is only one system-wide caret and the application must properly maintain the caret by watching for `w.WM_FOCUS` and `w.WM_KILLFOCUS` messages. Whenever a `w.WM_FOCUS` message comes along, the application should create a caret and make it visible in the window. Similarly, whenever the application loses the focus (i.e., a `w.WM_KILLFOCUS` message arrives) it needs to hide the caret and destory it. Here s the code that does this:

```
// SetFocus-
//
//  This procedure gets called whenever this application gains the
// input focus.

procedure SetFocus( hwnd: dword; wParam:dword; lParam:dword );
begin SetFocus;

    w.CreateCaret( hwnd, NULL, AverageCharWidth, AverageCharHeight );
    gotoxy( x, y );
    w.ShowCaret( hwnd );
    sub( eax, eax );  // Return success

end SetFocus;


// KillFocus-
//
//  Processes the WM_KILLFOCUS message that gets sent whenever this
// application is losing the input focus.

procedure KillFocus( hwnd: dword; wParam:dword; lParam:dword );
begin KillFocus;

    w.HideCaret( hwnd );
    w.DestroyCaret();
    sub( eax, eax );  // Return success

end KillFocus;
```

Beyond these simple routines, the majority of the remaining effort consists of dealing with the keyboard messages that come along and, if appropriate, shoving the ASCII codes for the keyboard characters into the memory buffer or performing whatever activity the (control) keystrokes require. Here s the complete code for the application:

```
// typewriter.hla-
//
//  This program simulates a "typewriter" by copying keystrokes to the display.

unit typewriter;

// Set the following to true to display interesting information
// during program operation. You must be running
// the "DebugWindow" application for this output to appear.

?debug := false;
```

```
#includeonce( "excepts.hhf" )
#includeonce( "conv.hhf" )
#includeonce( "hll.hhf" )
#includeonce( "memory.hhf" )
#includeonce( "w.hhf" )
#includeonce( "wpa.hhf" )
#includeonce( "winmain.hhf" )

?@NoDisplay := true;
?@NoStackAlign := true;

const
    ESC := #$1b;    // Escape character code.


static

    TheFont             :dword;         // Font we'll use.

    AverageCapsWidth    :dword;         // Font metric values.
    AverageCharWidth    :dword;
    AverageCharHeight   :dword;

    maxCharWidth        :dword;
    maxCharHeight       :dword;

    ClientSizeX         :int32 := 0;    // Size of the client area
    ClientSizeY         :int32 := 0;    //  where we can paint.
    x                   :int32 := 0;    // Caret x-coordinate.
    y                   :int32 := 0;    // Caret y-coordinate.
    scrnBuf             :pointer to char := NULL;

readonly

    ClassName   :string := "typewriterWinClass";        // Window Class Name
    AppCaption  :string := "typewriter Program";        // Caption for Window


    // The dispatch table:
    //
    //  This table is where you add new messages and message handlers
    //  to the program.  Each entry in the table must be a MsgProcPtr_t
    //  record containing two entries: the message value (a constant,
    //  typically one of the w.WM_***** constants found in windows.hhf)
    //  and a pointer to a "MsgProcPtr_t" procedure that will handle the
    //  message.


    Dispatch    :MsgProcPtr_t; @nostorage;

        MsgProcPtr_t
            MsgProcPtr_t:[  w.WM_PAINT,      &Paint             ],
            MsgProcPtr_t:[  w.WM_KEYDOWN,    &KeyDown           ],
            MsgProcPtr_t:[  w.WM_CHAR,       &CharMsg           ],
            MsgProcPtr_t:[  w.WM_SETFOCUS,   &SetFocus          ],
            MsgProcPtr_t:[  w.WM_KILLFOCUS,  &KillFocus         ],
            MsgProcPtr_t:[  w.WM_SIZE,       &Size              ],
```

```
            MsgProcPtr_t:[  w.WM_CREATE,     &Create            ],
            MsgProcPtr_t:[  w.WM_DESTROY,    &QuitApplication   ],

            // Insert new message handler records here.

            MsgProcPtr_t:[  0, NULL ];   // This marks the end of the list.




/*************************************************************************/
/*          W I N M A I N   S U P P O R T   C O D E                      */
/*************************************************************************/

// initWC - We don't have any initialization to do, so just return:

procedure initWC; @noframe;
begin initWC;

    ret();

end initWC;

// appCreateWindow- the default window creation code is fine, so just
//                  call defaultCreateWindow.

procedure appCreateWindow; @noframe;
begin appCreateWindow;

    jmp defaultCreateWindow;

end appCreateWindow;


// appException-
//
// Gives the application the opportunity to clean up before
// aborting when an unhandled exception comes along:

procedure appException( theException:dword in eax );
begin appException;

    raise( eax );

end appException;



// This is the custom message translation procedure.
// We're not doing any custom translation, so just return EAX=0
// to tell the caller to go ahead and call the default translation
// code.

procedure LocalProcessMsg( var lpmsg:w.MSG );
begin LocalProcessMsg;
```

```
    xor( eax, eax );

end LocalProcessMsg;


/*****************************************************************************/
/*          A P P L I C A T I O N   S P E C I F I C   C O D E                */
/*****************************************************************************/


// gotoxy-
//
// Positions the cursor at the specified (x,y) *character* coordinate.

procedure gotoxy( x:dword in eax; y:dword in edx );
begin gotoxy;

    intmul( AverageCharWidth, eax );
    intmul( AverageCharHeight, edx );
    w.SetCaretPos( eax, edx );

end gotoxy;


// QuitApplication:
//
//  This procedure handles the w.WM_DESTROY message.
//  It tells the application to terminate.  This code sends
//  the appropriate message to the main program's message loop
//  that will cause the application to terminate.

procedure QuitApplication( hwnd: dword; wParam:dword; lParam:dword );
begin QuitApplication;

    w.PostQuitMessage( 0 );

end QuitApplication;


// Create-
//
//  This procedure responds to the w.WM_CREATE message.
//  Windows sends this message once when it creates the
//  main window for the application.  We will use this
//  procedure to do any one-time initialization that
//  must take place in a message handler.

procedure Create( hwnd: dword; wParam:dword; lParam:dword );
var
    hdc:    dword;              // Handle to video display device context
    tm:     w.TEXTMETRIC;
begin Create;

    // First, create a useful font (fixed pitch):

    w.GetStockObject( w.SYSTEM_FIXED_FONT );
```

```
    mov( eax, TheFont );



// Paint:
//
//  This procedure handles the w.WM_PAINT message.
//  For this keytest program, the Paint procedure
//  displays the list of key events we've saved in memory.

procedure Paint( hwnd: dword; wParam:dword; lParam:dword );
var
    value       :string;
    valData     :char[ 256];
    vallen      :uns32;         // Length of value string.

    hdc         :dword;         // Handle to video display device context
    ps          :w.PAINTSTRUCT; // Used while painting text.


begin Paint;

    // Message handlers must preserve EBX, ESI, and EDI.
    // (They've also got to preserve EBP, but HLA's procedure
    // entry code already does that.)

    push( ebx );
    push( esi );
    push( edi );

    // Initialize the value->valData string object:

    mov( str.init( (type char valData), 256 ), value );

    // When Windows requests that we draw the window,
    // fill in the string in the center of the screen.
    // Note that all GDI calls (e.g., w.DrawText) must
    // appear within a BeginPaint..EndPaint pair.

    BeginPaint( hwnd, ps, hdc );

        // Select the fixed-pitch font into our context:

        SelectObject( TheFont );
        mov( y, edi );
        intmul( maxCharHeight, edi );
        mov( scrnBuf, ebx );
        for( mov( 0, esi ); esi <= edi; add( AverageCharHeight, esi )) do

            TextOut( 0, esi, ebx, maxCharWidth );
            add( maxCharWidth, ebx );

        endfor;

    EndPaint;

    pop( edi );
```

```
    pop( esi );
    pop( ebx );
    xor( eax, eax ); // Return success

end Paint;



// Size-
//
//  This procedure handles the w.WM_SIZE message
//
//  L.O. word of lParam contains the new X Size
//  H.O. word of lParam contains the new Y Size

procedure Size( hwnd: dword; wParam:dword; lParam:dword );
begin Size;

    // Convert new X size to 32 bits and save:

    movzx( (type word lParam), eax );
    mov( eax, ClientSizeX );
    cdq();
    idiv( AverageCharWidth );
    mov( eax, maxCharWidth );

    // Convert new Y size to 32 bits and save:

    movzx( (type word lParam[2]), eax );
    mov( eax, ClientSizeY );
    cdq();
    idiv( AverageCharHeight );
    mov( eax, maxCharHeight );

    // Allocate storage for a screen buffer here:

    if( scrnBuf <> NULL ) then

        free( scrnBuf );

    endif;
    mov( maxCharWidth, eax );
    intmul( maxCharHeight, eax );
    mov( eax, ecx );
    malloc( eax );
    mov( eax, scrnBuf );

    // Fill the buffer with spaces:

    push( edi );
    mov( eax, edi );
    mov( ' ', al );
    rep.stosb();
    pop( edi );

    xor( eax, eax );
    mov( eax, x );   // Home caret to (0,0)
```

```
    mov( eax, y );

    w.GetFocus();
    if( eax = hwnd ) then

        gotoxy( 0, 0 );

    endif;
    xor( eax, eax ); // Return success.


end Size;




// SetFocus-
//
//  This procedure gets called whenever this application gains the
// input focus.

procedure SetFocus( hwnd: dword; wParam:dword; lParam:dword );
begin SetFocus;

    w.CreateCaret( hwnd, NULL, AverageCharWidth, AverageCharHeight );
    gotoxy( x, y );
    w.ShowCaret( hwnd );
    sub( eax, eax );  // Return success

end SetFocus;


// KillFocus-
//
//  Processes the WM_KILLFOCUS message that gets sent whenever this
// application is losing the input focus.

procedure KillFocus( hwnd: dword; wParam:dword; lParam:dword );
begin KillFocus;

    w.HideCaret( hwnd );
    w.DestroyCaret();
    sub( eax, eax );  // Return success

end KillFocus;




// KeyDown-
//
//  This procedure handles WM_KEYDOWN messages. Mainly, we process
// cursor control keystrokes here.

procedure KeyDown( hwnd: dword; wParam:dword; lParam:dword );
var
    hdc         :dword;         // Handle to video display device context
```

```
begin KeyDown;

    mov( wParam, eax );
    switch( eax )

        case( w.VK_LEFT )

            // Move the cursor one position to the left.
            // However, if we are already in column zero,
            // then ignore the request.

            dec( x );
            if( @s ) then

                mov( 0, x );

            endif;

        case( w.VK_RIGHT )

            // Bump the cursor position one spot to the right.
            // If we are already on the right hand side of the
            // screen, then ignore this request.

            inc( x );
            mov( x, eax );
            if( eax > maxCharWidth ) then

                mov( maxCharWidth, eax );
                mov( eax, x );

            endif;

        case( w.VK_UP )

            // Move the cursor up one line unless we're
            // already at line zero:

            dec( y );
            if( @s ) then

                mov( 0, y );

            endif;

        case( w.VK_DOWN )

            // Move the cursor down one line unless
            // we're already on the bottom line of the display.

            inc( y );
            mov( y, eax );
            if( eax > maxCharHeight ) then

                mov( maxCharHeight, eax );
                mov( eax, y );
```

```
    endif;

case( w.VK_END )

    // Move the cursor to the first spot beyond the last non-space
    // character on the line.
    //
    // Calculate the end of the line as the last non-blank
    // character in the buffer on the current line:

    mov( y, edx );                  // Calculate start of buffer adrs
    mov( maxCharWidth, ecx );
    intmul( ecx, edx );
    add( scrnBuf, edx );
    add( edx, ecx );                // Point ecx at end of buffer.
    repeat

        dec( ecx );

    until( ecx = edx || (type char[ecx]) <> ' ');
    sub( edx, ecx );
    inc( ecx );     // Move just beyond character
    if( ecx > maxCharWidth ) then

        // If last position contains a character, don't
        // increment beyond that point.

        mov( maxCharWidth, ecx );

    endif;
    mov( ecx, x );

case( w.VK_HOME )

    // Pressing the HOME key moves the cursor to the
    // beginning of the current line.

    mov( 0, x );

case( w.VK_PRIOR )

    // PgUp positions the caret on the top line
    // of the display:

    mov( 0, y );

case( w.VK_NEXT )

    // PgDn positions the caret on the bottom line of
    // the display:

    mov( maxCharHeight, eax );
    dec( eax );
    mov( eax, y );

case( w.VK_DELETE )
```

```
            // Delete the key under the cursor.

            push( esi );
            push( edi );

            // First, we've got to determine the address
            // of this character:

            mov( maxCharWidth, esi );
            mov( y, edi );
            intmul( esi, edi );
            add( scrnBuf, edi );        // Adrs of start of line
            push( edi );                // Save for later
            add( x, edi );              // Address of character position
            lea( esi, [ edi+1] );       // Start copying with next char

            // Now, copy the remaining characters on the line
            // over the current character:

            mov( maxCharWidth, ecx );   // Compute chars remaining on line
            sub( x, ecx );
            rep.movsb();                // Copy chars over char to delete.
            mov( ' ', (type char [ edi] )); // Put a space in last position.

            // Redraw the line of text so we can see the effect
            // of the deletion on the display:

            pop( edi );                 // Retrieve address of line.
            w.HideCaret( hwnd );
            GetDC( hwnd, hdc );

                SelectObject( TheFont );

                mov( y, eax );
                intmul( AverageCharHeight, eax );
                TextOut( 0, eax, edi, maxCharWidth );

            ReleaseDC;
            w.ShowCaret( hwnd );

            pop( edi );
            pop( esi );

    endswitch;
    gotoxy( x, y );
    sub( eax, eax );  // Return success

end KeyDown;


// CharMsg-
//
//  This procedure handles the WM_CHAR messages that come along:

procedure CharMsg( hwnd: dword; wParam:dword; lParam:dword );
var
    hdc         :dword;         // Handle to video display device context
```

```
    charAdrs    :dword;
begin CharMsg;

    mov( wParam, eax );     // Get ASCII code

    if( al = stdio.bs ) then

        // Handle the backspace key.
        // Just like the DEL key, except we delete the
        // character to the left of the cursor rather than
        // the character under the cursor. If in column zero,
        // do nothing. (See the comments for the DEL key for
        // more details.)

        if( x <> 0 ) then

            push( esi );
            push( edi );

            mov( maxCharWidth, ecx );
            mov( y, esi );              // esi = y*maxCharWidth+scrnBuf
            intmul( ecx, esi );         //  which is the address of the
            add( scrnBuf, esi );        //  start of the line.
            push( esi );                // Save start address for later.
            add( x, esi );              // Address of character position
            lea( edi, [ esi-1] );       // Start copying with this char
            sub( x, ecx );              // Compute remaining chars
            rep.movsb();                // Copy chars over char to delete.
            mov( ' ', (type char [ edi])); // Put a space in last position.

            // Reprint the line we've just modified:

            pop( edi );
            w.HideCaret( hwnd );
            GetDC( hwnd, hdc );

                SelectObject( TheFont );

                mov( y, eax );
                intmul( AverageCharHeight, eax );
                TextOut( 0, eax, edi, maxCharWidth );

            ReleaseDC;
            w.ShowCaret( hwnd );

            // We need to move the cursor to the left one
            // spot to move it over the character we
            // just deleted:

            dec( x );
            if( @s ) then

                mov( 0, x );

            endif;
```

```
        pop( edi );
        pop( esi );

    endif;

elseif( al = stdio.tab ) then

    // If the user presses the TAB key, then write spaces
    // from the current caret position up to the next
    // x-coordinate that is an even multiple of eight
    // characters (assume 8-character tab stop positions).

    push( ebx );
    mov( x, ebx );
    repeat

        w.SendMessage( hwnd, w.WM_CHAR, ' ', 1 );
        inc( bl );
        test( %111, bl );

    until( @z );

elseif( al = stdio.lf ) then

    // Treat LF just like the "down" key:

    w.SendMessage( hwnd, w.WM_KEYDOWN, w.VK_DOWN, 1 );
    w.SendMessage( hwnd, w.WM_KEYUP, w.VK_DOWN, 1 );


elseif( al = stdio.cr ) then

    // When the user presses enter, go to the beginning
    // of the next line:

    mov( 0, x );
    w.SendMessage( hwnd, w.WM_CHAR, stdio.lf, 1 );

elseif( al = ESC ) then

    // If the user presses the ESC key, then clear the
    // screen:

    mov( maxCharWidth, eax );
    intmul( maxCharHeight, eax );
    mov( eax, ecx );

    push( edi );
    mov( scrnBuf, edi );
    mov( ' ', al );
    rep.stosb();
    pop( edi );

    xor( eax, eax );
    mov( eax, x );   // Home caret to (0,0)
    mov( eax, y );
```

```
        // Force Windows to redraw this window.

        w.InvalidateRect( hwnd, NULL, true );


    else // we've got a normal character - print it

        // Compute index into buffer:

        mov( y, ecx );
        intmul( maxCharWidth, ecx );
        add( x, ecx );
        add( scrnBuf, ecx );

        mov( al, [ecx] );
        mov( ecx, charAdrs );

        // Display the character on the screen:

        w.HideCaret( hwnd );
        GetDC( hwnd, hdc );

            SelectObject( TheFont );

            mov( y, eax );
            intmul( AverageCharHeight, eax );
            mov( x, ecx );
            intmul( AverageCharWidth, ecx );
            TextOut( ecx, eax, charAdrs, 1 );

        ReleaseDC;
        w.ShowCaret( hwnd );

        // Update the caret position:

        inc( x );
        mov( x, eax );
        if( eax > maxCharWidth ) then

            mov( 0, x );
            inc( y );
            mov( y, eax );
            if( eax > maxCharHeight ) then

                mov( 0, y );

            endif;

        endif;

    endif;
    gotoxy( x, y );
    xor( eax, eax );  // Return success

end CharMsg;

end typewriter;
```

