Tutorial 3: A Simple Window

In this tutorial, we will build a Windows program that displays a fully functional window on the desktop.

Source Code for This Tutorial:

```
// Iczelion's tutorial #3: A Simple Window

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;

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 );
        sub( eax, eax );

    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 );

    endif;

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;
    s:string;

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:

Windows programs rely heavily on API functions for their GUI. This approach benefits both users and programmers. For users, they don't have to learn how to navigate the GUI of each new programs, the GUI of Windows programs are alike. For programmers, the GUI codes are already there,tested, and ready for use. The downside for programmers is the increased complexity involved. In order to create or manipulate any GUI objects such as windows, menu or icons, programmers must follow a strict recipe. But that can be overcome by modular programming or OOP paradigm.

I'll outline the steps required to create a window on the desktop below:

- Get the instance handle of your program (required)
- Get the command line (not required unless your program wants to process a command line)
- Register window class (required ,unless you use predefined window types, eg. MessageBox or a dialog box)
- Create the window (required)
- Show the window on the desktop (required unless you don't want to show the window immediately)
- Refresh the client area of the window
- Enter an infinite loop, checking for messages from Windows
- If messages arrive, they are processed by a specialized function that is responsible for the window

- Quit program if the user closes the window

As you can see, the structure of a Windows program is rather complex compared to a DOS program. But the world of Windows is drastically different from the world of DOS. Windows programs must be able to coexist peacefully with each other. They must follow stricter rules. You, as a programmer, must also be more strict with your programming style and habit.

Content:

Above is the source code of our simple window program. Before jumping into the gory details of Win32 ASM programming, I'll point out some fine points which will ease your programming.

You should put all Windows constants, structures and function prototypes in an include file and include it at the beginning of your .hla file. It'll save you a lot of effort and typing. Currently, you can find many of the Windows definitions in the win32.hhf header file. You can also define your own constants & structure definitions but you should put them into a separate include file.

When declaring API function prototypes, structures, or constants in your include file, try to stick to the original names used in Windows include files, including case. This will save you a lot of headache when looking up some item in Win32 API reference. About the only exception should be the reasonable use of namespaces (e.g., the “win.” prefix in front of the names exported by the “win32.hhf” header file).

HLA automatically links in the Windows kernel32.lib and user32.lib libraries (at some point it will probably automatically link in gdi32.lib as well). If you start needing to link in additional Windows library files, you may want to start using makefiles to automate your compilation process. This will save you a lot of typing.

Analysis:

You may be taken aback that a simple Windows program requires so much coding. But most of those codes are just *template* codes that you can copy from one source code file to another. Or if you prefer, you could assemble some of these codes into a library to be used as prologue and epilogue codes. You can write only the codes in WinMain function. In fact, this is what C compilers do. They let you write WinMain codes without worrying about other housekeeping chores. The only catch is that you must have a function named WinMain else C compilers will not be able to combine your codes with the prologue and epilogue. You do not have such restriction with assembly language. You can use any function name instead of WinMain or no function at all.

Prepare yourself. This's going to be a long, long tutorial. Let's analyze this program to death!

```
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" )
```

We must include win32.hhf at the beginning of the source code. It contains important structures and constants that are used by our program. The include file , win32.hhf, is just a text file. You can open it with any text editor. Please note that win32.hhf does not contain all structures, and constants (yet). There are being added as needed. You can add in new items if they are not already in the file.

Our program calls HLA Standard Library routines in the string, memory management, command line argument, and conversions modules, so we must provide #include statements for each of these (alternatively, we could have simply included the “stdlib.hhf” header file which includes all HLA Standard Library header files). The win32 routines appear in the “kernel32.lib” and “use32.lib” library modules. As long as your programs only use routines from these modules and the HLA Standard Library modules you don’t have to do any special other than supply the appropriate #include statements. If you need to call win32 API routines above and beyond those in kerne32.lib and user32.lib, you will need to supply an appropriate header file and link in the appropriate LIB file. The next question : “how can I know which import library should be linked to my program?” The answer: You must know where the API functions called by your program reside. For example, if you call an API function in gdi32.dll, you must link with gdi32.lib.

Next are the "DATA" sections.

