Chapter 6: Text in a GUI World

6.1: Text Display Under Windows

An ancient Chinese proverb tells us that a picture is worth a thousand words. While this may certainly be true in many instances, pictures alone cannot convey information as efficiently as text. Imagine, for example, trying to write this book using only images, no text. So while a picture may be worth a thousand words, sometimes two or three words does the job a whole lot better than those thousand words. Anyone who has used a Windows application suffering from icon overload can appreciate the fact that text is still important in a GUI world.

Of course, text under Windows is a far cry from console or green-screen applications of yesterday. Writing console applications (even under Windows) is a fairly trivial exercise when you view the console display as a glass teletype to which you send a stream of characters. Fancier console applications treat the display as a two-dimensional object, allowing the software to position the cursor and text on the screen. However, such applications generally deal with fixed size (fixed pitch) character cells, so keeping track of the information on the display is still fairly trivial. Windows, of course, supports text of different fonts and sizes and you can finely position text on the display. This vastly complicates the display of text in the system. Another big difference between typical console applications and a Windows application is that the console device itself remembers the text written to the display (at least, for as long as such display is necessary). A Windows application, however, must remember all the text it s written to the display and be ready to redraw that text whenever Windows sends the application a redraw message. As such, a simple text application under Windows can be quite a bit more complex than an equivalent text-based console application.

As you saw in the previous chapter, writing text to a Windows GUI display is not quite as trivial as calling HLA s `stdout.put` procedure. Console output is fundamentally different than text output in a GUI application. In console applications, the output characters are all the same size, they all use the same type style (typeface), and they typically employ a *monospaced font* (that is, each character is exactly the same width) that makes it easy to create columns of text by controlling the number of characters you write after the beginning of the line. The Windows console API automatically handles several control characters such as backspaces, tabs, carriage returns, and linefeeds that make it easy to manipulate strings of text on the display. All the assumptions one can make about how the system displays text in a console application are lost when displaying text in a GUI application. There are three fundamental differences between console applications and GUI applications:

- ¥ Console applications use a fixed font whereas GUI applications allow multiple fonts in a window display.
- ¥ Console applications can assume that character output always occurs in fixed character cells on the display; this simplifies actions like aligning columns of text and predicting the output position of text on the screen. Furthermore, console applications always write lines of text to fixed line positions on the display (whereas GUI apps can write a line of text at any pixel position on the display).
- ¥ Console applications can use special control characters in the output stream, like tab, carriage return, and line feed characters, to control the position of text on the display. GUI applications have to explicitly position this text on the display.

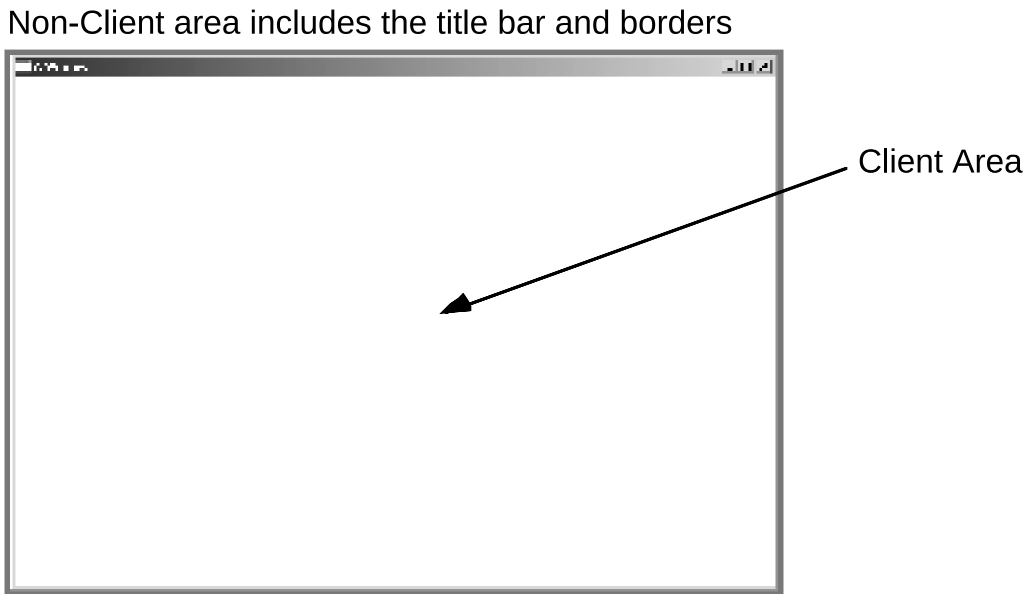
Because of the simplifying nature of text output in a console app, writing GUI apps is going to involve more complexity than writing console apps. The purpose of this chapter is to explain that additional complexity.