8.5: Mouse Messages

Although the mouse is a central input device for Windows, it is actually a fairly simple device and the messages Windows sends to your applications concerning the mouse are very straight-forward.

Windows sends two types of mouse messages to an application s window procedure: client area messages and non-client area messages. Client area messages are sent to an application whenever there is a mouse event within the client area of the application s window, non-client messages are sent whenever there is a mouse event outside the client area, but still within the bounds of the window (e.g., in the title bar or along the window s border). Note that an application does not normally receive messages for mouse events outside the window (though there are a couple of exceptions to this rule).

Table 8-3 lists the messages that Windows sends an application whenever a mouse event occurs within the client area.

When a mouse event message arrives, the `lParam` parameter contains the current mouse (x,y) position using client-window coordinates. The L.O. word of `lParam` contains the x-coordinate and the H.O. word contains the y-coordinate. Note that these two values are 16-bit *signed* integers. If you want to convert these values to 32-bit integers (as is usually the case) you will need to sign extend them, e.g.,

```
movsx( (type word lParam), eax );
movsx( (type word lParam[2]), ecx );
```

Under certain circumstances, mouse coordinates can actually be negative. If you simply zero extend these values to 32-bits such negative values will start looking like really large positive values.

The `wParam` parameter that Windows passes to your mouse message handling procedure is a set of bits specifying the state of the various mouse buttons in addition to the shift and control keys on the keyboard. Windows provides a set of constants (see Table 8-4) that you can use to test the bits in the `wParam` parameter.

Table 8-3: Client Area Mouse Message

Message	Description	lParam	wParam
w.WM_LBUTTONDBLCLK	Sent after the second click of a double-click sequence by the left mouse button.	L.O. word holds the mouse cursor's x-coordinate, H.O. word holds the cursor's y-coordinate.	Flags (bits) containing modifier key and mouse button status (see text for details)
w.WM_LBUTTONDOWN	Sent when the user presses the left mouse button.		
w.WM_LBUTTONUP	Sent when the user releases the left mouse button.		
w.WM_MBUTTONDBLCLK	Sent after the second click of a double-click sequence by the middle mouse button.		
w.WM_MBUTTONDOWN	Sent when the user presses the middle mouse button		
w.WM_MBUTTONUP	Sent when the user releases the middle mouse button.		
w.WM_RBUTTONDBLCLK	Sent after the second click of a double-click sequence by the right mouse button.		
w.WM_RBUTTONDOWN	Sent when the user presses the right mouse button		
w.WM_RBUTTONUP	Sent when the user releases the right mouse button.		
w.WM_MOUSEMOVE	Sent whenever the user moves the mouse in the client area.		

Table 8-4: wParam Flags for a Mouse Event

Flag to AND with wParam	Meaning (if bit set)
w.MK_CONTROL	Control key on keyboard was down during mouse event.
w.MK_LBUTTON	Left button on mouse was down during mouse event.
w.MK_MBUTTON	Middle button on mouse was down during mouse event.
w.MK_RBUTTON	Right button on mouse was down during mouse event.
w.MK_SHIFT	Shift key on keyboard was down during mouse event (either shift key).

While you are moving the mouse through the client area of your application s window, Windows will send a constant stream of `w.WM_MOUSEMOVE` messages to your application. The exact number and rate of mouse movement messages is dependent upon your hardware and OS settings, but suffice to say that you will not get a mouse movement message for every pixel the mouse cursor traverses; if you rapidly move the mouse, Windows will accelerate the mouse movements and skip over several pixels with each message it sends to your application. The sample program appearing in a little bit will demonstrate this.

Windows sends the mouse button up and down messages to your application whenever the user presses one of the three mouse buttons within the client area of your application s window (see Table 8-3). Note that these messages are not guaranteed to come in pairs. Though most of the time you will receive a mouse down and then a mouse up message, it is quite possible to receive a mouse down message and then never receive the corresponding mouse up; for example, if the user presses the mouse button in your client area, drags the mouse cursor outside the window s client area, and then releases the mouse, you will not receive a mouse up event. Other scenerios are also possible. So don t count on these messages always occuring in pairs.

Double-click messages are special. First of all, Windows will not automatically send double-click messages to your application s window procedure - you have to specifically request that it do this. To make this request, you will have to logically OR in the constant `w.CS_DBLCLKS` into the `wc.style` field of the window class. As you may recall, we ve buried the initialization of the `wc` variable into the winmain library module, so we don t have direct access to the initialization of this structure. Fortunately, the winmain code does call a procedure in our code, `initWC`, that allows us to make any changes to the `wc` object prior to actually registering the window class. Therefore, we can add the `w.CS_DBLCLKS` style to our window class by adding the following code to the `initWC` procedure:

```
procedure initWC; @noframe;
begin initWC;

   or( w.CS_DBLCLKS, wc.style );// Activate double-clicks
   ret();

end initWC;
```

Once you ve done this, Windows will send the various double-click messages to your application.

An important thing to note about double click messages is that you don t get a single double-click message when the user double-clicks the mouse. You will actually get a mouse down message, a mouse up message, a dou ble-click message and then a final mouse up message (assuming the user double clicked the mouse button within the application s client window). In other words, Windows simply replaces the second mouse down message with a double-click message.

Without further ado, here s a quick program that demonstrates the use of the mouse within an application. This simple program, *MousePts*, simply tracks movements throughout a window while the left mouse button is down (drawing pixels at each `w.WM_MOUSEMOVE` event) and then drawing lines between all the points when the user releases the mouse button. Figure 8-1 shows some sample output just before releasing the mouse button, Figure 8-2 shows the output immediately after releasing the mouse button.

```
//
//  A program that demonstrates the use of the mouse.
//
// Note: this is a unit because it uses the WinMail library module that
//       provides a win32 main program for us.

unit MousePoints;

// Set the following to true to display interesting information
// about the bitmap file this program opens. You must be running
// the "DebugWindow" application for this output to appear.

?debug := false;

#includeonce( "hll.hhf" )
#includeonce( "w.hhf" )
#includeonce( "wpa.hhf" )
#includeonce( "winmain.hhf" )

?@NoDisplay := true;
?@NoStackAlign := true;

const
    maxPoints   := 2048;

static
    PointCnt:       uns32 := 0;
    PointsArray:    w.POINT[ maxPoints ];


readonly

    ClassName   :string := "MousePts2WinClass";      // Window Class Name
    AppCaption  :string := "MousePts2 Program";      // Caption for Window


    // The dispatch table:
    //
    //  This table is where you add new messages and message handlers
    //  to the program.  Each entry in the table must be a MsgProcPtr_t
    //  record containing two entries: the message value (a constant,
```

```
    //  typically one of the w.WM_***** constants found in windows.hhf)
    //  and a pointer to a "MsgProcPtr_t" procedure that will handle the
    //  message.


    Dispatch    :MsgProcPtr_t; @nostorage;

        MsgProcPtr_t
            MsgProcPtr_t:[  w.WM_DESTROY,        &QuitApplication    ],
            MsgProcPtr_t:[  w.WM_PAINT,          &Paint              ],
            MsgProcPtr_t:[  w.WM_LBUTTONDOWN,    &LButtonDown        ],
            MsgProcPtr_t:[  w.WM_LBUTTONUP,      &LButtonUp          ],
            MsgProcPtr_t:[  w.WM_MOUSEMOVE,      &MouseMove          ],

            // Insert new message handler records here.

            MsgProcPtr_t:[  0, NULL ];   // This marks the end of the list.



/*****************************************************************************/
/*          W I N M A I N   S U P P O R T   C O D E                          */
/*****************************************************************************/

// initWC - We don't have any initialization to do, so just return:

procedure initWC; @noframe;
begin initWC;

    ret();

end initWC;

// appCreateWindow- the default window creation code is fine, so just
//                  call defaultCreateWindow.

procedure appCreateWindow; @noframe;
begin appCreateWindow;

    jmp defaultCreateWindow;

end appCreateWindow;


// appException-
//
// Gives the application the opportunity to clean up before
// aborting when an unhandled exception comes along:

procedure appException( theException:dword in eax );
begin appException;

    raise( eax );

end appException;
```

```
// This is the custom message translation procedure.
// We're not doing any custom translation, so just return EAX=0
// to tell the caller to go ahead and call the default translation
// code.

procedure LocalProcessMsg( var lpmsg:w.MSG );
begin LocalProcessMsg;

    xor( eax, eax );

end LocalProcessMsg;


/*****************************************************************************/
/*          A P P L I C A T I O N   S P E C I F I C   C O D E                */
/*****************************************************************************/


// QuitApplication:
//
//  This procedure handles the w.WM_DESTROY message.
//  It tells the application to terminate.  This code sends
//  the appropriate message to the main program's message loop
//  that will cause the application to terminate.

procedure QuitApplication( hwnd: dword; wParam:dword; lParam:dword );
begin QuitApplication;

    // Tell the application to quit:

    w.PostQuitMessage( 0 );

end QuitApplication;




// LButtonDown:
//
//  This procedure handles the w.WM_LBUTTONDOWN message.

procedure LButtonDown( hwnd: dword; wParam:dword; lParam:dword );
begin LButtonDown;

    mov( 0, PointCnt );
    w.InvalidateRect( hwnd, NULL, true );
    xor( eax, eax ); // Return zero to indicate success.

end LButtonDown;


// LButtonUp:
//
//  This procedure handles the w.WM_LBUTTONUP message.
```

```
procedure LButtonUp( hwnd: dword; wParam:dword; lParam:dword );
begin LButtonUp;

    w.InvalidateRect( hwnd, NULL, true );
    xor( eax, eax ); // Return zero to indicate success.

end LButtonUp;


// MouseMove:
//
//  This procedure handles the w.WM_MOUSEMOVE message.

procedure MouseMove( hwnd: dword; wParam:dword; lParam:dword );
var
    hdc: dword;
begin MouseMove;

    test( w.MK_LBUTTON, wParam );
    if( @nz && PointCnt < maxPoints ) then

        GetDC( hwnd, hdc );
            mov( PointCnt, ecx );
            movzx( (type word lParam), eax );
            mov( eax, PointsArray.x[ ecx*8 ] );
            movzx( (type word lParam[2]), edx );
            mov( edx, PointsArray.y[ ecx*8 ] );
            SetPixel( eax, edx, 0 );
            inc( PointCnt );
        ReleaseDC;

    endif;
    xor( eax, eax );
end MouseMove;


// Paint:
//
//  This procedure handles the w.WM_PAINT message.

procedure Paint( hwnd: dword; wParam:dword; lParam:dword );

var
    hdc         :dword;         // Handle to video display device context.
    ps          :w.PAINTSTRUCT; // Used while painting text.

begin Paint;


    // Message handlers must preserve EBX, ESI, and EDI.
    // (They've also got to preserve EBP, but HLA's procedure
    // entry code already does that.)

    push( ebx );
    push( esi );
    push( edi );
```

```
    // Note that all GDI calls must appear within a
    // BeginPaint..EndPaint pair.

    BeginPaint( hwnd, ps, hdc );

        w.SetCursor( w.LoadCursor( NULL, @string( w.IDC_WAIT ) ));
        w.ShowCursor( true );
        MoveTo( PointsArray.x[ 0], PointsArray.y[ 0] );
        for( mov( 1, ebx ); ebx < PointCnt; inc( ebx )) do

            LineTo( PointsArray.x[ ebx*8], PointsArray.y[ ebx*8] );

        endfor;
        w.ShowCursor( false );
        w.SetCursor( w.LoadCursor( NULL, @string(w.IDC_ARROW )));

    EndPaint;


    pop( edi );
    pop( esi );
    pop( ebx );

end Paint;
end MousePoints;
```

