Tutorial 5: More About Text

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 5: More about Text

We will experiment more with text attributes, ie. font and color.

Source Code for this Tutorial

```
// Iczelion's tutorial #5: More About Text

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" );


macro RGB( red, green, blue );

    xor( eax, eax );
    mov( blue, ah );
    shl( 8, eax );
    mov( green, ah );
    mov( red, al );

endmacro;


// 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;
    hfont:  dword;

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_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.CreateFont
    (
        24,
        16,
        0,
        0,
        400,
        0,
        0,
        0,
        win.OEM_CHARSET,
        win.OUT_DEFAULT_PRECIS,
        win.CLIP_DEFAULT_PRECIS,
        win.DEFAULT_QUALITY,
        win.DEFAULT_PITCH | win.FF_SCRIPT,
        "script"
    );
    win.SelectObject( hdc, eax );
    mov( eax, hfont );

    RGB( 200,200,50 );
    win.SetTextColor( hdc, eax );

    RGB( 0,0,255 );
    win.SetBkColor( hdc, eax );

    win.TextOut
    (
        hdc,
        0,
        0,
        TestString,
        @length( TestString )
    );
    win.SelectObject( hdc, hfont );
    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:

Windows color system is based on RGB values, R=red, G=Green, B=Blue. If you want to specify a color in Windows, you must state your desired color in terms of these three major colors. Each color value has a range from 0 to 255 (a byte value). For example, if you want pure red color, you should use 255,0,0. Or if you want pure white color, you must use 255,255,255. You can see from the examples that getting the color you need is very difficult with this system since you have to have a good grasp of how to mix and match colors.

For text color and background, you use win.SetTextColor and win.SetBkColor, both of them require a handle to device context and a 32-bit RGB value. The 32-bit RGB value's structure is defined as:

```
RGB_value:record
    unused:byte;
    blue  :byte;
    green :byte;
    red   :byte;
endrecord;
```

Note that the first byte is not used and should be zero. The order of the remaining three bytes is reversed,ie. blue, green, red. However, we will not use this structure since it's cumbersome to initialize and use. We will create a macro instead. The macro will receive three parameters: red, green and blue values. It'll produce the desired 32-bit RGB value and store it in eax. The macro is as follows:

```
macro RGB( red, green, blue );
    xor( eax, eax );
    mov( blue, ah );
    shl( 8, eax);
    mov( green, ah );
    mov( red, al );
endmacro;
```

You can put this macro in the include file for future use.

You can "create" a font by calling win.CreateFont or win.CreateFontIndirect. The difference between the two functions is that win.CreateFontIndirect receives only one parameter: a pointer to a logical font structure, win.LOGFONT. win.CreateFontIndirect is the more flexible of the two especially if your programs need to change fonts frequently. However, in our example, we will "create" only one font for demonstration, we can get away with win.CreateFont. After the call to win.CreateFont, it will return a handle to a font which you must select into the device context. After that, every text API function will use the font we have selected into the device context.

Analysis:

```
win.CreateFont
(
    24,
    16,
    0,
    0,
    400,
    0,
    0,
    0,
    win.OEM_CHARSET,
    win.OUT_DEFAULT_PRECIS,
    win.CLIP_DEFAULT_PRECIS,
    win.DEFAULT_QUALITY,
    win.DEFAULT_PITCH | win.FF_SCRIPT,
    "script"
);
```

win.CreateFont creates a logical font that is the closest match to the given parameters and the font data available. This function has more parameters than any other function in Windows. It returns a handle to logical font to be used by win.SelectObject function. We will examine its parameters in detail.

```
nHeight:dword
nWidth:dword
nEscapement:dword
nOrientation:dword
nWeight:dword
cItalic:dword
cUnderline:dword
cStrikeOut:dword
cCharSet:dword
cOutputPrecision:dword
```

```
                cClipPrecision:dword
                cQuality:dword
                cPitchAndFamily:dword
                lpFacename:dword
```

nHeight The desired height of the characters . 0 means use default size.

nWidth The desired width of the characters. Normally this value should be 0 which allows Windows to match the width to the height. However, in our example, the default width makes the characters hard to read, so I use the width of 16 instead.

nEscapement Specifies the orientation of the next character output relative to the previous one in tenths of a degree. Normally, set to 0. Set to 900 to have all the characters go upward from the first character, 1800 to write backwards, or 2700 to write each character from the top down.

nOrientation Specifies how much the character should be rotated when output in tenths of a degree. Set to 900 to have all the characters lying on their backs, 1800 for upside-down writing, etc.

nWeight Sets the line thickness of each character. Windows defines the following sizes:

FW_DONTCARE	0
FW_THIN	$100
FW_EXTRALIGHT	$200
FW_ULTRALIGHT	$200
FW_LIGHT	$300
FW_NORMAL	$400
FW_REGULAR	$400
FW_MEDIUM	$500
FW_SEMIBOLD	$600
FW_DEMIBOLD	$600
FW_BOLD	$700
FW_EXTRABOLD	$800
FW_ULTRABOLD	$800
FW_HEAVY	$900
FW_BLACK	$900

cItalic 0 for normal, any other value for italic characters.

cUnderline 0 for normal, any other value for underlined characters.

cStrikeOut 0 for normal, any other value for characters with a line through the center.

cCharSet The character set of the font. Normally should be win.OEM_CHARSET which allows Windows to select font which is operating system-dependent.

cOutputPrecision Specifies how much the selected font must be closely matched to the characteristics we want. Normally should be win.OUT_DEFAULT_PRECIS which defines default font mapping behavior.

cClipPrecision Specifies the clipping precision. The clipping precision defines how to clip characters that are partially outside the clipping region. You should be able to get by with win.CLIP_DEFAULT_PRECIS which defines the default clipping behavior.

cQuality Specifies the output quality. The output quality defines how carefully GDI must attempt to match the logical-font attributes to those of an actual physical font. There are three choices: win.DEFAULT_QUALITY, win.PROOF_QUALITY and win.DRAFT_QUALITY.

cPitchAndFamily Specifies pitch and family of the font. You must combine the pitch value and the family value with the "|" operator.

lpFacename A pointer to a null-terminated string that specifies the typeface of the font.

The description above is by no means comprehensive. You should refer to your Win32 API reference for more details.

```
        win.SelectObject( hdc, eax );
        mov( eax, hfont );
```

After we get the handle to the logical font, we must use it to select the font into the device context by calling win.SelectObject. win.SelectObject puts the new GDI objects such as pens, brushs, and fonts into the device context to be used by GDI functions. It returns the handle to the replaced object which we should save for future win.SelectObject calls. After win.SelectObject call, any text output function will use the font we just selected into the device context.

```
        RGB( 200,200,50 );
        win.SetTextColor( hdc, eax );

        RGB( 0,0,255 );
        win.SetBkColor( hdc, eax );
```

```
Use RGB macro to create a 32-bit RGB value to be used by SetColorText and Set-
BkColor.
```

```
        win.TextOut
        (
            hdc,
            0,
            0,
            TestString,
            @length( TestString )
        );
```

Call win.TextOut function to draw the text on the client area. The text will be in the font and color we specified previously.

```
win.SelectObject( hdc, hfont );
```

When we are through with the font, we should restore the old font back into the device context. You should always restore the object that you replaced in the device context.