6.2: Painting

Although this chapter specifically deals with text output in a GUI application, you must remember that on the display screen there really is no such thing as text; everything is a graphic image. The text that appears on the screen is really nothing more than a graphic image. Therefore, drawing text is really nothing more than a special case of drawing some graphic image on a Windows display. Therefore, before we can begin discussing text output under Windows, we need to first discuss some generic information about drawing objects under Windows.

Windows that an application displays are usually divided into two different areas or *regions*: a *client region* and a *non-client region*. The client region consists of the area in the window that the application is responsible for maintaining. The non-client region is that portion of the window that Microsoft Windows maintains, including the borders, scroll bars, and title bar (see Figure 6-1). Although it is possible for an application to affect the non-client region under some special circumstances, for the most part, all drawing by the application in response to a `w.WM_PAINT` message occurs in the client region.

Figure 6-1: Client Versus Non-Client Areas in a Window



As the previous chapter explains, an application does not arbitrarily draw information in the application s window (client area or otherwise). Instead, Windows sends a message (`w.WM_PAINT`) to the application s window procedure and the window procedure takes the responsibility of (re)drawing the information in the window. One immediate question that should come up is when does Windows notify the application that it needs to redraw the client area? As you saw in the last chapter, one way to force Windows to send this message is by calling the `w.UpdateWindow` API function. When you called `w.UpdateWindow` and pass it the handle of a given window, Microsoft Windows will *invalidate* that window and post a message telling the Window to redraw itself.

Calling the `w.UpdateWindow` function isn t the only way that the contents of some window could become invalid. An application s window could become invalid because the user drags (or opens) some other window over the top of the window and then closes that other window. Because the area covered by the second window in the original application s window now contains the image drawn for that second window, the client area of the original window is *invalid* - it does not contain a valid image for the original window. When this happens, Microsoft Windows will invalidate the original window and post a `w.WM_PAINT` message to that window so it will redraw itself and repair the damage caused by overlaying that second window over the first.

Generally, an application will only call the `w.UpdateWindow` API function to force a redraw of the window from the application s main program. Normally to achieve this, you d call the `w.InvalidateRect` API function:

```
type
   InvalidateRect: procedure
   (
            hWnd :dword;
      var   lpRect :RECT;
            bErase :boolean
   );
   @stdcall;
   @returns( "eax" );
   @external( "__imp__InvalidateRect@12" );
```

The first parameter is the handle of the window you want redrawn (e.g., the application s main window if you want everything redrawn); if you specify NULL here, Windows will redraw all windows. The second parameter is a pointer to a `w.RECT` data structure that specifies the rectangular region of the window you want redrawn. If you specify NULL as this parameter, Windows will tell the application to repaint the entire window; we ll discuss the use of this parameter later (as a means to improve window redrawing performance), for now you can safely specify NULL as the value for this parameter. The third parameter is a boolean value (`true` or `false`) that specifies whether Windows will first erase the background before you actually redraw the window. Again, this is an optimization that can save time redrawing the window by skipping the erase operation. Until you understand how to use this parameter, it s probably safest to pass `true` as this parameter s value.

When a window becomes invalid, Microsoft Windows sends the `w.WM_PAINT` message to the window procedure which, presumably, invokes the code or function that redraws the window. As you saw in the previous chapter, the drawing code is sandwiched between calls to the `w.BeginPaint` and `w.EndPaint` API calls. We re actually going to create a set of macros to handle the calls to `w.BeginPaint` and `w.EndPaint` a little later in this chapter; however, to understand how to write those macros you ll need to understand how these functions operate. But first, we ve got to discuss another Windows object: the device context.

6.2.1: Device Contexts

A device context is a Windows abstraction of an output device that is capable of rendering information sent to a window. Device contexts, or simply DCs, provide a machine-independent view of various display and printer devices allowing an application to write data to a single device yet have that information appear properly on a multitude of different real-world devices. For the purposes of this chapter (and several that follow), the DC will generally represent a video display device; however, keep in mind that much of what this chapter discusses applies to printers and certain other output devices as well.

The actual device context data structure is internal to Windows; an application cannot directly manipulate a DC. Instead, Windows returns a handle to a device context and the application references the DC via this handle. Any modifications an application wants to make to the DC is done via API functions. We ll take a look at some of these functions that manipulate device context attributes in the very next section.

So the first question to ask is how does an application obtain a device context handle for a given window? Well, you ve already seen one way: by calling the `w.BeginPaint` API function. For example, the following code is taken from the `Paint` procedure of the *HelloWorld* program from the previous chapter:

```
   w.BeginPaint( hWnd, ps ); // Returns device context handle in EAX
   mov( eax, hDC );
```

Calls to Win32 API functions that actually draw data on the display will require the device context handle that `w.BeginPaint` returns.

Whenever a window procedure receives a `w.WM_PAINT` message, the window procedure (or some subservient procedure it calls, like the `Paint` procedure in the *HelloWorld* program) typically calls `w.BeginPaint` to get the device context and do other DC initialization prior to actually drawing information to the device context. That device context is valid until the corresponding call to `w.EndPaint`.