```
static
    hInstance:      dword;
    CommandLine:    string;

readonly

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



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

In READONLY, we declare two HLA strings. The win32 APIs require zero-terminated strings(ASCIIZ strings). Fortunately, HLA strings are zero terminated so we can use HLA strings for this purpose. The two strings are ClassName which is the name of our window class and AppName which is the name of our window. Note that the two variables are initialized (as is required in the READONLY section).

In the STATIC section, two variables are declared: hInstance (instance handle of our program) and CommandLine (command line of our program). Note that these variables in the STATIC section are not initialized, that is, they don't have to hold any specific value on startup, but we want to reserve the space for future use (actually, the win32 OS will initialize these variables with zero upon startup, but the code does not take advantage of this fact).

```
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;
```
The main program sits between the “begin aSimpileWindow;” and “end aSimpleWindow;” clauses.

Our first instruction is the call to GetModuleHandle to retrieve the instance handle of our program. Under Win32, instance handle and module handle are one and the same. You can think of instance handle as the ID of your program. It is used as parameter to several API functions our program must call, so it's generally a good idea to retrieve it at the beginning of our program.

Note: Actually under win32, instance handle is the linear address of your program in memory.

Upon returning from a Win32 function, the function's return value, if any, can be found in EAX. All other values are returned through variables passed in the function parameter list you defined for the call.

A Win32 function that you call will nearly always preserve the segment registers and the EBX, EDI, ESI and EBP registers. Conversely, ECX and EDX are considered scratch registers and are always undefined upon return from a Win32 function.

Note: Don't expect the values of EAX, ECX, EDX to be preserved across API function calls.

The bottom line is that: when calling an API function, expect return values in EAX. If any of your functions will be called by Windows, you must also play by the rule: preserve and restore the values of the segment registers, EBX, EDI, ESI and EBP upon function return else your program will crash very shortly, this includes your window procedure and windows callback functions.

The arg.CmdLn call is unnecessary if your program doesn't process a command line. In this example, I show you how to call it in case you need it in your program.

Next is the WinMain call. Here it receives four parameters: the instance handle of our program, the instance handle of the previous instance of our program, the command line and window state at first appearance. Under Win32, there's NO previous instance. Each program is alone in its address space, so the value of hPrevInst is always 0 (NULL). This is a leftover from the day of Win16 when all instances of a program run in the same address space and an instance wants to know if it's the first instance. Under win16, if hPrevInst is NULL, then this instance is the first one.

Note: You don't have to declare the function name as WinMain. In fact, you have complete freedom in this regard. You don't have to use any WinMain-equivalent function at all. You can paste the codes inside WinMain function after the arg.CmdLn routine and your program will still be able to function perfectly. WinMain is just a Windows tradition because this is the name of the C function that the OS calls when writing applications in C.

Upon returning from WinMain, EAX is filled with exit code. We pass that exit code as the parameter to ExitProcess which terminates our application.

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

The above line is the function declaration of WinMain. Note the parameter:type pairs that follow procedure line. They are parameters that WinMain receives from the caller. You can refer to these parameters by name instead of by stack manipulation. In addition, HLA will generate the prologue and epilogue codes for the function. So we don't have to concern ourselves with stack frame on function enter and exit. The NODISPLAY option tells HLA to skip the generation of a "display" for this procedure. Displays are only necessary if you have nested procedures and the nested procedures refer to VAR objects in the global (nesting) code. Since this is hardly ever necessary, it's rare that a procedure would actually need a display. Were HLA to generate a display, that would not affect the execution of program other than to add a few bytes of code and stack data.

```
var
    wc:     win.WNDCLASSEX;
    msg:    win.MSG;
    hwnd:   dword;
    s:string;

begin WinMain;
```

