Tutorial 8: Menus

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 8: Menus

In this tutorial, we will learn how to incorporate a menu into our window.

Source Code: Resource File (tut8.rc)

```
#define IDM_TEST 1
#define IDM_HELLO 2
#define IDM_GOODBYE 3
#define IDM_EXIT 4

FirstMenu MENU
{
 POPUP "&PopUp"
        {
         MENUITEM "&Say Hello",IDM_HELLO
         MENUITEM "Say &GoodBye", IDM_GOODBYE
```

```
         MENUITEM SEPARATOR
         MENUITEM "E&xit",IDM_EXIT
        }
 MENUITEM "&Test", IDM_TEST
}
```

Source Code: Tut8.hla (part 1)

```
// Iczelion's tutorial #8a: Menus, Part I

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
    IDM_TEST    := 1;
    IDM_HELLO   := 2;
    IDM_GOODBYE := 3;
    IDM_EXIT    := 4;

    AppNameStr  := "Our First Window";
    MenuNameStr := "FirstMenu";

readonly

    ClassName:  string := "SimpleWinClass";
    AppName:    string := AppNameStr;
    MenuName:   string := MenuNameStr;
    TestStr:    string := "You selected Test menu item";
    Hello_str:  string := "Hello, my friend";
    Goodbye_str:string := "See you again, bye";



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 );


    elseif( uMsg = win.WM_COMMAND ) then

        mov( wParam, eax );
        if( ax = IDM_TEST ) then

            win.MessageBox( NULL, TestStr, AppName, win.MB_OK );

        elseif( ax = IDM_HELLO ) then

            win.MessageBox( NULL, Hello_str, AppName, win.MB_OK );

        elseif( ax = IDM_GOODBYE ) then

            win.MessageBox( NULL, Goodbye_str, AppName, win.MB_OK );

        else

            win.DestroyWindow( hWnd );

        endif;

    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( MenuName, 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;
```

Source Code: Tut8b.hla (Part 2)

```
// Iczelion's tutorial #8b: Menus, Part II

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
    hMenu:          dword;
    hInstance:      dword;
    CommandLine:    string;

const
    IDM_TEST    := 1;
    IDM_HELLO   := 2;
    IDM_GOODBYE := 3;
    IDM_EXIT    := 4;

    AppNameStr  := "Our First Window";
    MenuNameStr := "FirstMenu";

readonly

    ClassName:  string := "SimpleWinClass";
    AppName:    string := AppNameStr;
    MenuName:   string := MenuNameStr;
    TestStr:    string := "You selected Test menu item";
    Hello_str:  string := "Hello, my friend";
    Goodbye_str:string := "See you again, bye";
```

```
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 );


    elseif( uMsg = win.WM_COMMAND ) then

        mov( wParam, eax );
        if( ax = IDM_TEST ) then

            win.MessageBox( NULL, TestStr, AppName, win.MB_OK );

        elseif( ax = IDM_HELLO ) then

            win.MessageBox( NULL, Hello_str, AppName, win.MB_OK );

        elseif( ax = IDM_GOODBYE ) then

            win.MessageBox( NULL, Goodbye_str, AppName, win.MB_OK );

        else

            win.DestroyWindow( hWnd );

        endif;

    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 );

// Attach the menu under program control

win.LoadMenu( hInst, MenuName );
mov( eax, hMenu );


win.CreateWindowEx
(
    NULL,
    ClassName,
    AppName,
    win.WS_OVERLAPPEDWINDOW,
    win.CW_USEDEFAULT,
    win.CW_USEDEFAULT,
    win.CW_USEDEFAULT,
    win.CW_USEDEFAULT,
    NULL,
    hMenu,
    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:

Menus are one of the most important components in your window. Menus present a list of services a program offers to the user. The user doesn't have to read the manual included with the program to be able to use it, he can peruse the menus to get an overview of the capability of a particular program and start playing with it immediately. Since a menu is a tool to get the user up and running quickly, you should follow the standard. Succintly put, the first two menu items should be File and Edit and the last should be Help. You can insert your own menu items between Edit and Help. If a menu item invokes a dialog box, you should append an ellipsis (...) to the menu string.

Menus are a kind of resource. There are several kinds of resources such as dialog box, string table, icon, bitmap, menu etc. Resources are described in a separated file called a resource file which normally has .rc extension. You then combine the resources with the source code during the link stage. The final product is an executable file which contains both instructions and resources.

You can write resource scripts using any text editor. They're composed of phrases which describe the appearances and other attributes of the resources used in a particular program Although you can write resource scripts with a text editor, it's rather cumbersome. A better alter-

native is to use a resource editor which lets you visually design resources with ease. Resource editors are usually included in compiler packages such as Visual C++, Borland C++, etc.

You describe a menu resource like this:

```
MyMenu  MENU
{
  [menu list here]
}
```

C programmers may recognize that it is similar to declaring a structure. MyMenu being a menu name followed by MENU keyword and menu list within curly brackets. Alternatively, you can use BEGIN and END instead of the curly brackets if you wish. This syntax is more palatable to Pascal programmers.

Menu list can be either MENUITEM or POPUP statement.

MENUITEM statement defines a menu bar which doesn't invoke a popup menu when selected.The syntax is as follows:

```
MENUITEM "&text", ID [,options]
```

It begins by MENUITEM keyword followed by the text you want to use as menu bar string. Note the ampersand. It causes the character that follows it to be underlined. Following the text string is the ID of the menu item. The ID is a number that will be used to identify the menu item in the message sent to the window procedure when the menu item is selected. As such, each menu ID must be unique among themselves.

Options are optional. Available options are as follows:

*GRAYED The menu item is inactive, and it does not generate a WM_COMMAND message. The text is grayed.

*INACTIVE The menu item is inactive, and it does not generate a WM_COMMAND message. The text is displayed normally.

*MENUBREAK This item and the following items appear on a new line of the menu.

*HELP This item and the following items are right-justified.

You can use one of the above option or combine them with "or" operator. Beware that INACTIVE and GRAYED cannot be combined together.

POPUP statement has the following syntax:

```
POPUP "&text" [,options]
{
  [menu list]
}
```

POPUP statement defines a menu bar that, when selected, drops down a list of menu items in a small popup window. The menu list can be a MENUTIEM or POPUP statement. There's a special kind of MENUITEM statement, MENUITEM SEPARATOR, which will draw a horizontal line in the popup window.

The next step after you are finished with the menu resource script is to reference it in your program.

You can do this in two different places in your program.

*In lpszMenuName member of WNDCLASSEX structure. Say, if you have a menu named "FirstMenu", you can assigned the menu to your window like this:

```
const
    MenuNameStr := "FirstMenu";

readonly
    MenuName:   string := MenuNameStr;
       .
       .
       .

    mov( MenuName, wc.lpszMenuName );
```

*In menu handle parameter of CreateWindowEx like this:

```
const
    MenuNameStr := "FirstMenu";

static
    hMenu:          dword;
      .
      .
      .
readonly
    MenuName:   string := MenuNameStr;
      .
      .
      .
    win.LoadMenu( hInst, MenuName );
    mov( eax, hMenu );
    win.CreateWindowEx
    (
        NULL,
        ClassName,
        AppName,
        win.WS_OVERLAPPEDWINDOW,
        win.CW_USEDEFAULT,
        win.CW_USEDEFAULT,
        win.CW_USEDEFAULT,
        win.CW_USEDEFAULT,
```