Listing 8-6: The MousePts Program

Figure 8-1: MousePts Output, Before Releasing Button

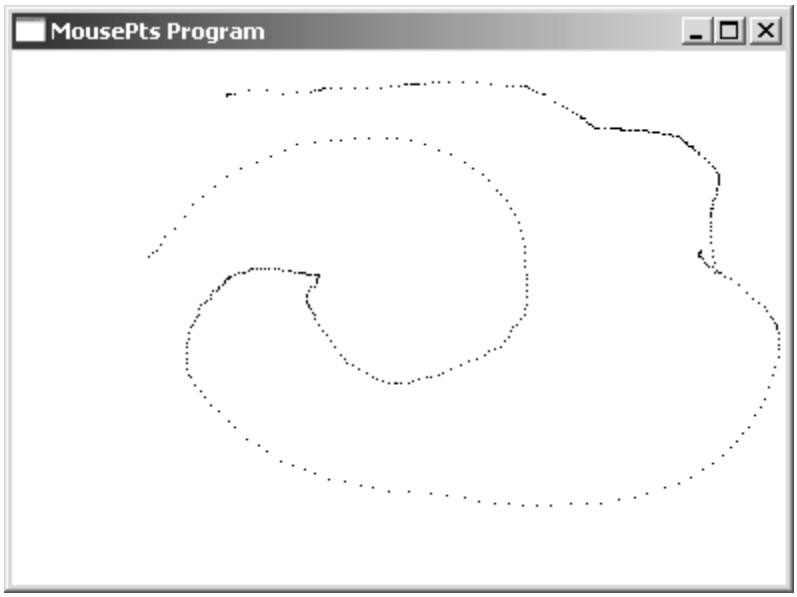
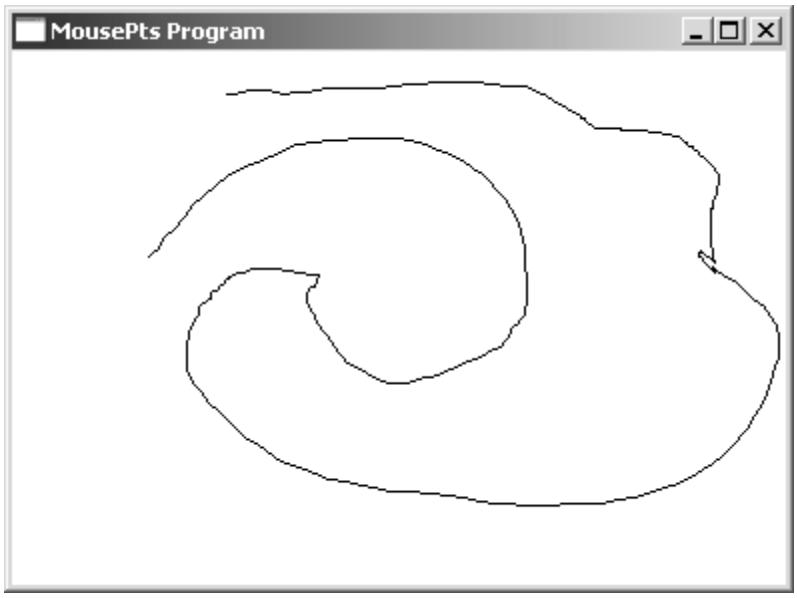


Figure 8-2: MousePts Output, After Releasing Button



The *MousePts* program continually selects points whenever a mouse movement occurs. In most applications you will want to choose the pixels to draw based on some even such as a mouse click. The MousePts2 program is a slight modification of the MousePts program that chooses pixels to plot based on mouse clicks. Another difference between the MousePts and MousePts2 programs is that MousePts2 draws straight lines between each point and every other point, rather than simply between adjacent points. The listing follows and some sample output appears in Figure 8-3.

```
// MousePts2.hla-
//
//  Another program that demonstrates the use of the mouse.
//
// Note: this is a unit because it uses the WinMail library module that
//       provides a win32 main program for us.

unit MousePoints2;

// Set the following to true to display interesting information
// about the bitmap file this program opens. You must be running
// the "DebugWindow" application for this output to appear.

?debug := false;

#includeonce( "hll.hhf" )
#includeonce( "w.hhf" )
#includeonce( "wpa.hhf" )
#includeonce( "winmain.hhf" )

?@NoDisplay := true;
?@NoStackAlign := true;

const
```

```
    maxPoints   := 128;

static
    PointCnt:       uns32 := 0;
    PointsToPlot:   uns32 := 0;
    PointsArray:    w.POINT[ maxPoints ];


readonly

    ClassName   :string := "MousePts2WinClass";     // Window Class Name
    AppCaption  :string := "MousePts2 Program";     // Caption for Window


    // The dispatch table:
    //
    //  This table is where you add new messages and message handlers
    //  to the program.  Each entry in the table must be a MsgProcPtr_t
    //  record containing two entries: the message value (a constant,
    //  typically one of the w.WM_***** constants found in windows.hhf)
    //  and a pointer to a "MsgProcPtr_t" procedure that will handle the
    //  message.


    Dispatch    :MsgProcPtr_t; @nostorage;

        MsgProcPtr_t
            MsgProcPtr_t:[  w.WM_DESTROY,        &QuitApplication    ],
            MsgProcPtr_t:[  w.WM_PAINT,          &Paint              ],
            MsgProcPtr_t:[  w.WM_LBUTTONDOWN,    &LButtonDown        ],
            MsgProcPtr_t:[  w.WM_LBUTTONDBLCLK,  &LButtonDblClk      ],

            // Insert new message handler records here.

            MsgProcPtr_t:[  0, NULL ];   // This marks the end of the list.



/***************************************************************************/
/*          W I N M A I N   S U P P O R T   C O D E                        */
/***************************************************************************/

// initWC - We have to activate the double-click feature in the window style

procedure initWC; @noframe;
begin initWC;

    or( w.CS_DBLCLKS, wc.style );   // Activate double-clicks
    ret();

end initWC;

// appCreateWindow- the default window creation code is fine, so just
//                  call defaultCreateWindow.

procedure appCreateWindow; @noframe;
begin appCreateWindow;
```

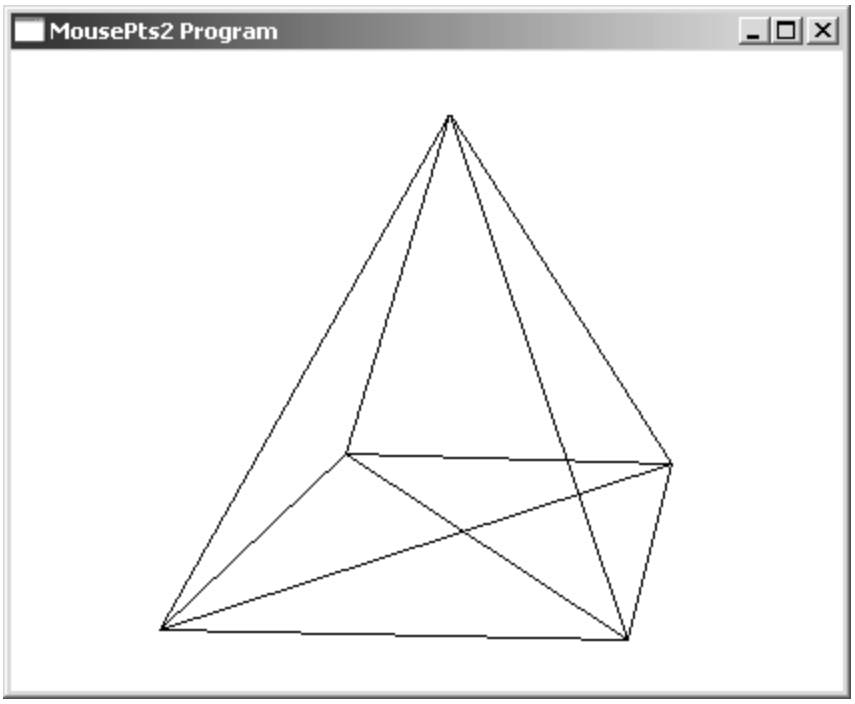
```
    jmp defaultCreateWindow;

end appCreateWindow;


// appException-
//
// Gives the application the opportunity to clean up before
// aborting when an unhandled exception comes along:

procedure appException( theException:dword in eax );
begin appException;

    raise( eax );

end appException;




// This is the custom message translation procedure.
// We're not doing any custom translation, so just return EAX=0
// to tell the caller to go ahead and call the default translation
// code.

procedure LocalProcessMsg( var lpmsg:w.MSG );
begin LocalProcessMsg;

    xor( eax, eax );

end LocalProcessMsg;


/*****************************************************************************/
/*          A P P L I C A T I O N   S P E C I F I C   C O D E                */
/*****************************************************************************/


// QuitApplication:
//
//  This procedure handles the w.WM_DESTROY message.
//  It tells the application to terminate.  This code sends
//  the appropriate message to the main program's message loop
//  that will cause the application to terminate.

procedure QuitApplication( hwnd: dword; wParam:dword; lParam:dword );
begin QuitApplication;

    // Tell the application to quit:

    w.PostQuitMessage( 0 );

end QuitApplication;
```

```
// LButtonDown:
//
//  This procedure handles the w.WM_LBUTTONDOWN message.

procedure LButtonDown( hwnd: dword; wParam:dword; lParam:dword );
var
    hdc:    dword;

begin LButtonDown;

    GetDC( hwnd, hdc );
        mov( PointCnt, ecx );
        movzx( (type word lParam), eax );
        mov( eax, PointsArray.x[ ecx*8 ] );
        movzx( (type word lParam[2]), edx );
        mov( edx, PointsArray.y[ ecx*8 ] );
        SetPixel( eax, edx, 0 );
        inc( PointCnt );
    ReleaseDC;
    xor( eax, eax ); // Return zero to indicate success.

end LButtonDown;

// LButtonDblClk:
//
//  This procedure handles the w.WM_LBUTTONSBLCLK message.

procedure LButtonDblClk( hwnd: dword; wParam:dword; lParam:dword );
begin LButtonDblClk;

    mov( PointCnt, PointsToPlot );
    mov( 0, PointCnt );
    w.InvalidateRect( hwnd, NULL, true );
    xor( eax, eax ); // Return zero to indicate success.

end LButtonDblClk;


// Paint:
//
//  This procedure handles the w.WM_PAINT message.

procedure Paint( hwnd: dword; wParam:dword; lParam:dword );

var
    hdc         :dword;         // Handle to video display device context.
    ps          :w.PAINTSTRUCT; // Used while painting text.

begin Paint;


    // Message handlers must preserve EBX, ESI, and EDI.
    // (They've also got to preserve EBP, but HLA's procedure
    // entry code already does that.)

    push( ebx );
```

```
    push( esi );
    push( edi );


    // Note that all GDI calls must appear within a
    // BeginPaint..EndPaint pair.

    BeginPaint( hwnd, ps, hdc );

        w.SetCursor( w.LoadCursor( NULL, @string( w.IDC_WAIT ) ));
        w.ShowCursor( true );

        // Draw a line between each point and every other point
        // in our points array.

        for( mov( 0, ebx ); ebx < PointsToPlot; inc( ebx )) do

            for( lea( esi, [ebx+1]); esi < PointsToPlot; inc( esi )) do

                MoveTo( PointsArray.x[ebx*8], PointsArray.y[ebx*8] );
                LineTo( PointsArray.x[esi*8], PointsArray.y[esi*8] );

            endfor;

        endfor;
        mov( 0, PointsToPlot );
        w.ShowCursor( false );
        w.SetCursor( w.LoadCursor( NULL, @string(w.IDC_ARROW )));

    EndPaint;


    pop( edi );
    pop( esi );
    pop( ebx );

end Paint;
end MousePoints2;
```

Listing 8-7: MousePts2 Program

Figure 8-3: MousePts2 Sample Output



Windows only sends the messages we ve discussed thus far when the mouse cursor is within the client area of your application s window. If the mouse cursor is in one of the non-client areas of your application s window (e.g., in the title bar, on the boundary, or on the resize tab) then your application will not receive any messages from Windows. So, for example, if you want to trap a mouse-down event on the resize tab so you can do something just prior to the window being resized, you will not be able to do this using the mouse event messages we ve discussed thus far. To detect such events, your application needs to handle Windows *non-client mouse event messages* (see Table 8-5).

