Tutorial 9:Child Window Controls

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 9: Child Window Controls

In this tutorial, we will explore child window controls which are very important input and output devices of our programs.

Source Code

```
// Iczelion's tutorial #9: Controls
//
//  To compile this program use the command line:
//
//  hla -w tut9.hla tut9.rc


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;
    hwndButton:     dword;
    hwndEdit:       dword;
    buffer:         char[ 512 ];

const
    ButtonID    := 1;
    EditID      := 2;

    IDM_HELLO   := 1;
    IDM_CLEAR   := 2;
    IDM_GETTEXT := 3;
    IDM_EXIT    := 4;


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

readonly

    ClassName:  string := "SimpleWinClass";
    AppName:    string := AppNameStr;
    MenuName:   string := MenuNameStr;
    TestStr:    string := "Wow! I'm in an edit box now";



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_CREATE ) then

    win.CreateWindowEx
    (
        win.WS_EX_CLIENTEDGE,
        "edit",
        NULL,
        win.WS_CHILD | win.WS_VISIBLE |
            win.WS_BORDER | win.ES_LEFT | win.ES_AUTOHSCROLL,
        50,
        35,
        200,
        25,
        hWnd,
        EditID,
        hInstance,
        NULL
    );
    mov( eax, hwndEdit );
    win.SetFocus( hwndEdit );

    win.CreateWindowEx
    (
        NULL,
        "button",
        "My First Button",
        win.WS_CHILD | win.WS_VISIBLE | win.BS_DEFPUSHBUTTON,
        75,
        70,
        140,
        25,
        hWnd,
        ButtonID,
        hInstance,
        NULL
    );
    mov( eax, hwndButton );


elseif( uMsg = win.WM_COMMAND ) then

    mov( wParam, eax );
    if( lParam = 0 ) then
```

```
        if( ax = IDM_HELLO ) then

            win.SetWindowText( hwndEdit, TestStr );
            win.SendMessage( hwndEdit, win.WM_KEYDOWN, win.VK_END, NULL );

        elseif( ax = IDM_CLEAR ) then

            win.SetWindowText( hwndEdit, NULL );

        elseif( ax = IDM_GETTEXT ) then

            win.GetWindowText( hwndEdit, buffer, 512 );
            win.MessageBox
            (
                NULL,
                #{ pushd( &buffer ); }#,
                AppName,
                win.MB_OK
            );

        else

            win.DestroyWindow( hWnd );

        endif;

    else

        if( ax = ButtonID ) then

            shr( 16, eax );
            if( ax = win.BN_CLICKED ) then

                win.SendMessage
                (
                    hWnd,
                    win.WM_COMMAND,
                    IDM_GETTEXT,
                    0
                );

            endif;

        endif;

    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_BTNFACE+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
    (
        win.WS_EX_CLIENTEDGE,
        ClassName,
        AppName,
        win.WS_OVERLAPPEDWINDOW,
        win.CW_USEDEFAULT,
        win.CW_USEDEFAULT,
        300,
        200,
        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 provides several predefined window classes which we can readily use in our own programs. Most of the time we use them as components of a dialog box so they're usually called child window controls. The child window controls process their own mouse and keyboard messages and notify the parent window when their states have changed. They relieve the burden from programmers enormously so you should use them as much as possible. In this tutorial, I put them on a normal window just to demonstrate how you can create and use them but in reality you should put them in a dialog box.

Examples of predefined window classes are buttons, listbox, checkboxes, radio buttons, edit controls, etc.

In order to use a child window control, you must create it with win.CreateWindow or win.CreateWindowEx. Note that you don't have to register the window class since it's registered for you by Windows. The class name parameter MUST be the predefined class name. Say, if you want to create a button, you must specify "button" as the class name in win.CreateWindowEx. The other parameters you must fill in are the parent window handle and the control ID. The control ID must be unique among the controls. The control ID is the ID of that control. You use it to differentiate between the controls.

After the control was created, it will send messages notifying the parent window when its state has changed. Normally, you create the child windows during win.WM_CREATE message of the parent window. The child window sends win.WM_COMMAND messages to the parent window with its control ID in the low word of wParam, the notification code in the high word of wParam, and its window handle in lParam. Each child window control has different notification codes, refer to your Win32 API reference for more information.

The parent window can send commands to the child windows too, by calling win.SendMessage function. win.SendMessage function sends the specified message with accompanying values

in wParam and lParam to the window specified by the window handle. It's an extremely useful function since it can send messages to any window provided you know its window handle.

So, after creating the child windows, the parent window must process win.WM_COMMAND messages to be able to receive notification codes from the child windows.

Example:

We will create a window which contains an edit control and a pushbutton. When you click the button, a message box will appear showing the text you typed in the edit box. There is also a menu with 4 menu items:

1.Say Hello -- Put a text string into the edit box

2.Clear Edit Box -- Clear the content of the edit box

3.Get Text -- Display a message box with the text in the edit box

4.Exit -- Close the program.

See the source code for more details.

Analysis:

Let's analyze the program.

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

     win.CreateWindowEx
     (
         win.WS_EX_CLIENTEDGE,
         "edit",
         NULL,
         win.WS_CHILD | win.WS_VISIBLE |
             win.WS_BORDER | win.ES_LEFT | win.ES_AUTOHSCROLL,
         50,
         35,
         200,
         25,
         hWnd,
         EditID,
         hInstance,
         NULL
     );
     mov( eax, hwndEdit );
     win.SetFocus( hwndEdit );
```

```
win.CreateWindowEx
(
    NULL,
    "button",
    "My First Button",
    win.WS_CHILD | win.WS_VISIBLE | win.BS_DEFPUSHBUTTON,
    75,
    70,
    140,
    25,
    hWnd,
    ButtonID,
    hInstance,
    NULL
);
mov( eax, hwndButton );
```