The VAR section is were the WinMain procedure allocates memory from the stack for local variables used in the procedure. The variable declarations must be immediately below the procedure directive and before the "begin WinMain;" clause. The variable declarations immediately follow the VAR directive. So wc:win.WNDCLASSEX tells HLA to allocate memory from the stack the size of the win.WNDCLASSEX structure for the variable named wc. We can refer to wc in our programs without any difficulty involved in stack manipulation. That's really a godsend, I think. The downside is that local variables cannot be used outside the function they're created and will be automatically destroyed when the function returns to the caller. Another drawback is that you cannot initialize local variables automatically because they're just stack memory allocated dynamically when the function is entered . You have to manually assign them with desired values after VAR directives (Note: if you want static objects in your HLA procedures, declare them in the STATIC, DATA, or READONLY sections; see the HLA documentation for more details).

```
    // 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 );
```

The inimidating lines above are really simple in concept. It just takes several lines of instruction to accomplish. The concept behind all these lines is the window class. A window class is nothing more than a blueprint or specification of a window. It defines several important characteristics of a window such as its icon, its cursor, the function responsible for it, its color etc. You create a window from a window class. This is some sort of object oriented concept. If you want to create more than one window with the same characteristics, it stands to reason to store all these characteristics in only one place and refer to them when needed. This scheme will save lots of memory by avoiding duplication of information. Remember, Windows is designed in the past when memory chips are prohibitive and most computers have 1 MB of memory. Windows must be very efficient in using the scarce memory resource. The point is: if you define your own window, you must fill the desired characteristics of your window in a win.WNDCLASS or win.WNDCLASSEX structure and call win.RegisterClass or win.RegisterClassEx before you're able to create your window. You only have to register the window class once for each window *type* you want to create a window from.

Windows has several predefined Window classes, such as button and edit box. For these windows (or controls), you don't have to register a window class, just call win.CreateWindowEx with the predefined class name.

The single most important member in the win.WNDCLASSEX is lpfnWndProc. lpfn stands for long pointer to function. Under Win32, there's no "near" or "far" pointer, just pointer because of the new FLAT memory model. But this is again a leftover from the days of Win16. Each window class must be associated with a function called window procedure. The window procedure is responsible for message handling of all windows created from the associated window class. Windows will send messages to the window procedure to notify it of important events concerning the windows it 's responsible for,such as user keyboard or mouse input. It's up to the window procedure to respond intelligently to each window message it receives. You will spend most of your time writing event handlers in window procedure.

I describe each member of win.WNDCLASSEX below:

```
WNDCLASSEX: record
  cbSize            :dword;
  style             :dword;
  lpfnWndProc       :dword;
  cbClsExtra        :dword;
  cbWndExtra        :dword;
  hInstance         :dword;
  hIcon             :dword;
  hCursor           :dword;
  hbrBackground     :dword;
  lpszMenuName      :dword;
  lpszClassName     :dword;
  hIconSm           :dword;
endrecord;
```

cbSize: The size of win.WNDCLASSEX structure in bytes. We can use the @size compile-time function to get the value for this field.

style: The style of windows created from this class. You can combine several styles together using compile time "|" operator. The win.hhf header file contains many of the Windows style constants; they typically begin with “win.WS_...”

lpfnWndProc: The address of the window procedure responsible for windows created from this class.

cbClsExtra: Specifies the number of extra bytes to allocate following the window-class structure. The operating system initializes the bytes to zero. You can store window class-specific data here.

cbWndExtra: Specifies the number of extra bytes to allocate following the window instance. The operating system initializes the bytes to zero. If an application uses the win.WNDCLASS structure to register a dialog box created by using the CLASS directive in the resource file, it must set this member to DLGWINDOWEXTRA.

hInstance: Instance handle of the module.

hIcon: Handle to the icon. Get it from LoadIcon call.

hCursor: Handle to the cursor. Get it from LoadCursor call.

hbrBackground: Background color of windows created from the class.

lpszMenuName: Default menu handle for windows created from the class.

lpszClassName: The name of this window class.