Windows sends a non-client mouse event message whenever there is a mouse event (movement, click, double-click) within the window, but outside the client area. It is important to realize that Windows does not usually send your application any mouse messages when the mouse cursor is outside the application s window. Windows send non-client mouse event messages when the mouse cursor is in the non-client area *of your application s window* (e.g., the title bar and borders). The non-client mouse messages mirror the client area messages (see Table 8-5). By handling non-client area mouse messages, you application can check for events such as clicking the mouse on the resize section of the window s frame.

Table 8-5: Non-Client Area Mouse Message

Message	Description	lParam	wParam
w.WM_NCLBUTTONDBLCLK	Sent after the second click of a double-click sequence by the left mouse button.	L.O. word holds the mouse cursor's x-coordinate, H.O. word holds the cursor's y-coordinate. Note that these coordinates are screen coordinates, not client area coordinates.	Flags (bits) containing modifier key and mouse button status.
w.WM_NCLBUTTONDOWN	Sent when the user presses the left mouse button.		
w.WM_NCLBUTTONUP	Sent when the user releases the left mouse button.		
w.WM_NCMBUTTONDBLCLK	Sent after the second click of a double-click sequence by the middle mouse button.		
w.WM_NCMBUTTONDOWN	Sent when the user presses the middle mouse button		
w.WM_NCMBUTTONUP	Sent when the user releases the middle mouse button.		
w.WM_NCRBUTTONDBLCLK	Sent after the second click of a double-click sequence by the right mouse button.		
w.WM_NCRBUTTONDOWN	Sent when the user presses the right mouse button		
w.WM_NCRBUTTONUP	Sent when the user releases the right mouse button.		
w.WM_NCMOUSEMOVE	Sent whenever the user moves the mouse in the client area.		

An important difference between client area mouse messages and non-client area mouse messages is the value of the (x,y) coordinate that Windows passes in the `lParam` parameter. With normal client window mouse messages, these coordinates are always client-area coordinates. Coordinate (0,0) is the upper-left hand corner of the client area of the window. Obviously, this coordinate system does not apply to the non-client portion of the application s window. Therefore, Windows returns screen coordinates in the `lParam` parameter when it sends your application a non-client area mouse message. Fortunately, Windows provides a pair of API functions that let you easily convert between screen and client area coordinate systems:

```
w.ScreenToClient( hwnd, point );
w.ClientToScreen( hwnd, point );
```

As their names suggest, these functions take a point and translate its coordinates between the two different coordinate systems. The `hwnd` parameter is the handle of the window whose client area coordinate system you want to use, the point parameter is an object of type `w.POINT` (having a pair of int32 `x` and `y` fields). These calls replace the input values of the `point` parameter with the new, converted, value. It is very important that you realize that the screen-to-client conversion can produce coordinates whose values are negative. If a point is above or to the left of the client area of a window, then the w.ScreenToClient API will will produce negative values for the x and/or y coordinate values. This is why it is always important to treat screen coordinate values as signed integers.

Although the non-client mouse messages are useful for detecting events outside the client area, but still inside the application s window, there are times when an application needs to track the mouse outside the window area. For example, many drawing programs allow you to move objects beyond the current limits of the client area by clicking on an object, holding the mouse button down, dragging the object (beyond the bounds of the window), and then releasing the mouse button. By scrolling the window (or resizing the window) you can see where you ve moved the object.

Normally, this activity is not possible with Windows because Windows will stop sending your application mouse events when the mouse leaves the application s window. To overcome this limitation, Windows provides a facility known as mouse capture . Mouse capture tells Windows, under certain circumstances, to continue sending mouse messages to an application even if the mouse cursor leaves the window s boundaries. You have to explicitly request a mouse capture with the following API call:

```
w.SetCapture( hwnd );
```

The `hwnd` parameter is the handle of the window that is capturing the mouse. Windows will send all future mouse messages to the window procedure associated with this handle until the application calls the release API function:

```
w.ReleaseCapture;
```

For very practical reasons, you must only call `w.SetCapture` while the mouse button is depressed and you must call `w.ReleaseCapture` whenever the user releases the mouse button. The following application is a quick rectangle drawing application that demonstrates the use of `w.SetCapture` and `w.ReleaseCapture`. The *MouseCapture* application allows users to draw a rectangle in the window by pressing the mouse button, holding the button down, and then dragging the mouse cursor to some other point and releasing the button. Upon releasing the button, the application fills in the rectangle described by the two points. Figure 8-4 shows what the window looks like while dragging the mouse cursor across the window. Figure 8-5 shows the output when the user releases the mouse button. Figure 8-6 shows what the window looks like while the user is dragging the mouse cursor from the client area to a spot outside the window (when the user releases the button, the original rectangle will disappear, replaced by one that goes beyond the edge of the window). Note that by resizing the window, you can see the full rectangle that was originally drawn beyond the end of the window s boundaries.

Windows always returns client-window relative coordinates when you ve got the mouse captured. This means that the coordinates could very well be negative values (if you ve move the cursor to the left or above the client window). Once again, don t forget to treat the values that Windows returns in `lParam` as signed 16-bit integers or you may get incorrect results.

Figure 8-4: MouseCapture - Creating a Rectangle's Outline

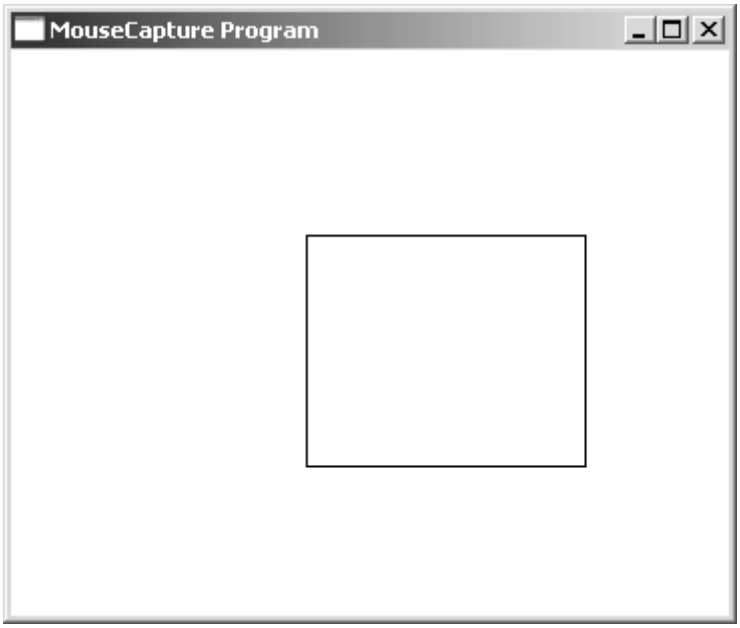


Figure 8-5: MouseCapture - Releasing the Mouse Button

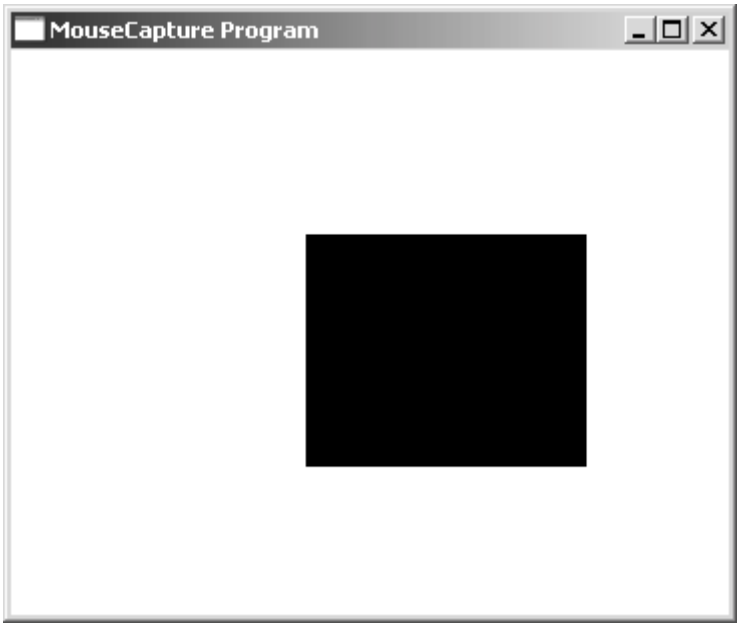
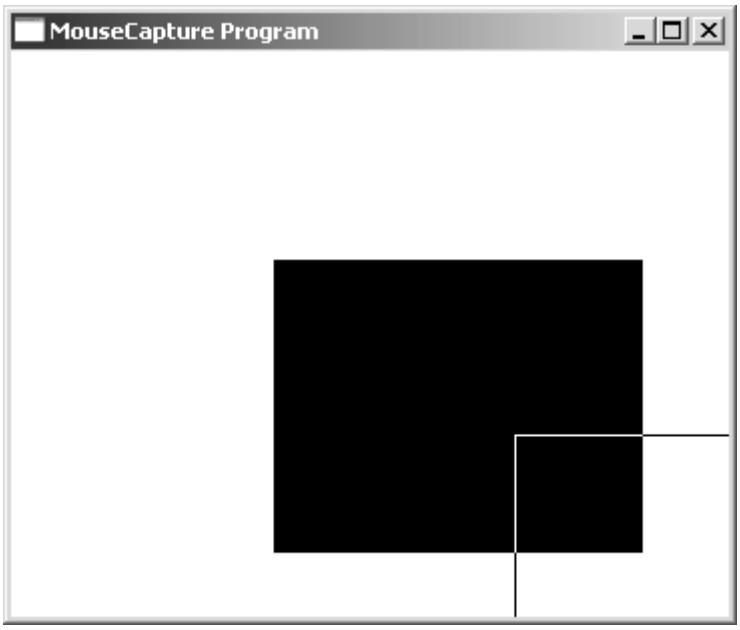


