Tutorial 4: Painting with 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:

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.

Painting With Text

In this tutorial, we will learn how to "paint" text in the client area of a window. We'll also learn about device context.

Source Code for this Tutorial:

```
// Iczelion's tutorial #4: Painting With 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" );


// 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;

var
    hdc:    dword;
    ps:     win.PAINTSTRUCT;
    rect:   win.RECT;

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 );

    /* New Code Added for Tutorial 4 */

    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.GetClientRect( hWnd, rect );
        win.DrawText
        (
            hdc,
            "Win32 assembly is great and easy!",
            -1,
            rect,
            win.DT_SINGLELINE | win.DT_CENTER | win.DT_VCENTER
        );
        win.EndPaint( hWnd, ps );



    /* End of new code */

    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:

Text in Windows is a type of GUI object. Each character is composed of numerous pixels (dots) that are lumped together into a distinct pattern. That's why it's called "painting" instead of "writing". Normally, you paint text in your own client area (actually, you can paint outside client area but that's another story). Putting text on screen in Windows is drastically different from DOS. In DOS, you can think of the screen in 80x25 dimension. But in Windows, the screen are shared by several programs. Some rules must be enforced to avoid programs writing over each other's screen. Windows ensures this by limiting painting area of each window to its own client area only. The size of client area of a window is also not constant. The user can change the size anytime. So you must determine the dimensions of your own client area dynamically.

Before you can paint something on the client area, you must ask for permission from Windows. That's right, you don't have absolute control of the screen as you aren't in DOS anymore. You must ask Windows for permission to paint your own client area. Windows will determine the size of your client area, font, colors and other GDI attributes and sends a handle to device context back to your program. You can then use the device context as a passport to painting on your client area.

What is a device context? It's just a data structure maintained internally by Windows. A device context is associated with a particular device, such as a printer or video display. For a video display, a device context is usually associated with a particular window on the display.

Some of the values in the device context are graphic attributes such as colors, font etc. These are default values which you can change at will. They exist to help reduce the load from having to specify these attributes in every GDI function calls.

You can think of a device context as a default environment prepared for you by Windows. You can override some default settings later if you so wish.

When a program needs to paint, it must obtain a handle to a device context. Normally, there are several ways to accomplish this.

- Call win.BeginPaint in response to WM_PAINT message.
- Call win.GetDC in response to other messages.
- Call win.CreateDC to create your own device context

One thing you must remember, after you're through with the device context handle, you must release it during the processing of a single message. Don't obtain the handle in response to one message and release it in response to another.

Windows posts win.WM_PAINT messages to a window to notify that it's now time to repaint its client area. Windows does not save the content of client area of a window. Instead, when a situation occurs that warrants a repaint of client area (such as when a window was covered by another and is just uncovered), Windows puts win.WM_PAINT message in that window's message queue. It's the responsibility of that window to repaint its own client area. You must gather all information about how to repaint your client area in the win.WM_PAINT section of your window procedure, so the window procudure can repaint the client area when win.WM_PAINT message arrives.

Another concept you must come to terms with is the invalid rectangle. Windows defines an invalid rectangle as the smallest rectangular area in the client area that needs to be repainted. When Windows detects an invalid rectangle in the client area of a window , it posts win.WM_PAINT message to that window. In response to win.WM_PAINT message, the window can obtain a paintstruct structure which contains, among others, the coordinate of the invalid rectangle. You call BeginPaint in response to win.WM_PAINT message to validate the invalid rectangle. If you don't process win.WM_PAINT message, at the very least you must call win.DefWindowProc or win.ValidateRect to validate the invalid rectangle else Windows will repeatedly send you win.WM_PAINT message.

Below are the steps you should perform in response to a win.WM_PAINT message:

- Get a handle to device context with win.BeginPaint.

- Paint the client area.
- Release the handle to device context with win.EndPaint

Note that you don't have to explicitly validate the invalid rectangle. It's automatically done by the win.BeginPaint call. Between win.BeginPaint-win.Endpaint pair, you can call any GDI functions to paint your client area. Nearly all of them require the handle to device context as a parameter.

Content:

We will write a program that displays a text string "Win32 assembly is great and easy!" in the center of the client area. (See the previous source code.)

Analysis:

The majority of the code is the same as the example in tutorial 3. I'll explain only the important changes.