hIconSm: Handle to a small icon that is associated with the window class. If this member is NULL, the system searches the icon resource specified by the hIcon member for an icon of the appropriate size to use as the small icon.

```
    win.CreateWindowEx
    (
        NULL,
        ClassName,
        AppName,
        win.WS_OVERLAPPEDWINDOW,
        win.CW_USEDEFAULT,
        win.CW_USEDEFAULT,
```

```
        win.CW_USEDEFAULT,
        win.CW_USEDEFAULT,
        NULL,
        NULL,
        hInst,
        NULL
    );
```

After registering the window class, we can call CreateWindowEx to create our window based on the submitted window class. The following is a pseudo-prototype for this function (actually, iif you look in win.hhf, you'll find that "CreateWindowEx" is a macro that calls CreateWindowExA. We'll ignore this issue here). Notice that there are 12 parameters to this function.

```
procedure CreateWindowEx
(
   dwExStyle:dword;
   lpClassName:dword;
   lpWindowName:dword;
   dwStyle:dword;
   X:dword;
   Y:dword;
   nWidth:dword;
   nHeight:dword;
   hWndParent:dword;
   hMenu:dword;
   hInstance:dword;
   lpParam:dword
);
```

Let's see detailed description of each parameter:

dwExStyle: Extra window styles. This is the new parameter that is added to the old CreateWindow. You can put new window styles for Windows 95 & NT here.You can specify your ordinary window style in dwStyle but if you want some special styles such as topmost window, you must specify them here. You can use NULL if you don't want extra window styles.

lpClassName: (Required). Address of the HLA (ASCIIZ) string containing the name of window class you want to use as template for this window. The Class can be your own registered class or predefined window class. As stated above, every window you created must be based on a window class.

lpWindowName: Address of the ASCIIZ string containing the name of the window. It'll be shown on the title bar of the window. If this parameter is NULL, the title bar of the window will be blank.

dwStyle: Styles of the window. You can specify the appearance of the window here. Passing NULL is ok but the window will have no system menu box, no minimize-maximize buttons, and no close-window button. The window would not be of much use at all. You will need to press Alt+F4 to close it. The most common window style is win.WS_OVERLAPPEDWINDOW. A window style is only a bit flag. Thus you can combine several window styles by using the "|" operator to achieve the desired appearance of the window. win.WS_OVERLAPPEDWINDOW style is actually a combination of the most common window styles by this method.

X,Y: The coordinates of the upper left corner of the window. Normally this values should be win.CW_USEDEFAULT, that is, you want Windows to decide for you where to put the window on the desktop.

nWidth, nHeight: The width and height of the window in pixels. You can also use win.CW_USEDEFAULT to let Windows choose the appropriate width and height for you.

hWndParent: A handle to the window's parent window (if exists). This parameter tells Windows whether this window is a child (subordinate) of some other window and, if it is, which window is the parent. Note that this is not the parent-child relationship of multiple document interface (MDI). Child windows are not bound to the client area of the parent window. This relationship is specifically for Windows internal use. If the parent window is destroyed, all child windows will be destroyed automatically. It's really that simple. Since in our example, there's only one window, we specify this parameter as NULL.

hMenu: A handle to the window's menu. NULL if the class menu is to be used. Look back at the a member of win.WNDCLASSEX structure, lpszMenuName. lpszMenuName specifies *default* menu for the window class. Every window created from this window class will have the same menu by default. Unless you specify an *overriding* menu for a specific window via its hMenu parameter. hMenu is actually a dual-purpose parameter. In case the window you want to create is of a predefined window type (ie. control), such control cannot own a menu. hMenu is used as that control's ID instead. Windows can decide whether hMenu is really a menu handle or a control ID by looking at lpClassName parameter. If it's the name of a predefined window class, hMenu is a control ID. If it's not, then it's a handle to the window's menu.

hInstance: The instance handle for the program module creating the window.

