Tutorial 7: Mouse Input

This win32 tutorial was created and written by Iczelion for MASM32. It was translated for use by HLA (High Level Assembly) users by Randall Hyde. All original copyrights and other issues still apply to this text. The following is the copyright notice from Iczelion's Win32 Assembly Home Page:

The tutorials written by me are copyright freeware. That means they are available freely so long as they are not included in any commercial package. Commercial use is strictly prohibited. "Knowledge, like sex, is better when it's free"

Note that I don't claim to be the win32asm wizard or a coding guru. I'm also learning my ropes. Those tutorials were written as reminders of what I have learned. They will grow in number as I learn more about win32asm programming.

You can read more about Iczelion's tutorials at the "Iczelion's Win32 Assembly Home Page" found at

http://win32asm.cjb.net

That site provides the original MASM examples as well as providing additional win32 assembly language programming information. Note that the MASM tutorials provide an excellent contrast between MASM and HLA as you can see the differences between these two languages since MASM code exists at Iczelion's site and the HLA translation appears at this site.

Note that references to the first person ("I") refer to Iczelion, not Randall Hyde. Randy Hyde has attempted to maintain the tutorial in as "pure" a state as possible, only making the modifications necessary to support HLA rather than MASM along with a few minor changes to the English. All credit, glory, damnation, etc., is due Iczelion; Randall Hyde's modifications to this tutorial were rather trivial in nature.

Tutorial 7: Mouse Input

We will learn how to receive and respond to mouse input in our window procedure. The example program will wait for left mouse clicks and display a text string at the exact clicked spot in the client area.

Source Code for this Tutorial:

```
// Iczelion's tutorial #7: Mouse Input

program aSimpleWindow;
#include( "win32.hhf" )     // Standard windows stuff.
#include( "strings.hhf" )   // Defines HLA string routines.
#include( "memory.hhf" )    // Defines "NULL" among other things.
#include( "args.hhf" )      // Command line parameter stuff.
```

```
#include( "conv.hhf" )


static
    hInstance:      dword;
    CommandLine:    string;

const
    AppNameStr  := "Our First Window";
readonly

    ClassName:  string := "SimpleWinClass";
    AppName:    string := AppNameStr;



static GetLastError:procedure; external( "__imp__GetLastError@0" );



// The window procedure.  Since this gets called directly from
// windows we need to explicitly reverse the parameters (compared
// to the standard STDCALL declaration) in order to make HLA's
// Pascal calling convention compatible with Windows.
//
// This is actually a function that returns a return result in
// EAX.  If this function returns zero in EAX, then the event
// loop terminates program execution.

procedure WndProc( lParam:dword; wParam:dword; uMsg:uns32; hWnd:dword );
    nodisplay;

const
    TestString:= "Win32 assembly is great and easy!";

var
    hdc:        dword;
    ps:         win.PAINTSTRUCT;


static
    mouseClick:     boolean := false;
    hitPoint:       win.POINT;

begin WndProc;


    // If the WM_DESTROY message comes along, then we've
    // got to post a message telling the event loop that
    // it's time to quit the program.  The return value in
    // EAX must be false (zero).  The GetMessage function
    // will return this value to the event loop which is
    // the indication that it's time to quit.
```

```
if( uMsg = win.WM_DESTROY ) then

    win.PostQuitMessage( 0 );


elseif( uMsg = win.WM_LBUTTONDOWN ) then

    movzx( (type word lParam), eax );
    mov( eax, hitPoint.x );
    movzx( (type word lParam[2]), eax );
    mov( eax, hitPoint.y );

    mov( true, mouseClick );
    win.InvalidateRect( hWnd, NULL, true );


elseif( uMsg = win.WM_PAINT ) then

    // When Windows requests that we draw the window,
    // fill in the string in the center of the screen.

    win.BeginPaint( hWnd, ps );
    mov( eax, hdc );

    if( mouseClick ) then


        win.TextOut
        (
            hdc,
            hitPoint.x,
            hitPoint.y,
            AppName,
            @length( AppNameStr )
        );
        mov( false, mouseClick );

    endif;
    win.EndPaint( hWnd, ps );



else

    // If a WM_DESTROY message doesn't come along,
    // let the default window handler process the
    // message.  Whatever (non-zero) value this function
    // returns is the return result passed on to the
    // event loop.

    win.DefWindowProc( hWnd, uMsg, wParam, lParam );
    exit WndProc;

endif;
```

```
    sub( eax, eax );

end WndProc;



// WinMain-
//
// This is the "main" windows program.  It sets up the
// window and then enters an "event loop" processing
// whatever messages are passed along to that window.
// Since our code is the only code that calls this function,
// we'll use the Pascal calling conventions for the parameters.

procedure WinMain
(
    hInst:dword;
    hPrevInst:  dword;
    CmdLine:    string;
    CmdShow:    dword
);  nodisplay;

var
    wc:     win.WNDCLASSEX;
    msg:    win.MSG;
    hwnd:   dword;

begin WinMain;


    // Set up the window class (wc) object:

    mov( @size( win.WNDCLASSEX ), wc.cbSize );
    mov( win.CS_HREDRAW | win.CS_VREDRAW, wc.style );
    mov( &WndProc, wc.lpfnWndProc );
    mov( NULL, wc.cbClsExtra );
    mov( NULL, wc.cbWndExtra );

    mov( hInstance, wc.hInstance );
    mov( win.COLOR_WINDOW+1, wc.hbrBackground );
    mov( NULL, wc.lpszMenuName );
    mov( ClassName, wc.lpszClassName );

    // Get the icons and cursor for this application:

    win.LoadIcon( NULL, win.IDI_APPLICATION );
    mov( eax, wc.hIcon );
    mov( eax, wc.hIconSm );

    win.LoadCursor( NULL, win.IDC_ARROW );
    mov( eax, wc.hCursor );


    // Okay, register this window with Windows so it
```

```
    // will start passing messages our way.  Once this
    // is accomplished, create the window and display it.

    win.RegisterClassEx( wc );


    win.CreateWindowEx
    (
        NULL,
        ClassName,
        AppName,
        win.WS_OVERLAPPEDWINDOW,
        win.CW_USEDEFAULT,
        win.CW_USEDEFAULT,
        win.CW_USEDEFAULT,
        win.CW_USEDEFAULT,
        NULL,
        NULL,
        hInst,
        NULL
    );
    mov( eax, hwnd );


    win.ShowWindow( hwnd, win.SW_SHOWNORMAL );
    win.UpdateWindow( hwnd );

    // Here's the event loop that processes messages
    // sent to our window.  On return from GetMessage,
    // break if EAX contains false and quit the
    // program.

    forever

        win.GetMessage( msg, NULL, 0, 0 );
        breakif( !eax );
        win.TranslateMessage( msg );
        win.DispatchMessage( msg );

    endfor;
    mov( msg.wParam, eax );


end WinMain;


begin aSimpleWindow;

    // Get this process' handle:

    win.GetModuleHandle( NULL );
    mov( eax, hInstance );

    // Get a copy of the command line string passed to this code:
```

```
    mov( arg.CmdLn(), CommandLine );



    WinMain( hInstance, NULL, CommandLine, win.SW_SHOWDEFAULT );

    // WinMain returns a return code in EAX, exit the program
    // and pass along that return code.

    win.ExitProcess( eax );

end aSimpleWindow;
```