```
        NULL,
        hMenu,
        hInst,
        NULL
    );
    mov( eax, hwnd );
```

So you may ask, what's the difference between these two methods?

When you reference the menu in the win.WNDCLASSEX structure, the menu becomes the "default" menu for the window class. Every window of that class will have the same menu.

If you want each window created from the same class to have different menus, you must choose the second form. In this case, any window that is passed a menu handle in its win.CreateWindowEx function will have a menu that "overrides" the default menu defined in the win.WNDCLASSEX structure.

Next we will examine how a menu notifies the window procedure when the user selects a menu item.

When the user selects a menu item, the window procedure will receive a win.WM_COMMAND message. The low word of wParam contains the menu ID of the selected menu item.

Now we have sufficient information to create and use a menu. Let's do it. (See the first source listing [part I]).

Analysis:

Let's analyze the resource file first.

```
#define IDM_TEST 1                /* equal to “const IDM_TEST := 1;” */
#define IDM_HELLO 2
#define IDM_GOODBYE 3
#define IDM_EXIT 4
```

The above lines define the menu IDs used by the menu script. You can assign any value to the ID as long as the value is unique in the menu.

```
FirstMenu MENU
```

Declare your menu with MENU keyword.

```
POPUP "&PopUp"
        {
         MENUITEM "&Say Hello",IDM_HELLO
         MENUITEM "Say &GoodBye", IDM_GOODBYE
```

```
 MENUITEM SEPARATOR
 MENUITEM "E&xit",IDM_EXIT
}
```

Define a popup menu with four menu items, the third one is a menu separator.

```
MENUITEM "&Test", IDM_TEST
```

Define a menu bar in the main menu.

Next we will examine the source code.

```
readonly
      .
      .
      .
    MenuName:   string := MenuNameStr;
    TestStr:    string := "You selected Test menu item";
    Hello_str:  string := "Hello, my friend";
    Goodbye_str:string := "See you again, bye";
```

MenuName is the name of the menu in the resource file. Note that you can define more than one menu in the resource file so you must specify which menu you want to use. The remaining three lines define the text strings to be displayed in message boxes that are invoked when the appropriate menu item is selected by the user.

```
const
    IDM_TEST    := 1;
    IDM_HELLO   := 2;
    IDM_GOODBYE := 3;
    IDM_EXIT    := 4;
```

Define menu IDs for use in the window procedure. These values MUST be identical to those defined in the resource file.

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

        mov( wParam, eax );
        if( ax = IDM_TEST ) then

            win.MessageBox( NULL, TestStr, AppName, win.MB_OK );

        elseif( ax = IDM_HELLO ) then
```

```
            win.MessageBox( NULL, Hello_str, AppName, win.MB_OK );

        elseif( ax = IDM_GOODBYE ) then

            win.MessageBox( NULL, Goodbye_str, AppName, win.MB_OK );

        else

            win.DestroyWindow( hWnd );

        endif;
```

In the window procedure, we process win.WM_COMMAND messages. When the user selects a menu item, the menu ID of that menu item is sended to the window procedure in the low word of wParam along with the win.WM_COMMAND message. So when we store the value of wParam in eax, we compare the value in ax to the menu IDs we defined previously and act accordingly. In the first three cases, when the user selects Test, Say Hello, and Say GoodBye menu items, we just display a text string in a message box.

If the user selects Exit menu item, we call win.DestroyWindow with the handle of our window as its parameter which will close our window.

As you can see, specifying menu name in a window class is quite easy and straightforward. However you can also use an alternate method to load a menu in your window. I won't show the entire source code here. The resource file is the same in both methods. There are some minor changes in the source file which I 'll show below.

```
static
    hMenu:          dword;    // handle of our menu
```

Define a variable of type dword to store our menu handle.

```
    win.LoadMenu( hInst, MenuName );
    mov( eax, hMenu );


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

```
);
mov( eax, hwnd );
```

Before calling win.CreateWindowEx, we call win.LoadMenu with the instance handle and a pointer to the name of our menu. win.LoadMenu returns the handle of our menu in the resource file which we pass to win.CreateWindowEx.