Figure 8-6: MouseCapture - Extending Beyond the End of the Window



Here s the source code for the *MouseCapture* program:

```
// MouseCapture.hla-
//
//  A program that demonstrates "capturing the mouse to track mouse positions
// outside the client area.
//
// Note: this is a unit because it uses the WinMail library module that
//       provides a win32 main program for us.

unit MouseCapture;

// Set the following to true to display interesting information
// about the bitmap file this program opens. You must be running
// the "DebugWindow" application for this output to appear.

?debug := false;

#includeonce( "hll.hhf" )
#includeonce( "w.hhf" )
#includeonce( "wpa.hhf" )
#includeonce( "winmain.hhf" )

?@NoDisplay := true;
?@NoStackAlign := true;

static
    ptBegin:    w.POINT;
    ptEnd:      w.POINT;
    ptBoxBegin: w.POINT;
    ptBoxEnd:   w.POINT;
```

```
    fBlocking:  boolean;
    fValidBox:  boolean;


readonly

    ClassName   :string := "MouseCaptureWinClass";      // Window Class Name
    AppCaption  :string := "MouseCapture Program";      // Caption for Window


    // The dispatch table:
    //
    //  This table is where you add new messages and message handlers
    //  to the program.  Each entry in the table must be a MsgProcPtr_t
    //  record containing two entries: the message value (a constant,
    //  typically one of the w.WM_***** constants found in windows.hhf)
    //  and a pointer to a "MsgProcPtr_t" procedure that will handle the
    //  message.


    Dispatch    :MsgProcPtr_t; @nostorage;

        MsgProcPtr_t
            MsgProcPtr_t:[  w.WM_DESTROY,        &QuitApplication    ],
            MsgProcPtr_t:[  w.WM_PAINT,          &Paint              ],
            MsgProcPtr_t:[  w.WM_MOUSEMOVE,      &MouseMove          ],
            MsgProcPtr_t:[  w.WM_LBUTTONDOWN,    &LButtonDown        ],
            MsgProcPtr_t:[  w.WM_LBUTTONUP,      &LButtonUp          ],

            // Insert new message handler records here.

            MsgProcPtr_t:[  0, NULL ];   // This marks the end of the list.



/*****************************************************************************/
/*          W I N M A I N   S U P P O R T   C O D E                          */
/*****************************************************************************/

// initWC - We don't have any initialization to do, so just return:

procedure initWC; @noframe;
begin initWC;

    ret();

end initWC;

// appCreateWindow- the default window creation code is fine, so just
//                  call defaultCreateWindow.

procedure appCreateWindow; @noframe;
begin appCreateWindow;

    jmp defaultCreateWindow;

end appCreateWindow;
```

```
// appException-
//
// Gives the application the opportunity to clean up before
// aborting when an unhandled exception comes along:

procedure appException( theException:dword in eax );
begin appException;

    raise( eax );

end appException;




// This is the custom message translation procedure.
// We're not doing any custom translation, so just return EAX=0
// to tell the caller to go ahead and call the default translation
// code.

procedure LocalProcessMsg( var lpmsg:w.MSG );
begin LocalProcessMsg;

    xor( eax, eax );

end LocalProcessMsg;


/**************************************************************************/
/*          A P P L I C A T I O N   S P E C I F I C   C O D E             */
/**************************************************************************/


// QuitApplication:
//
//  This procedure handles the w.WM_DESTROY message.
//  It tells the application to terminate.  This code sends
//  the appropriate message to the main program's message loop
//  that will cause the application to terminate.

procedure QuitApplication( hwnd: dword; wParam:dword; lParam:dword );
begin QuitApplication;

    // Tell the application to quit:

    w.PostQuitMessage( 0 );

end QuitApplication;


procedure drawBoxOutline( hwnd:dword; ptBegin:w.POINT; ptEnd:w.POINT ); @nodisplay;
var
    hdc:    dword;
begin drawBoxOutline;
```

```
    GetDC( hwnd, hdc );

        SetROP2( w.R2_NOT );
        SelectObject( w.GetStockObject( w.NULL_BRUSH ));
        Rectangle( ptBegin.x, ptBegin.y, ptEnd.x, ptEnd.y );

    ReleaseDC;

end drawBoxOutline;


// MouseMove:
//
//  This procedure handles the w.WM_MOUSEMOVE message (mouse movement within the client
window).

procedure MouseMove( hwnd: dword; wParam:dword; lParam:dword );
var
    hdc: dword;
begin MouseMove;

    if( fBlocking ) then

        w.SetCursor( w.LoadCursor( NULL, @string(w.IDC_CROSS) ));
        drawBoxOutline( hwnd, ptBegin, ptEnd );
        movsx( (type word lParam), eax );
        mov( eax, ptEnd.x );
        movsx( (type word lParam[2]), eax );
        mov( eax, ptEnd.y );
        drawBoxOutline( hwnd, ptBegin, ptEnd );

    endif;
    xor( eax, eax );

end MouseMove;




// LButtonDown:
//
//  This procedure handles the w.WM_LBUTTONDOWN message, which this
// program uses to capture the mouse.

procedure LButtonDown( hwnd: dword; wParam:dword; lParam:dword );
begin LButtonDown;

    movsx( (type word lParam), eax );
    mov( eax, ptBegin.x );
    mov( eax, ptEnd.x );
    movsx( (type word lParam[2]), eax );
    mov( eax, ptBegin.y );
    mov( eax, ptEnd.y );
    drawBoxOutline( hwnd, ptBegin, ptEnd );
    w.SetCapture( hwnd );
    w.SetCursor( w.LoadCursor( NULL, @string( w.IDC_CROSS)));
    mov( true, fBlocking );
```

```
    xor( eax, eax ); // Return zero to indicate success.

end LButtonDown;


// LButtonUp:
//
//  This procedure handles the w.WM_LBUTTONUP message which
// this program uses to release the mouse.

procedure LButtonUp( hwnd: dword; wParam:dword; lParam:dword );
begin LButtonUp;

    if( fBlocking ) then

        drawBoxOutline( hwnd, ptBegin, ptEnd );
        mov( ptBegin.x, eax );
        mov( eax, ptBoxBegin.x );
        mov( ptBegin.y, eax );
        mov( eax, ptBoxBegin.y );
        movsx( (type word lParam), eax );
        mov( eax, ptBoxEnd.x );
        movsx( (type word lParam[2]), eax );
        mov( eax, ptBoxEnd.y );
        w.ReleaseCapture();
        w.SetCursor( w.LoadCursor( NULL, @string(w.IDC_ARROW)));
        mov( false, fBlocking );
        mov( true, fValidBox );
        w.InvalidateRect( hwnd, NULL, true );

    endif;
    xor( eax, eax ); // Return zero to indicate success.

end LButtonUp;



// Paint:
//
//  This procedure handles the w.WM_PAINT message.

procedure Paint( hwnd: dword; wParam:dword; lParam:dword );

var
    hdc         :dword;         // Handle to video display device context.
    ps          :w.PAINTSTRUCT; // Used while painting text.

begin Paint;


    // Message handlers must preserve EBX, ESI, and EDI.
    // (They've also got to preserve EBP, but HLA's procedure
    // entry code already does that.)

    push( ebx );
    push( esi );
    push( edi );
```

```
    // Note that all GDI calls must appear within a
    // BeginPaint..EndPaint pair.

    BeginPaint( hwnd, ps, hdc );

        if( fValidBox ) then

            SelectObject( w.GetStockObject( w.BLACK_BRUSH ));
            Rectangle( ptBoxBegin.x, ptBoxBegin.y, ptBoxEnd.x, ptBoxEnd.y );
            if( fBlocking ) then

                SetROP2( w.R2_NOT );
                SelectObject( w.GetStockObject( w.NULL_BRUSH ));
                Rectangle( ptBoxBegin.x, ptBoxBegin.y, ptBoxEnd.x, ptBoxEnd.y );

            endif;

        endif;

    EndPaint;


    pop( edi );
    pop( esi );
    pop( ebx );

end Paint;
end MouseCapture;
```

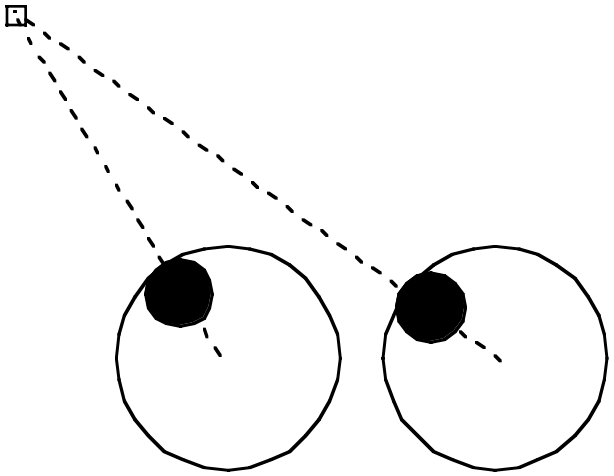
Listing 8-8: MouseCapture Source Code

There are a couple of things worth nothing in the MouseCapture source code. First of all, whenever you press a mouse button or move the mouse, this application draws an outline of the rectangle by first erasing the existing rectangle (if present) and then reframing that rectangle in the window. Erasure is accomplished by first noting whether the application has previously draw the rectangle s outline, and if so, then changing the pen copy mode to `w.R2_NOT` via the `SetROP2` call. NOT drawing the outline over the old one erases the outline. Once the outline is erased, the code draws a new outline using the new rectangle coordinates. When you first press the mouse button, the application captures the mouse cursor using the `w.SetCapture` API call. When you release the mouse button, the application releases the mouse capture using the `w.ReleaseCapture` API call and then fills in the rectangle by setting the device context s brush to `w.BLACK_BRUSH`.

MouseEyes is a second program that uses the `w.SetCapture` and `w.ReleaseCapture` to allow mouse activity outside the application s window. The idea here is somewhat whimsical - to use a pair of eyeballs in a window that help you pinpoint the mouse cursor anywhere on the screen. This first version of *MouseEyes* will not be very practical, it s intent is to demonstrate mouse capture and non-client area messages. However, we ll fix the practicality problems in a little bit.

The basic operation of *MouseEyes* is shown in Figure 8-7. Two round circles (the eyeballs) inside a pair of larger circles determine the position of the mouse cursor on the screen by looking at the cursor. With such a program running on your display, you can easily locate a small mouse cursor, even on a crowded or dimmed display, by simply looking in the direction where the mouse eyes are pointing.

Figure 8-7: MouseEyes Functionality

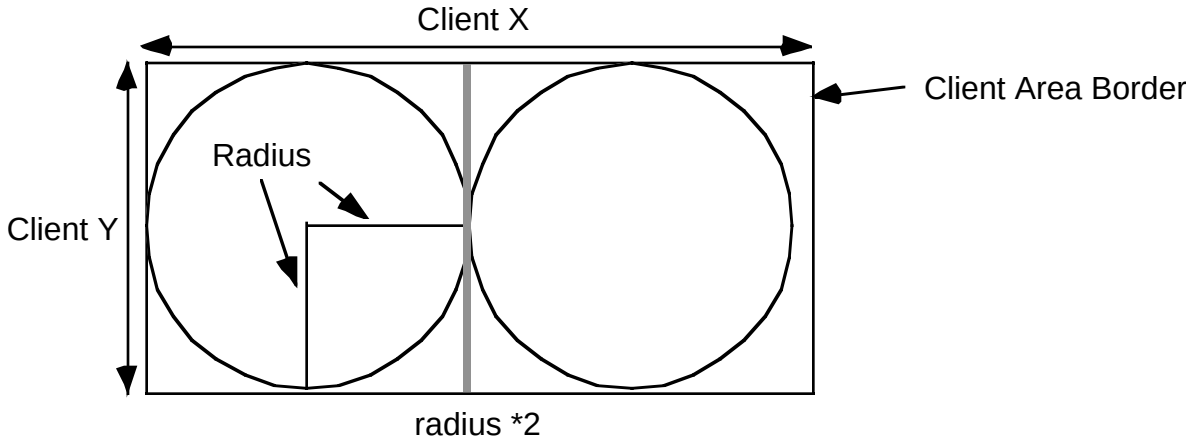


If *MouseEyes* were a real application, we d want the eyeballs to look in the direction of the mouse cursor, even when the current application doesn t have the input focus (meaning that the current application is not receiving mouse event messages). Indeed, in a little bit, we ll correct this oversight. For the time being, however, we re going to use the *MouseEyes* program to demonstrate non-client area mouse messages, so we don t want this additional functionality.

Most of the programs appearing in this chapter (indeed, in this book up to this point) have been fairly straight-forward. Most of the complex or new material has been learning the Win32 API. The *MouseEyes* application, however, requires a bit of geometry and trigonometry to pull off, so we need to spend a few moments discussing exactly how to draw the eyeballs.

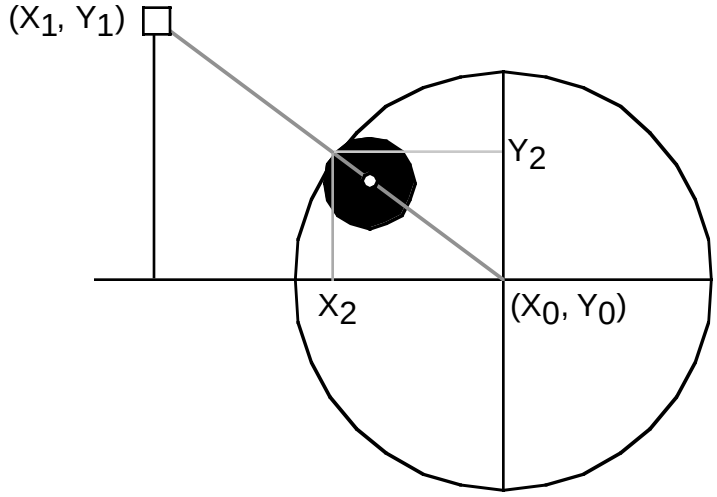
The first step is to draw the outer circles that represent the eyes. For simplicity, we will always draw circles rather than ellipses. The radius of the two circles representing the eyes will be the minimum of the height of the client area or the width of the client area, divided by two (divided by two because we will need to draw two of the eyes, side-by-side, in the window). The first circle will be draw using a bounding box of [(0,0), (radius*2,radius*2)] and the second circle will have the bounding box [(radius*2,0), (radius*4, radius*2)], see Figure 8-8.

Figure 8-8: Drawing the Two Eyes



Calculating the bounding box for the outer circles is relatively easy. The inner circles (the eyeballs) are much more difficult to deal with. We want to draw the inner circle eyeballs so that they just touch the inside of the outer circle. To see how to do this, consider Figure 8-9. We are given (X_1, Y_1), the current mouse cursor position, and (X_0, Y_0), the center of one of the outer circles. We need to compute two points, (X_2,Y_2), the point where the line described by $((X_0, Y_0), (X_1, Y_1))$ intersects the outer circle, and (X_3, Y_3), the centerpoint of the inner circle (not explicitly labelled in Figure 8-9).

Figure 8-9: Computing the Coordinates of the "Eyeball"



From geometry, we know that the ratio between R_1 (the distance from (X_0, Y_0) to (X_1, Y_1)) and R_2 (the distance from (X_0, Y_0) to (X_2, Y_2)) is the same as the ratio between (X_1-X_0) and (X_2-X_0). The same fact holds for the ratio between R_1/R_2 and $(Y_1-Y_0)/(Y_2-Y_0)$. We know X_0, Y_0, X_1, Y_1, and R_2 (the radius of the outer circle). We can easily compute R_1 using the formula:

$$R = \sqrt{X^2 + Y^2}$$

Given Y_1 (the y-coordinate of the mouse cursor), we can then compute Y_2 as follows:

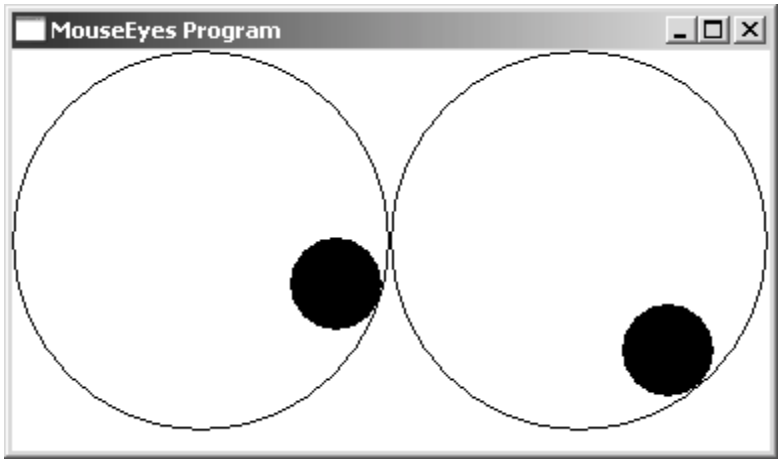
$$Y_2 = \frac{Y_1}{R_1}$$

Likewise, we can compute X_2 as follows:

$$X_2 = \frac{X_1}{R_1}$$

However, if we draw the circle centered around this point, our eyeball will overlap the outer circle; we want it just inside the circle. This correction is easily achieved by adjusting the bounding box inward by the radius of the smaller circle (that radius, by the way, is simply computed as one-eighth of the outer circle s radius in this program). The actual output of the *MouseEyes* program appears in Figure 8-10.