Theory:

As with keyboard input, Windows detects and sends notifications about mouse activities that are relevant to each window. Those activities include left and right clicks, mouse cursor movement over window, double clicks. Unlike keyboard input which is directed to the window that has input focus, mouse messages are sent to any window that the mouse cursor is over, active or not. In addition, there are mouse messages about the non-client area too. But most of the time, we can blissfully ignore them. We can focus on those relating to the client area.

There are two messages for each mouse button: win.WM_LBUTTONDOWN,win.WM_RBUTTONDOWN and win.WM_LBUTTONUP, win.WM_RBUTTONUP messages. For a mouse with three buttons, there are also win.WM_MBUTTONDOWN and win.WM_MBUTTONUP. When the mouse cursor moves over the client area, Windows sends win.WM_MOUSEMOVE messages to the window under the cursor.

A window can receive double click messages, win.WM_LBUTTONDBCLK or win.WM_RBUTTONDBCLK, if and only if its window class has win.CS_DBLCLKS style flag, else the window will receive only a series of mouse button up and down messages.

For all these messages, the value of lParam contains the position of the mouse. The low word is the x-coordinate, and the high word is the y-coordinate relative to upper left corner of the client area of the window. wParam indicates the state of the mouse buttons and Shift and Ctrl keys.

Analysis:

```
   elseif( uMsg = win.WM_LBUTTONDOWN ) then

        movzx( (type word lParam), eax );
        mov( eax, hitPoint.x );
        movzx( (type word lParam[2]), eax );
        mov( eax, hitPoint.y );
```

```
        mov( true, mouseClick );
        win.InvalidateRect( hWnd, NULL, true );
```

The window procedure waits for left mouse button click. When it receives win.WM_LBUTTONDOWN, lParam contains the coordinate of the mouse cursor in the client area. It saves the coordinate in a variable of type win.POINT which is defined as:

```
POINT:record
    x:dword;
    y:dword;
endrecord;
```

and sets the flag, MouseClick, to TRUE, meaning that there's at least a left mouse button click in the client area.

```
        movzx( (type word lParam), eax );
        mov( eax, hitPoint.x );
```

Since x-coordinate is the low word of lParam and the members of win.POINT structure are 32-bit in size, we have to zero out the high word of eax prior to storing it in hitpoint.x.

```
        movzx( (type word lParam[2]), eax );
        mov( eax, hitPoint.y );
```

Because y-coordinate is the high word of lParam, we must put it in the low word of eax prior to storing it in hitpoint.y.

After storing the mouse position, we set the flag, MouseClick, to TRUE in order to let the painting code in win.WM_PAINT section know that there's at least a click in the client area so it can draw the string at the mouse position. Next we call win.InvalidateRect function to force the window to repaint its entire client area.

```
        if( mouseClick ) then

            win.TextOut
            (
                hdc,
                hitPoint.x,
                hitPoint.y,
                AppName,
                @length( AppNameStr )
            );
            mov( false, mouseClick );

        endif;
```

The painting code in win.WM_PAINT section must check if MouseClick is true, since when the window was created, it received a win.WM_PAINT message which at that time, no mouse click had occurred so it should not draw the string in the client area. We initialize MouseClick to FALSE and change its value to TRUE when an actual mouse click occurs.

If at least one mouse click has occurred, it draws the string in the client area at the mouse position. Note that it calls lstrlen to get the length of the string to display and sends the length as the last parameter of win.TextOut function.