lpParam: Optional pointer to a data structure passed to the window. This is used by MDI window to pass the CLIENTCREATESTRUCT data. Normally, this value is set to NULL, meaning that no data is passed via CreateWindow(). The window can retrieve the value of this parameter by the call to GetWindowLong function.

```
mov( eax, hwnd );
win.ShowWindow( hwnd, win.SW_SHOWNORMAL );
win.UpdateWindow( hwnd );
```

On successful return from win.CreateWindowEx, the window handle is returned in EAX. We must keep this value for future use. The window we just created is not automatically displayed. You must call win.ShowWindow with the window handle and the desired *display state* of the window to make it display on the screen. Next you can call win.UpdateWindow to order your window to repaint its client area. This function is useful when you want to update the content of the client area. You can omit this call though.

```
// 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;
```

At this time, our window is up on the screen. But it cannot receive input from the world. So we have to *inform* it of relevant events. We accomplish this with a message loop. There's only one message loop for each module. This message loop continually checks for messages from Windows with GetMessage call. GetMessage passes a pointer to a win.MSG structure to Windows. This win.MSG structure will be filled with information about the message that Windows want to send to a window in the module. win.GetMessage function will not return until there's a message for a window in the module. During that time, Windows can give control to other programs. This is what forms the cooperative multitasking scheme of Win16 platform .win.GetMessage returns FALSE if win.WM_QUIT message is received which, in the message loop, will terminate the loop and exit the program.

win.TranslateMessage is a utility function that takes raw keyboard input and generates a new message (win.WM_CHAR) that is placed on the message queue. The message with win.WM_CHAR contains the ASCII value for the key pressed, which is easier to deal with than the raw keyboard scan codes. You can omit this call if your program doesn't process keystrokes.

win.DispatchMessage sends the message data to the window procedure responsible for the specific window the message is for.

```
    mov( msg.wParam, eax );

end WinMain;
```

If the message loop terminates, the exit code is stored in wParam member of the win.MSG structure. You can store this exit code into EAX to return it to Windows. At the present time, Windows does not make use of the return value, but it's better to be on the safe side and play by the rules.

```
// 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;

begin WndProc;
```

This is our window procedure. You don't have to name it WndProc. The last parameter, hWnd, is the window handle of the window that the message is destined for. **Note**: when using the STDCALL calling convention, hWnd is actually the first parameter, but with the Pascal calling convention it is the last parameter.. Since your code doesn't usually call WndProc, there is little need to create a macro that will reverse the parameters for us. uMsg is the message value. Note that uMsg is not a MSG structure. It's just a number, really. Windows defines hundreds of messages, most of which your programs will not be interested in. Windows will send an appropriate message to a window in case something relevant to that window happens. The window procedure receives the message and reacts to it intelligently. wParam and lParam are just extra parameters for use by some messages. Some messages do send accompanying data in addition to the message itself. Those data are passed to the window procedure by means of lParam and wParam.

```
    // 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 );
        sub( eax, eax );

    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 );

    endif;

end WndProc;
```

Here comes the crucial part. This is where most of your program's intelligence resides. The codes that respond to each Windows message are in the window procedure. Your code must check the Windows message to see if it's a message it's interested in. If it is, do anything you want to do in response to that message and then return with zero in eax. If it's not, you MUST call DefWindowProc, passing all parameters you received to it for default processing.. This DefWindowProc is an API function that processes the messages your program are not interested in.

The only message that you MUST respond to is WM_DESTROY. This message is sent to your window procedure whenever your window is closed. By the time your window procedure receives this message, your window is already removed from the screen. This is just a notification that your window was destroyed, you should prepare yourself to return to Windows. In response to this, you can perform housekeeping prior to returning to Windows. You have no choice but to quit when it comes to this state. If you want to have a chance to stop the user from closing your window, you

should process win.WM_CLOSE message. Now back to win.WM_DESTROY, after performing housekeeping chores, you must call win.PostQuitMessage which will post win.WM_QUIT back to your module. win.WM_QUIT will make win.GetMessage return with zero value in EAX, which in turn, terminates the message loop and quits to Windows. You can send win.WM_DESTROY message to your own window procedure by calling win.DestroyWindow function.