Figure 8-10: MouseEyes Output



The *MouseEyes* program updates the position of the eyeballs whenever it receives a mouse movement message. Normally, mouse movement messages are only sent to the application when a mouse event occurs within the client area - not very interesting for this application. Were you to move the mouse cursor into the non-client area (e.g., the title bar) the eyeballs would stop tracking the mouse. To correct this problem, and demonstrate non-client mouse messages (really, the whole purpose of this program), the MouseEyes application also

handles non-client area mouse messages so that the eyeballs still track the mouse cursor when it enters the non-client area of the window. Of course, once the mouse cursor leaves the application s window entirely, the application no longer receives mouse messages, so the eyeballs stop tracking. As yet another demonstration of mouse capture, the *MouseEyes* program will capture the mouse when you hold the mouse button down and it will track the mouse as long as you hold the button down (even if it leaves the application s window). This isn t great behavior for a true MouseEyes program, but it will suit our purposes for the time being.

Here s the full code to the *MouseEyes* application:

```
// MouseEyes.hla-
//
//  A program that demonstrates the use of the mouse and capturing the mouse,
// even in non-client areas.
//
// Note: this is a unit because it uses the WinMail library module that
//       provides a win32 main program for us.

unit MouseEyes;

// Set the following to true to display interesting information
// about the bitmap file this program opens. You must be running
// the "DebugWindow" application for this output to appear.

?debug := false;

#includeonce( "hll.hhf" )
#includeonce( "w.hhf" )
#includeonce( "wpa.hhf" )
#includeonce( "winmain.hhf" )
#includeonce( "math.hhf" )

?@NoDisplay := true;
?@NoStackAlign := true;

static
    MousePosn   :w.POINT;

readonly

    ClassName   :string := "MouseEyesWinClass";     // Window Class Name
    AppCaption  :string := "MouseEyes Program";     // Caption for Window


    // The dispatch table:
    //
    //  This table is where you add new messages and message handlers
    //  to the program.  Each entry in the table must be a MsgProcPtr_t
    //  record containing two entries: the message value (a constant,
    //  typically one of the w.WM_***** constants found in windows.hhf)
    //  and a pointer to a "MsgProcPtr_t" procedure that will handle the
    //  message.


    Dispatch    :MsgProcPtr_t; @nostorage;

        MsgProcPtr_t
```

```
            MsgProcPtr_t:[  w.WM_DESTROY,        &QuitApplication    ],
            MsgProcPtr_t:[  w.WM_PAINT,          &Paint              ],
            MsgProcPtr_t:[  w.WM_MOUSEMOVE,      &MouseMove          ],
            MsgProcPtr_t:[  w.WM_NCMOUSEMOVE,    &NCMouseMove        ],
            MsgProcPtr_t:[  w.WM_LBUTTONDOWN,    &LButtonDown        ],
            MsgProcPtr_t:[  w.WM_LBUTTONUP,      &LButtonUp          ],

            // Insert new message handler records here.

            MsgProcPtr_t:[  0, NULL ];   // This marks the end of the list.



/*****************************************************************************/
/*          W I N M A I N   S U P P O R T   C O D E                          */
/*****************************************************************************/

// initWC - We don't have any initialization to do, so just return:

procedure initWC; @noframe;
begin initWC;

    ret();

end initWC;

// appCreateWindow- the default window creation code is fine, so just
//                  call defaultCreateWindow.

procedure appCreateWindow; @noframe;
begin appCreateWindow;

    jmp defaultCreateWindow;

end appCreateWindow;


// appException-
//
// Gives the application the opportunity to clean up before
// aborting when an unhandled exception comes along:

procedure appException( theException:dword in eax );
begin appException;

    raise( eax );

end appException;




// This is the custom message translation procedure.
// We're not doing any custom translation, so just return EAX=0
// to tell the caller to go ahead and call the default translation
// code.
```

```
procedure LocalProcessMsg( var lpmsg:w.MSG );
begin LocalProcessMsg;

    xor( eax, eax );

end LocalProcessMsg;


/*****************************************************************************/
/*          A P P L I C A T I O N   S P E C I F I C   C O D E                */
/*****************************************************************************/


// QuitApplication:
//
//  This procedure handles the w.WM_DESTROY message.
//  It tells the application to terminate.  This code sends
//  the appropriate message to the main program's message loop
//  that will cause the application to terminate.

procedure QuitApplication( hwnd: dword; wParam:dword; lParam:dword );
begin QuitApplication;

    // Tell the application to quit:

    w.PostQuitMessage( 0 );

end QuitApplication;




// MouseMove:
//
//  This procedure handles the w.WM_MOUSEMOVE message (mouse movement within the client
window).

procedure MouseMove( hwnd: dword; wParam:dword; lParam:dword );
var
    hdc: dword;
begin MouseMove;

    movsx( (type word lParam), eax );
    mov( eax, MousePosn.x );
    movsx( (type word lParam[2]), eax );
    mov( eax, MousePosn.y );

    // Force a redraw of the window.

    w.InvalidateRect( hwnd, NULL, true );
    xor( eax, eax );

end MouseMove;
```

```
// NCMouseMove:
//
//  This procedure handles the w.WM_NCMOUSEMOVE message (mouse movement outside the client
window).

procedure NCMouseMove( hwnd: dword; wParam:dword; lParam:dword );
var
    hdc: dword;
begin NCMouseMove;

    // The coordinates we've received are screen coordinates:

    movsx( (type word lParam), eax );
    mov( eax, MousePosn.x );
    movsx( (type word lParam[2]), eax );
    mov( eax, MousePosn.y );

    // The coordinates we've received are screen window coordinates.
    // We need to convert them to client coordinates.

    w.ScreenToClient( hwnd, MousePosn );

    // Force a redraw of the window:

    w.InvalidateRect( hwnd, NULL, true );
    xor( eax, eax );

end NCMouseMove;


// LButtonDown:
//
//  This procedure handles the w.WM_LBUTTONDOWN message, which this
// program uses to capture the mouse.

procedure LButtonDown( hwnd: dword; wParam:dword; lParam:dword );
begin LButtonDown;

    w.SetCapture( hwnd );
    xor( eax, eax ); // Return zero to indicate success.

end LButtonDown;


// LButtonUp:
//
//  This procedure handles the w.WM_LBUTTONUP message which
// this program uses to release the mouse.

procedure LButtonUp( hwnd: dword; wParam:dword; lParam:dword );
begin LButtonUp;

    w.ReleaseCapture();
    xor( eax, eax ); // Return zero to indicate success.

end LButtonUp;
```

```
// Paint:
//
//  This procedure handles the w.WM_PAINT message.

procedure Paint( hwnd: dword; wParam:dword; lParam:dword );

var
    hdc         :dword;         // Handle to video display device context.
    ps          :w.PAINTSTRUCT; // Used while painting text.
    clientRect  :w.RECT;
    Diameter    :dword;
    smallRadius :int32;
    newRadius   :int32;
    eyeR        :int32;
    OuterX      :int32;
    OuterY      :int32;
    x           :int32;
    y           :int32;
    r           :real64;
    xSave       :real64;
    ySave       :real64;

begin Paint;


    // Message handlers must preserve EBX, ESI, and EDI.
    // (They've also got to preserve EBP, but HLA's procedure
    // entry code already does that.)

    push( ebx );
    push( esi );
    push( edi );

    // Note that all GDI calls must appear within a
    // BeginPaint..EndPaint pair.

    BeginPaint( hwnd, ps, hdc );

    // Get the coordinates of the client rectangle area so we know
    // where to draw our "eyes":

    w.GetClientRect( hwnd, clientRect );

        // Draw two circles within the window.
        // Draw the largest pair of circles that will
        // fit side-by-side in the window.
        // The largest diameter for the circles will be
        // min( clientRect.right/2, clientRect.bottom)
        // because the two circles are side-by-side.

        mov( clientRect.right, eax );
        shr( 1, eax );
        mov( clientRect.bottom, ebx );
        if( eax > ebx ) then
```

```
    mov( ebx, eax );

endif;

// EAX contains the diameter. Compute the center point (i.e., radius) of
// the (outer) circle by dividing this value in half. For the first circle,
// the x and y coordinates are both the same.

mov( eax, Diameter );
shr( 1, eax );
mov( eax, OuterX );
mov( eax, OuterY );

// Compute the radius of the smaller (eyeball) circle as 1/4th the larger Diameter.

shr( 2, eax );
mov( eax, smallRadius );
neg( eax );
add( OuterX, eax );
mov( eax, eyeR );

// Draw the two circles:

Ellipse( 0, 0, Diameter, Diameter );
mov( Diameter, eax );
shl( 1, eax );
Ellipse( Diameter, 0, eax, Diameter );

// Compute the coordinates for the "eyeball".
// r = sqrt( (MousePosn.x - OuterX)**2 + (MousePosn.y-OuterY)**2 )

fild( MousePosn.x );
fild( OuterX );
fsub;
fst( xSave );
fld( st0 );
fmul;
fild( MousePosn.y );
fild( OuterY );
fsub;
fst( ySave );
fld( st0 );
fmul;
fadd;
fsqrt;
fst( r );

// sin(theta) = y/r

fld( ySave );
fdivr;

// sin(theta) is on the stack now.
// Compute y-coordinate of intersection as y=rnew*sin(theta)
// where rnew is the radius of our circle and sin(theta) is from the above.

fild( eyeR );
```

```
fmul;
fistp( y );

// Compute the x-coordinate as x=rnew*cos(theta)
// cos(theta) = x/r

fld( xSave );
fld( r );
fdiv;
fild( eyeR );
fmul;
fistp( x );

// Draw a filled ellipse centered around this point:

SelectObject( w.GetStockObject( w.BLACK_BRUSH) );

mov( x, eax );          // The smaller circle's position
add( OuterX, eax );     // was computed relative to (0,0),
mov( eax, ebx );        // need to offset it to the (OuterX,OuterY)
mov( y, ecx );
add( OuterY, ecx );
mov( ecx, edx );

sub( smallRadius, eax );    // Move the smaller circle
sub( smallRadius, ecx );    // to just inside the larger one.
add( smallRadius, ebx );
add( smallRadius, edx );
Ellipse( eax, ecx, ebx, edx );


// Do the computations over for the second eyeball.
// Work is just about the same, the only difference
// is the fact that we have a different point as the
// center of the second eyeball:

mov( Diameter, eax );   // Compute the x-coordinate of the center
add( eax, OuterX );     // of the second eyeball.

fild( MousePosn.x );    // Everything from here on is the same.
fild( OuterX );
fsub;
fst( xSave );
fld( st0 );
fmul;
fild( MousePosn.y );
fild( OuterY );
fsub;
fst( ySave );
fld( st0 );
fmul;
fadd;
fsqrt;
fst( r );

// sin(theta) = y/r
```

```
        fld( ySave );
        fdivr;

        // sin(theta) is on the stack now.
        // Compute y-coordinate of intersection as y=rnew*sin(theta)
        // where rnew is the radius of our circle and sin(theta) is from the above.

        fild( eyeR );
        fmul;
        fistp( y );

        // Compute the x-coordinate as x=rnew*cos(theta)
        // cos(theta) = x/r

        fld( xSave );
        fld( r );
        fdiv;
        fild( eyeR );
        fmul;
        fistp( x );

        // Draw an ellipse centered around this point:

        SelectObject( w.GetStockObject( w.BLACK_BRUSH) );

        mov( x, eax );
        add( OuterX, eax );
        mov( eax, ebx );

        mov( y, ecx );
        add( OuterY, ecx );
        mov( ecx, edx );

        sub( smallRadius, eax );
        sub( smallRadius, ecx );
        add( smallRadius, ebx );
        add( smallRadius, edx );
        Ellipse( eax, ecx, ebx, edx );




    EndPaint;


    pop( edi );
    pop( esi );
    pop( ebx );

end Paint;
end MouseEyes;
```

Listing 8-9: The MouseEyes Application

To see the effects of the non-client messages, try commenting out the following line in the *MouseEyes* source code:

```
MsgProcPtr_t:[ w.WM_NCMOUSEMOVE,    &NCMouseMove       ],
```

i.e.,

```
Dispatch    :MsgProcPtr_t; @nostorage;

        MsgProcPtr_t
            MsgProcPtr_t:[  w.WM_DESTROY,         &QuitApplication    ],
            MsgProcPtr_t:[  w.WM_PAINT,           &Paint              ],
            MsgProcPtr_t:[  w.WM_MOUSEMOVE,       &MouseMove          ],
            // MsgProcPtr_t:[  w.WM_NCMOUSEMOVE,     &NCMouseMove          ],
            MsgProcPtr_t:[  w.WM_LBUTTONDOWN,     &LButtonDown        ],
            MsgProcPtr_t:[  w.WM_LBUTTONUP,       &LButtonUp          ],

            // Insert new message handler records here.

            MsgProcPtr_t:[  0, NULL ];   // This marks the end of the list.
```

Recompile and run this program with the non-client mouse message handler disabled. Now you ll find that the eyeballs stop tracking the mouse cursor when it enters the title bar and other non-client areas of the window. This quickly demonstrates the purpose of the non-client mouse messages. Of course, most applications won t need to process non-client mouse messages at all, but should you need to know when a mouse event occurs in a non-client area of the window, these messages provide you with everything you need.