We create the controls during processing of win.WM_CREATE message. We call win.CreateWindowEx with an extra window style, win.WS_EX_CLIENTEDGE, which makes the client area look sunken. The name of each control is a predefined one, "edit" for edit control, "button" for button control. Next we specify the child window's styles. Each control has extra styles in addition to the normal window styles. For example, the button styles are prefixed with "BS_" for "button style", edit styles are prefixed with "ES_" for "edit style". You have to look these styles up in a Win32 API reference. Note that you put a control ID in place of the menu handle. This doesn't cause any harm since a child window control cannot have a menu.

After creating each control, we keep its handle in a variable for future use.

win.SetFocus is called to give input focus to the edit box so the user can type the text into it immediately.

Now comes the really exciting part. Every child window control sends notification to its parent window with win.WM_COMMAND.

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

    mov( wParam, eax );
    if( lParam = 0 ) then
```

Recall that a menu also sends win.WM_COMMAND messages to notify the window about its state too. How can you differentiate between win.WM_COMMAND messages originated from a menu or a control? Below is the answer

Table 1: WM_COMMAND Differentiation

	Low word of wParam	High word of wParam	lParam
Menu	Menu ID	0	0
Control	Control ID	Notification code	Child Window Handle

You can see that you should check lParam. If it's zero, the current win.WM_COMMAND message is from a menu. You cannot use wParam to differentiate between a menu and a control since the menu ID and control ID may be identical and the notification code may be zero.

```
if( ax = IDM_HELLO ) then

    win.SetWindowText( hwndEdit, TestStr );
    win.SendMessage( hwndEdit, win.WM_KEYDOWN, win.VK_END, NULL );

elseif( ax = IDM_CLEAR ) then

    win.SetWindowText( hwndEdit, NULL );

elseif( ax = IDM_GETTEXT ) then

    win.GetWindowText( hwndEdit, buffer, 512 );
    win.MessageBox
    (
        NULL,
        #{ pushd( &buffer ); }#,
        AppName,
        win.MB_OK
    );
```

You can put a text string into an edit box by calling win.SetWindowText. You clear the content of an edit box by calling win.SetWindowText with NULL. win.SetWindowText is a general purpose API function. You can use win.SetWindowText to change the caption of a window or the text on a button.

To get the text in an edit box, you use win.GetWindowText.

```
if( ax = ButtonID ) then

    shr( 16, eax );
    if( ax = win.BN_CLICKED ) then

        win.SendMessage
        (
            hWnd,
            win.WM_COMMAND,
            IDM_GETTEXT,
            0
        );

    endif;

endif;
```

The above code snippet deals with the condition when the user presses the button. First, it checks the low word of wParam to see if the control ID matches that of the button. If it is, it

checks the high word of wParam to see if it is the notification code win.BN_CLICKED which is sent when the button is clicked.

The interesting part is after it's certain that the notification code is win.BN_CLICKED. We want to get the text from the edit box and display it in a message box. We can duplicate the code in the IDM_GETTEXT section above but it doesn't make sense. If we can somehow send a win.WM_COMMAND message with the low word of wParam containing the value IDM_GETTEXT to our own window procedure, we can avoid code duplication and simplify our program. SendMessage function is the answer. This function sends any message to any window with any wParam and lParam we want. So instead of duplicating the code, we call win.SendMessage with the parent window handle, win.WM_COMMAND, IDM_GETTEXT, and 0. This has identical effect to selecting "Get Text" menu item from the menu. The window procedure doesn't perceive any difference between the two.

You should use this technique as much as possible to make your code more organized.

Last but not least, do not forget the win.TranslateMessage function in the message loop. Since you must type in some text into the edit box, your program must translate raw keyboard input into readable text. If you omit this function, you will not be able to type anything into your edit box.