Tutorial 6: Keyboard 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:

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 6: Keyboard Input

We will learn how a Windows program receives keyboard input.

Source Code for this Tutorial:

```
// Iczelion's tutorial #6: Keyboard 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;

readonly

    ClassName:  string := "SimpleWinClass";
    AppName:    string := "Our First Window";



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
    kbdChar:    char := ’ ’;

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_CHAR ) then

        // When a keyboard character comes along, save
```

```
            // the ASCII code of the character and invalidate
            // the window’s rectangle so that we can force
            // a redraw.

            mov( (type byte wParam ), al );
            mov( al, kbdChar );
            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 );


            win.TextOut
            (
                hdc,
                0,
                0,
                #{ lea( eax, kbdChar ); push( eax ); }#,
                1
            );
            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:

Since normally there’s only one keyboard in each PC, all running Windows programs must share it between them. Windows is responsible for sending the key strokes to the window which has the input focus.

Although there may be several windows on the screen, only one of them has the input focus. The window which has input focus is the only one which can receive key strokes. You can differentiate the window which has input focus from other windows by looking at the title bar. The title bar of the window which has input focus is highlighted.

Actually, there are two main types of keyboard messages, depending on your view of the keyboard. You can view a keyboard as a collection of keys. In this case, if you press a key, Windows sends a win.WM_KEYDOWN message to the window which has input focus, notifying that a key is pressed. When you release the key, Windows sends a win.WM_KEYUP message. You treat a key as a button. Another way to look at the keyboard is that it’s a character input device. When you press "a" key, Windows sends a win.WM_CHAR message to the window which has input focus, telling it that the user sends "a" character to it. In fact, Windows sends win.WM_KEYDOWN and win.WM_KEYUP messages to the window which has input focus and those messages will be translated to win.WM_CHAR messages by win.TranslateMessage call. The window procedure may decide to process all three messages or only the messages it’s interested in. Most of the time, you can ignore win.WM_KEYDOWN and win.WM_KEYUP since win.TranslateMessage function call in the message loop translate win.WM_KEYDOWN and win.WM_KEYUP messages to win.WM_CHAR messages. We will focus on win.WM_CHAR in this tutorial.

þ

Analysis:

```
 kbdChar:char := ' ';þ//the character the program receives from keyboard
```

This is the variable that will store the character received from the keyboard. The initial value is $20 or the space since when our window refreshes its client area the first time, there is no character input. So we want to display space instead.

elseif(uMsg = win.WM_CHAR) then

```
    mov( (type byte wParam ), al );
    mov( al, kbdChar );
    win.InvalidateRect( hWnd, NULL, true );
```

This is added in the window procedure to handle the win.WM_CHAR message. It just puts the character into the variable named "char" and then calls win.InvalidateRect. win.InvalidateRect makes the specified rectangle in the client area invalid which forces Windows to send win.WM_PAINT message to the window procedure. Its syntax is as follows:

```
procedure InvalidateRect( hWnd:dword; lpRect:dword; bErase:dword );
```

lpRect is a pointer to the rectagle in the client area that we want to declare invalid. If this parameter is null, the entire client area will be marked as invalid.

bErase is a flag telling Windows if it needs to erase the background. If this flag is TRUE, then Windows will erase the backgroud of the invalid rectangle when win.BeginPaint is called.

So the strategy we used here is that: we store all necessary information relating to painting the client area and generate win.WM_PAINT message to paint the client area. Of course, the codes in win.WM_PAINT section must know beforehand what's expected of them. This seems a round-about way of doing things but it's the way of Windows.

Actually we can paint the client area during while processing the win.WM_CHAR message by calling win.GetDC and win.ReleaseDC pair. There is no problem there. But the fun begins when our window needs to repaint its client area. Since the codes that paint the character are in win.WM_CHAR section, the window procedure will not be able to repaint our character in the client area. So the bottom line is: put all necessary data and codes that do painting in win.WM_PAINT. You can send win.WM_PAINT message from anywhere in your code anytime you want to repaint the client area.

```
        win.TextOut
        (
            hdc,
            0,
            0,
            #{ lea( eax, kbdChar ); push( eax ); }#,
            1
        );
```

When win.InvalidateRect is called, it sends a win.WM_PAINT message back to the window procedure. So the codes in win.WM_PAINT section is called. It calls win.BeginPaint as usual to get the handle to device context and then calls win.TextOut which draws our character in the client area at x=0, y=0. When you run the program and press any key, you will see that character echo in the upper left corner of the client window. And when the window is minimized and maximized again, the character is still there since all the codes and data essential to repaint are all gathered in win.WM_PAINT section.