Note that if you capture the mouse, you will continue to receive mouse event messages as standard mouse messages, even when the cursor is over a non-client area. You don t have to process non-client mouse messages if the only time you re worried about the non-client area is when you ve captured the mouse.

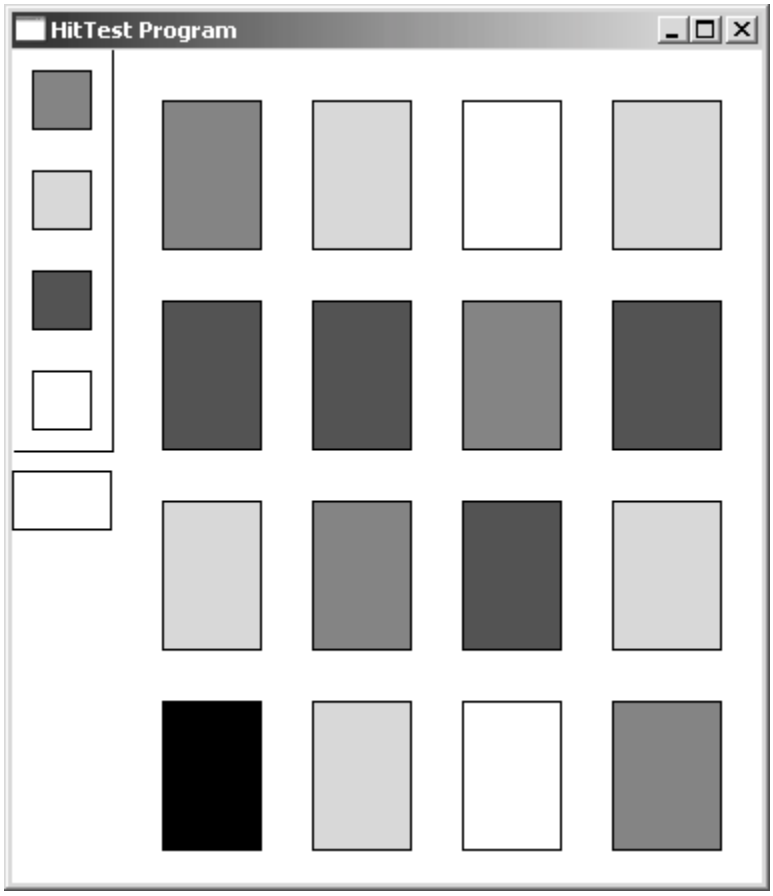
A very common problem in a modern Windows program is to determine if the mouse cursor is resting on a particular object whenever the user presses the mouse button. This is known as *hit testing*. Hit testing is actually a fairly simple process for regular objects (like rectangles) or objects whose hit test simply involves checking the cursor position to see if it lies within an object s bounding box (which is a rectangle, of course).

The *HitTest* application demonstrates this process. The HitTest program provides a palette of four colors on the left hand side of the window and a grid containing 16 rectangles in the main area of the window (see Figure 8-11). By clicking on one of the four color rectangles you can select a current color and then by clicking on one of the 16 rectangles in the main section of the window you can set that rectangle to the current color. This application uses the Windows API function w.PtInRect to do the actual hit testing. This function has the following prototype:

```
procedure w.PtInRect( rect:w.RECT; p:w.POINT );
```

This function returns true in EAX if the point is within the bounds of the specified rectangle, it returns false otherwise. The left button down message handling procedure first checks to see if the mouse button was pressed when the mouse was on one of the four set color rectangles in the HitTest application. If so, the code sets the current color and returns. If not, the code tests to see if the mouse is on top of one of the other rectangles in the display; if so, the code redraws that rectangle with the new color, if not, the button down message handler does nothing.

Figure 8-11: HitTest Output




```
// HitTest.hla-
//
//  A program that demonstrates testing a mouse press to see if it occurs within some
region.
//
// Note: this is a unit because it uses the WinMail library module that
//       provides a win32 main program for us.

unit HitTest;

// Set the following to true to display interesting information
// about the bitmap file this program opens. You must be running
// the "DebugWindow" application for this output to appear.

?debug := false;

#includeonce( "hll.hhf" )
#includeonce( "w.hhf" )
#includeonce( "wpa.hhf" )
#includeonce( "winmain.hhf" )

?@NoDisplay := true;
```

```
?@NoStackAlign := true;

const
    numColorBoxes := 4;
    numFillRects := 16;

type
    cSrc_t  :record

                r:  w.RECT;
                c:  w.COLORREF;

    endrecord;

static
    curColor    :w.COLORREF := RGB( $ff, $ff, $ff );
    ColorSrcs   :cSrc_t [ numColorBoxes] :=
                [
                    cSrc_t:[  w.RECT:[ 10, 10,40, 40], RGB(255,0,0)],
                    cSrc_t:[  w.RECT:[ 10, 60,40, 90], RGB(0,255,0)],
                    cSrc_t:[  w.RECT:[ 10,110,40,140], RGB(0,0,255)],
                    cSrc_t:[  w.RECT:[ 10,160,40,190], RGB(255,255,255)]
                ];

    RectsToFill :cSrc_t [ numFillRects] :=
                [
                    cSrc_t:[  w.RECT:[ 75,  25, 125, 100], RGB(0,0,0)],
                    cSrc_t:[  w.RECT:[ 150, 25, 200, 100], RGB(0,0,0)],
                    cSrc_t:[  w.RECT:[ 225, 25, 275, 100], RGB(0,0,0)],
                    cSrc_t:[  w.RECT:[ 300, 25, 355, 100], RGB(0,0,0)],

                    cSrc_t:[  w.RECT:[ 75,  125, 125, 200], RGB(0,0,0)],
                    cSrc_t:[  w.RECT:[ 150, 125, 200, 200], RGB(0,0,0)],
                    cSrc_t:[  w.RECT:[ 225, 125, 275, 200], RGB(0,0,0)],
                    cSrc_t:[  w.RECT:[ 300, 125, 355, 200], RGB(0,0,0)],

                    cSrc_t:[  w.RECT:[ 75,  225, 125, 300], RGB(0,0,0)],
                    cSrc_t:[  w.RECT:[ 150, 225, 200, 300], RGB(0,0,0)],
                    cSrc_t:[  w.RECT:[ 225, 225, 275, 300], RGB(0,0,0)],
                    cSrc_t:[  w.RECT:[ 300, 225, 355, 300], RGB(0,0,0)],

                    cSrc_t:[  w.RECT:[ 75,  325, 125, 400], RGB(0,0,0)],
                    cSrc_t:[  w.RECT:[ 150, 325, 200, 400], RGB(0,0,0)],
                    cSrc_t:[  w.RECT:[ 225, 325, 275, 400], RGB(0,0,0)],
                    cSrc_t:[  w.RECT:[ 300, 325, 355, 400], RGB(0,0,0)]
                ];

readonly

    ClassName   :string := "HitTestWinClass";       // Window Class Name
    AppCaption  :string := "HitTest Program";       // Caption for Window


    // The dispatch table:
    //
    //  This table is where you add new messages and message handlers
    //  to the program.  Each entry in the table must be a MsgProcPtr_t
```

```
    //  record containing two entries: the message value (a constant,
    //  typically one of the w.WM_***** constants found in windows.hhf)
    //  and a pointer to a "MsgProcPtr_t" procedure that will handle the
    //  message.


    Dispatch    :MsgProcPtr_t; @nostorage;

        MsgProcPtr_t
            MsgProcPtr_t:[  w.WM_DESTROY,        &QuitApplication    ],
            MsgProcPtr_t:[  w.WM_PAINT,          &Paint              ],
            MsgProcPtr_t:[  w.WM_LBUTTONDOWN,    &LButtonDown        ],

            // Insert new message handler records here.

            MsgProcPtr_t:[  0, NULL ];   // This marks the end of the list.



/*****************************************************************************/
/*          W I N M A I N   S U P P O R T   C O D E                          */
/*****************************************************************************/

// initWC - We don't have any initialization to do, so just return:

procedure initWC; @noframe;
begin initWC;

    ret();

end initWC;

// appCreateWindow- the default window creation code is fine, so just
//                  call defaultCreateWindow.

procedure appCreateWindow; @noframe;
begin appCreateWindow;

    jmp defaultCreateWindow;

end appCreateWindow;


// appException-
//
// Gives the application the opportunity to clean up before
// aborting when an unhandled exception comes along:

procedure appException( theException:dword in eax );
begin appException;

    raise( eax );

end appException;
```

```
// This is the custom message translation procedure.
// We're not doing any custom translation, so just return EAX=0
// to tell the caller to go ahead and call the default translation
// code.

procedure LocalProcessMsg( var lpmsg:w.MSG );
begin LocalProcessMsg;

    xor( eax, eax );

end LocalProcessMsg;


/*****************************************************************************/
/*          A P P L I C A T I O N   S P E C I F I C   C O D E                */
/*****************************************************************************/


// QuitApplication:
//
//  This procedure handles the w.WM_DESTROY message.
//  It tells the application to terminate.  This code sends
//  the appropriate message to the main program's message loop
//  that will cause the application to terminate.

procedure QuitApplication( hwnd: dword; wParam:dword; lParam:dword );
begin QuitApplication;

    // Tell the application to quit:

    w.PostQuitMessage( 0 );

end QuitApplication;




// LButtonDown:
//
//  This procedure handles the w.WM_LBUTTONDOWN message, which this
// program uses to capture the mouse.

procedure LButtonDown( hwnd: dword; wParam:dword; lParam:dword );
var
    p   :w.POINT;

begin LButtonDown;

    push( ebx );
    push( esi );
    movsx( (type word lParam), eax );
    mov( eax, p.x );
    movsx( (type word lParam[2]), eax );
    mov( eax, p.y );
    for( mov( 0, ebx ); ebx < numColorBoxes; inc( ebx )) do
```

```
        intmul( @size(cSrc_t), ebx, esi );
        w.PtInRect( ColorSrcs.r[ esi ], p );
        if( eax ) then

            mov( ColorSrcs.c[ esi], eax );
            mov( eax, curColor );

        endif;

    endfor;
    for( mov( 0, ebx ); ebx < numFillRects; inc( ebx )) do

        intmul( @size(cSrc_t), ebx, esi );
        w.PtInRect( RectsToFill.r[ esi ], p );
        if( eax ) then

            mov( curColor, eax );
            mov( eax, RectsToFill.c[ esi] );

        endif;

    endfor;


    w.InvalidateRect( hwnd, NULL, true );
    xor( eax, eax );
    pop( esi );
    pop( ebx );

end LButtonDown;




// Paint:
//
//  This procedure handles the w.WM_PAINT message.

procedure Paint( hwnd: dword; wParam:dword; lParam:dword );

var
    hbrush      :dword;
    hdc         :dword;         // Handle to video display device context.
    ps          :w.PAINTSTRUCT; // Used while painting text.
    r           :w.RECT;

begin Paint;


    // Message handlers must preserve EBX, ESI, and EDI.
    // (They've also got to preserve EBP, but HLA's procedure
    // entry code already does that.)

    push( ebx );
    push( esi );
    push( edi );
```

```
// Note that all GDI calls must appear within a
// BeginPaint..EndPaint pair.

BeginPaint( hwnd, ps, hdc );

    for( mov( 0, ebx ); ebx < numFillRects; inc( ebx )) do

        intmul( @size( cSrc_t ), ebx, esi );
        w.CreateSolidBrush( RectsToFill.c[ esi ] );
        mov( eax, hbrush );
        SelectObject( hbrush );
        FillRect( RectsToFill.r[ esi], hbrush );
        SelectObject( w.GetStockObject( w.NULL_BRUSH ));
        w.DeleteObject( hbrush );
        Rectangle
        (
            RectsToFill.r.left[ esi ],
            RectsToFill.r.top[ esi ],
            RectsToFill.r.right[ esi ],
            RectsToFill.r.bottom[ esi ]
        );

    endfor;

    for( mov( 0, ebx ); ebx < numColorBoxes; inc( ebx )) do

        intmul( @size( cSrc_t ), ebx, esi );
        w.CreateSolidBrush( ColorSrcs.c[ esi ] );
        mov( eax, hbrush );
        SelectObject( hbrush );
        FillRect( ColorSrcs.r[ esi], hbrush );
        SelectObject( w.GetStockObject( w.NULL_BRUSH ));
        w.DeleteObject( hbrush );
        Rectangle
        (
            ColorSrcs.r.left[ esi ],
            ColorSrcs.r.top[ esi ],
            ColorSrcs.r.right[ esi ],
            ColorSrcs.r.bottom[ esi ]
        );

    endfor;
    MoveTo( 50, 0 );
    LineTo( 50, 200 );
    LineTo( 0, 200 );

    // Display the currently selected color:

    w.SetRect( r, 0, 210, 50, 240 );
    w.CreateSolidBrush( curColor );
    mov( eax, hbrush );
    SelectObject( hbrush );
    FillRect( r, hbrush );
    SelectObject( w.GetStockObject( w.NULL_BRUSH ));
    w.DeleteObject( hbrush );
    Rectangle( 0, 210, 50, 240 );
```

```
    EndPaint;


    pop( edi );
    pop( esi );
    pop( ebx );

end Paint;
end HitTest;
```

Listing 8-10: HitTest Application

Limiting hit testing to rectangular regions isn t always appropriate. If you need to see if the user has pressed a mouse button within the confines of a complex object (only only within that object), you can always create a region from that object and then use the `w.PtInRegion` API function to check for a hit:

```
procedure w.PtInRegion( hrgn:dword; x:int32; y:int32 );
```

This procedure returns true if the point sepcified by (x,y) is within the region whose handle you pass as the first parameter.