```
procedure WndProc( lParam:dword; wParam:dword; uMsg:uns32; hWnd:dword );
    nodisplay;
var
    hdc:    dword;
    ps:     win.PAINTSTRUCT;
    rect:   win.RECT;
```

These are local variables that are used by GDI functions in our win.WM_PAINT section. hdc is used to store the handle to device context returned from win.BeginPaint call. ps is a win.PAINTSTRUCT structure. Normally you don't use the values in ps. It's passed to win.BeginPaint function and Windows fills it with appropriate values. You then pass ps to win.EndPaint function when you finish painting the client area. rect is a win.RECT structure defined as follows:

```
RECT:record
    left: dword;
    top:dword;
    right:dword;
    bottom:dword;
endrecord;
```

Left and top are the coordinates of the upper left corner of a rectangle Right and bottom are the coordinates of the lower right corner. One thing to remember: The origin of the x-y axes is at the upper left corner of the client area. So the point y=10 is BELOW the point y=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.GetClientRect( hWnd, rect );
        win.DrawText
        (
            hdc,
            "Win32 assembly is great and easy!",
            -1,
            rect,
            win.DT_SINGLELINE | win.DT_CENTER | win.DT_VCENTER
        );
        win.EndPaint( hWnd, ps );
```

In response to win.WM_PAINT message, you call win.BeginPaint with handle to the window you want to paint and an uninitialized win.PAINTSTRUCT structure as parameters. After successful call, EAX contains the handle to device context. Next you call win.GetClientRect to retrieve the dimension of the client area. The dimension is returned in rect variable which you pass to win.DrawText as one of its parameters. win.DrawText's syntax is:

```
procedure DrawText
(
    hdc:dword;
    lpString:string;
    nCount:dword;
    var lpRect:dword;
    uFormat:dword
);
```

win.DrawText is a high-level text output API function. It handles some gory details such as word wrap, centering etc. so you could concentrate on the string you want to paint. Its low-level brother, win.TextOut, will be examined in the next tutorial. win.DrawText formats a text string to fit within the bounds of a rectangle. It uses the currently selected font,color and background (in the device context) to draw the text.Lines are wrapped to fit within the bounds of the rectangle. It returns the height of the output text in device units, in our case, pixels. Let's see its parameters:

hdc handle to device context

lpString The pointer to the string you want to draw in the rectangle. The string must be null-terminated else you would have to specify its length in the next parameter, nCount.

nCount The number of characters to output. If the string is null-terminated, nCount must be -1. Otherwise nCount must contain the number of characters in the string you want to draw.

lpRect The pointer to the rectangle (a structure of type win.RECT) you want to draw the string in. Note that this rectangle is also a clipping rectangle, that is, you could not draw the string outside this rectangle.

uFormat The value that specifies how the string is displayed in the rectangle. We use three values combined by "or" operator:

- DT_SINGLELINE specifies a single line of text
- DT_CENTER centers the text horizontally.
- DT_VCENTER centers the text vertically. Must be used with DT_SINGLELINE.

After you finish painting the client area, you must call win.EndPaint function to release the handle to device context.

That's it. We can summarize the salient points here:

- You call win.BeginPaint-win.EndPaint pair in response to win.WM_PAINT message.
- Do anything you like with the client area between the calls to win.BeginPaint and win.EndPaint.
- If you want to repaint your client area in response to other messages, you have two choices:
- Use win.GetDC-win.ReleaseDC pair and do your painting between these calls
- Call win.InvalidateRect or win.UpdateWindow to invalidate the entire client area, forcing Windows to put win.WM_PAINT message in the message queue of your window and do your painting in win.WM_PAINT section