Newer versions of Windows (2000, XP, and later) provide additional mouse events for newer-style Microsoft mice as well as support for tool-tip messages. Specfically, there are three new messages that support XBUTTON down, up, and double-clicks (X-buttons are additional buttons on newer mice). There is also a w.TrackMouseEvent function that activates two other messages: w.WM_MOUSEHOVER and w.WM_MOUSELEAVE, that Windows sends when the mouse cursor hovers above a certain object for some length of time (e.g., applications use this to bring up little tool tip windows). Check out the Microsoft MSDN documentation (on-line) for more details on these messages.

8.11: Timer Messages

The *MouseEyes* application suffers from a very major problem: it only tracks the mouse cursor while the mouse cursor is actually within that application s window (or while you re holding down the mouse button if you pressed the mouse button while inside the window). For this application to be practical, it needs to track the mouse cursor position even when it is not receiving mouse event message (i.e., when some other window has the input focus and is receiving the mouse messages). Sometimes an application (like *MouseEyes*) needs some periodic messages to come along so that it can do some work (such as track the current cursor position) even though the window doesn t have the focus and isn t receiving any other messages. Windows timer messages can solve this problem.

A timer message is a message that Windows can send to your application on a periodic basis. Therefore, even if no other events occur (because the application is in the background and never gets the input focus), the application can receive the attention of the CPU so it can take care of some processing needs. The MouseEyes application is a classic example, it needs to periodically check the cursor position so it can update its eyeballs, even though it s not receiving a stream of mouse messages (because those messages are headed to another window). You can also use timers to implement various timeout or timing features in an application such as periodically updating status information, autosaving files every few minutes, or controlling how long an application runs before it terminates (e.g., for demo versions of an application).

To receive timer messages, you must explicitly request that Windows do this. You accomplish this with the w.SetTimer API call:

```
   procedure w.SetTimer
   (
      hwnd:dword;
      timerID:dword;
      timeout:uns32;
      timerproc: w.TIMERPROC
   );

type
   TIMERPROC:
      procedure( hwnd:dword; uMsg:dword; idEvent:dword; dwTime:dword );
```

The first parameter to w.SetTimer is the handle of the window procedure that is going to receive the w.WM_TIMER messages that Windows will generate in response to this request. In this situation, the last parameter (timerproc) should be NULL. The second parameter specifies a non-zero *timer ID*. An application can activate multiple timers; it differentiates those timers by specifying differing numeric values for the timer ID. Whenever an application receives a w.WM_TIMER message, the wParam parameter contains the timer ID value for the timer that triggered the message. The third parameter is the timeout period, in milliseconds. The last parameter is the address of a *callback routine*; this parameter should be NULL if you want Windows to pass the message through to the window procedure you specify by the hwnd handle. If you prefer, you can tell Windows to call a procedure of your choosing (compatible with the w.TIMERPROC type). The advantage of the callback procedure is that you get more information in the parameter list that Windows passes to the callback procedure.

You can kill a timer by calling the w.KillTimer API function:

```
procedure w.KillTimer( hwnd:dword; timerID:dword );
```

The hwnd and timerID values must match the values you passed to the corresponding w.SetTimer call. As timers are system-wide resources, you must ensure that you kill all timers you start in your application. In general, it s a good idea to start all timers in your Create procedure and kill them in your Destroy procedure, assuming your application allows this. This avoids paths through the application that start timers without ever killing them.

To conclude this chapter, we ll fix the aforementioned problems with the *MouseEyes* application so that it runs off of a timer event rather than mouse events. The *MouseEyes2* application arms a timer to send it a message every 150ms. When a timer message arrives, the *MouseEyes2* application gets the current mouse cursor position via the w.GetCursorPos API call:

```
procedure w.GetCursorPos( var point:w.POINT );
```

This function stores the current mouse position into the point parameter you pass by reference. Note that this function returns screen coordinates, not *MouseEyes2* client area coordinates. Therefore, the application will have to convert those coordinates to client area coordinates before using them. After getting these coordinates from Windows, *MouseEyes2* draws the eyeballs exactly as in the *MouseEyes* program. Here s the code:

```
// MouseEyes.hla-
//
//  A program that demonstrates the use of the mouse and capturing the mouse,
// even in non-client areas.
//
// Note: this is a unit because it uses the WinMail library module that
//       provides a win32 main program for us.
```

```
unit MouseEyes2;

// Set the following to true to display interesting information
// about the bitmap file this program opens. You must be running
// the "DebugWindow" application for this output to appear.

?debug := false;

#includeonce( "hll.hhf" )
#includeonce( "w.hhf" )
#includeonce( "wpa.hhf" )
#includeonce( "winmain.hhf" )
#includeonce( "math.hhf" )

?@NoDisplay := true;
?@NoStackAlign := true;

static
    MousePosn   :w.POINT;

readonly

    ClassName   :string := "MouseEyesWinClass";     // Window Class Name
    AppCaption  :string := "MouseEyes Program";     // Caption for Window


    // The dispatch table:
    //
    //  This table is where you add new messages and message handlers
    //  to the program.  Each entry in the table must be a MsgProcPtr_t
    //  record containing two entries: the message value (a constant,
    //  typically one of the w.WM_***** constants found in windows.hhf)
    //  and a pointer to a "MsgProcPtr_t" procedure that will handle the
    //  message.


    Dispatch    :MsgProcPtr_t; @nostorage;

        MsgProcPtr_t
            MsgProcPtr_t:[  w.WM_CREATE,         &Create            ],
            MsgProcPtr_t:[  w.WM_DESTROY,        &QuitApplication   ],
            MsgProcPtr_t:[  w.WM_PAINT,          &Paint             ],
            MsgProcPtr_t:[  w.WM_TIMER,          &TimerMsg          ],

            // Insert new message handler records here.

            MsgProcPtr_t:[  0, NULL ];   // This marks the end of the list.



/*****************************************************************/
/*          W I N M A I N   S U P P O R T   C O D E              */
/*****************************************************************/

// initWC - We don't have any initialization to do, so just return:
```

```
procedure initWC; @noframe;
begin initWC;

    ret();

end initWC;

// appCreateWindow- the default window creation code is fine, so just
//                  call defaultCreateWindow.

procedure appCreateWindow; @noframe;
begin appCreateWindow;

    jmp defaultCreateWindow;

end appCreateWindow;


// appException-
//
// Gives the application the opportunity to clean up before
// aborting when an unhandled exception comes along:

procedure appException( theException:dword in eax );
begin appException;

    raise( eax );

end appException;




// This is the custom message translation procedure.
// We're not doing any custom translation, so just return EAX=0
// to tell the caller to go ahead and call the default translation
// code.

procedure LocalProcessMsg( var lpmsg:w.MSG );
begin LocalProcessMsg;

    xor( eax, eax );

end LocalProcessMsg;


/*****************************************************************************/
/*          A P P L I C A T I O N   S P E C I F I C   C O D E                */
/*****************************************************************************/


// Create:
//
//  Called when we first create the window. Initializes the timer.

procedure Create( hwnd: dword; wParam:dword; lParam:dword );
begin Create;
```

```
    // Turn off the timer:

    w.SetTimer( hwnd, 1, 150, NULL );
    xor( eax, eax );

end Create;



// QuitApplication:
//
//  This procedure handles the w.WM_DESTROY message.
//  It tells the application to terminate.  This code sends
//  the appropriate message to the main program's message loop
//  that will cause the application to terminate.

procedure QuitApplication( hwnd: dword; wParam:dword; lParam:dword );
begin QuitApplication;

    // Turn off the timer:

    w.KillTimer( hwnd, 1 );

    // Tell the application to quit:

    w.PostQuitMessage( 0 );

end QuitApplication;




// MouseMove:
//
//  This procedure handles the w.WM_MOUSEMOVE message (mouse movement within the client
window).

procedure TimerMsg( hwnd: dword; wParam:dword; lParam:dword );
begin TimerMsg;

    // Get the current cursor position:

    w.GetCursorPos( MousePosn );
    w.ScreenToClient( hwnd, MousePosn );

    // Force a redraw of the window.

    w.InvalidateRect( hwnd, NULL, true );
    xor( eax, eax );

end TimerMsg;




// Paint:
```

```
//
//  This procedure handles the w.WM_PAINT message.

procedure Paint( hwnd: dword; wParam:dword; lParam:dword );

var
    hdc         :dword;         // Handle to video display device context.
    ps          :w.PAINTSTRUCT; // Used while painting text.
    clientRect  :w.RECT;
    Diameter    :dword;
    smallRadius :int32;
    newRadius   :int32;
    eyeR        :int32;
    OuterX      :int32;
    OuterY      :int32;
    x           :int32;
    y           :int32;
    r           :real64;
    xSave       :real64;
    ySave       :real64;

begin Paint;


    // Message handlers must preserve EBX, ESI, and EDI.
    // (They've also got to preserve EBP, but HLA's procedure
    // entry code already does that.)

    push( ebx );
    push( esi );
    push( edi );

    // Note that all GDI calls must appear within a
    // BeginPaint..EndPaint pair.

    BeginPaint( hwnd, ps, hdc );


        // Get the coordinates of the client rectangle area so we know
        // where to draw our "eyes":

        w.GetClientRect( hwnd, clientRect );

        // Draw two circles within the window.
        // Draw the largest pair of circles that will
        // fit side-by-side in the window.
        // The largest diameter for the circles will be
        // min( clientRect.right/2, clientRect.bottom)
        // because the two circles are side-by-side.

        mov( clientRect.right, eax );
        shr( 1, eax );
        mov( clientRect.bottom, ebx );
        if( eax > ebx ) then

            mov( ebx, eax );
```

```
endif;

// EAX contains the diameter. Compute the center point (i.e., radius) of
// the (outer) circle by dividing this value in half. For the first circle,
// the x and y coordinates are both the same.

mov( eax, Diameter );
shr( 1, eax );
mov( eax, OuterX );
mov( eax, OuterY );

// Compute the radius of the smaller (eyeball) circle as 1/4th the larger Diameter.

shr( 2, eax );
mov( eax, smallRadius );
neg( eax );
add( OuterX, eax );
mov( eax, eyeR );

// Draw the two circles:

Ellipse( 0, 0, Diameter, Diameter );
mov( Diameter, eax );
shl( 1, eax );
Ellipse( Diameter, 0, eax, Diameter );

// Compute the coordinates for the "eyeball".
// r = sqrt( (MousePosn.x - OuterX)**2 + (MousePosn.y-OuterY)**2 )

fild( MousePosn.x );
fild( OuterX );
fsub;
fst( xSave );
fld( st0 );
fmul;
fild( MousePosn.y );
fild( OuterY );
fsub;
fst( ySave );
fld( st0 );
fmul;
fadd;
fsqrt;
fst( r );

// sin(theta) = y/r

fld( ySave );
fdivr;

// sin(theta) is on the stack now.
// Compute y-coordinate of intersection as y=rnew*sin(theta)
// where rnew is the radius of our circle and sin(theta) is from the above.

fild( eyeR );
fmul;
fistp( y );
```

```
// Compute the x-coordinate as x=rnew*cos(theta)
// cos(theta) = x/r

fld( xSave );
fld( r );
fdiv;
fild( eyeR );
fmul;
fistp( x );

// Draw a filled ellipse centered around this point:

SelectObject( w.GetStockObject( w.BLACK_BRUSH) );

mov( x, eax );          // The smaller circle's position
add( OuterX, eax );     // was computed relative to (0,0),
mov( eax, ebx );        // need to offset it to the (OuterX,OuterY)
mov( y, ecx );
add( OuterY, ecx );
mov( ecx, edx );

sub( smallRadius, eax );    // Move the smaller circle
sub( smallRadius, ecx );    // to just inside the larger one.
add( smallRadius, ebx );
add( smallRadius, edx );
Ellipse( eax, ecx, ebx, edx );


// Do the computations over for the second eyeball.
// Work is just about the same, the only difference
// is the fact that we have a different point as the
// center of the second eyeball:

mov( Diameter, eax );   // Compute the x-coordinate of the center
add( eax, OuterX );     // of the second eyeball.

fild( MousePosn.x );    // Everything from here on is the same.
fild( OuterX );
fsub;
fst( xSave );
fld( st0 );
fmul;
fild( MousePosn.y );
fild( OuterY );
fsub;
fst( ySave );
fld( st0 );
fmul;
fadd;
fsqrt;
fst( r );

// sin(theta) = y/r

fld( ySave );
fdivr;
```

```
        // sin(theta) is on the stack now.
        // Compute y-coordinate of intersection as y=rnew*sin(theta)
        // where rnew is the radius of our circle and sin(theta) is from the above.

        fild( eyeR );
        fmul;
        fistp( y );

        // Compute the x-coordinate as x=rnew*cos(theta)
        // cos(theta) = x/r

        fld( xSave );
        fld( r );
        fdiv;
        fild( eyeR );
        fmul;
        fistp( x );

        // Draw an ellipse centered around this point:

        SelectObject( w.GetStockObject( w.BLACK_BRUSH) );

        mov( x, eax );
        add( OuterX, eax );
        mov( eax, ebx );

        mov( y, ecx );
        add( OuterY, ecx );
        mov( ecx, edx );

        sub( smallRadius, eax );
        sub( smallRadius, ecx );
        add( smallRadius, ebx );
        add( smallRadius, edx );
        Ellipse( eax, ecx, ebx, edx );




    EndPaint;


    pop( edi );
    pop( esi );
    pop( ebx );

end Paint;
end MouseEyes2;
```

Listing 8-12: MouseEyes2 Application

8.13: But Wait! There's More!

As long as this chapter is, it isn t the final word on Windows input by any stretch of the imagination. Rather, it s just an introduction to the subject. Of course, completely missing from the discussion in this chapter are input issues such as reading text from textedit boxes and input from other controls. Fear not, we re getting to that before too much longer.