Between the `w.BeginPaint` and `w.EndPaint` calls, the window procedure (and any subservient procedures like `Paint`) calls various functions that update the window s client region, like the `w.DrawText` function from the *HelloWorld* program of the previous chapter. An important thing to realize is that these individual calls don t immediately update the display (or whatever physical device is associated with the device context). Instead, Windows stores up the drawing requests (as part of the `w.PAINTSTRUCT` parameter, `ps` in the current example, that you pass to the `w.BeginPaint` API function). When you call the `w.EndPaint` function, Windows will begin updating the physical display device.

One issue with the calls to `w.BeginPaint` and `w.EndPaint` is that Windows assigns a very low priority `w.WM_PAINT` messages. This was done to help prevent a massive number of window redraws in response to every little change the application wants to make to the display. By making the `w.WM_PAINT` message low priority, Windows tends to save up (and coalesce) a sequence of window updates so the window procedure receives only a single `w.WM_PAINT` message in response to several window updates that need to be done. This reduces the number of `w.WM_PAINT` messages sent to the application and, therefore, reduces the number of times that the application redraws its window (thus speeding up the whole process). The only problem with this approach is that sometimes the system is processing a large number of higher-priority messages and the window doesn t get updated for some time. You ve probably seen a case where you ve closed a window and the system doesn t redraw the windows underneath for several seconds. When this happens, Windows (and various Windows applications) are busy processing other higher-priority messages and the `w.WM_PAINT` message has to wait.

On occasion, you ll need to immediately update some window immediately, without waiting for Windows to pass a `w.WM_PAINT` message to your window procedure and without waiting for the application to handle all the other higher-priority messages. You can achieve this by calling the `w.GetDC` and `w.ReleaseDC` API functions. These two functions have the following prototypes:

```
type
   GetDC: procedure
   (
      hWnd :dword
   );
   @stdcall;
   @returns( "eax" );
   @external( "__imp__GetDC@4" );

   ReleaseDC: procedure
   (
      hWnd :dword;
      hDC :dword
   );
   @stdcall;
   @returns( "eax" );
   @external( "__imp__ReleaseDC@8" );
```

You use these two functions just like w.BeginPaint and w.EndPaint (respectively) with just a couple differences. First of all, you don t pass these two parameters a w.PAINTSTRUCT parameter. This is because all calls to the drawing routines between these two functions immediately update the window (rather than storing those drawing operations up in a w.PAINTSTRUCT object). The second major difference is that you can call w.GetDC and w.ReleaseDC anywhere you have a valid window handle, not just in the section of code that handles the w.WM_PAINT message.

There is another variant of w.GetDC, w.GetWindowDC that returns a device context for the entire window, including the non-client areas. An application can use this API function to draw into the non-client areas. Generally, Microsoft recommends against drawing in the non-client area, but if you have a special requirement (like drawing a special image within the title bar), then this function is available. You use w.GetWindowDC just like w.GetDC. Like the call to w.GetDC, you must call the w.ReleaseDC function when you are through drawing to the window. Also like w.GetDC, you can call w.GetWindowDC anywhere, not just when handling a w.WM_PAINT message. Here s the function prototype for the w.GetWindowDC call:

```
type
   GetWindowDC: procedure
   (
      hWnd :dword
   );
   @stdcall;
   @returns( "eax" );
   @external( "__imp__GetWindowDC@4" );
```

6.2.2: Device Context Attributes

Although a DC attempts to abstract away the differences between various physical devices, there is no escaping the fact that behind the abstraction is a real, physical, device. And different real-world devices have some real-world differences that an application cannot simply ignore. For example, some devices (like display adapters) are capable of displaying color information whereas some devices (e.g., most laser printers) are only capable of black & white output. Likewise, some display devices are capable of displaying millions of different colors, while some displays are only capable of placing thousands or hundreds of different colors on the screen simultaneously. Although Windows will attempt to convert data intended for a generic device to each of these specific devices, the actual display (or printout) that the user sees will probably be much better if the application limits itself to the capabilities of the actual physical device. For this reason, Windows provides a set of API calls that let an application determine the capabilities of the underlying physical device.

Some of the attributes associated with a device context exist for convenience, not because of the physical capabilities of the underlying device. For example, the device context maintains default information such as what color to use when drawing text, what background color to use, the current font, the current line drawing style, how to actually draw an image onto the display, and lots of other information. Carrying this information around in the device context saves the programmer from having to pass this information as parameter data to each and every drawing function the application calls. For example, when drawing text one rarely changes the font with each string written to the display. It would be rather painful to have to specify the font information on each call to a function like w.DrawText. Instead, there are various calls you can make to Windows that will set the current device context attributes for things like fonts, colors, and so on, that will take effect on the next API call that draws information to the device context. We ll take a look at some of these functions a little later in this chapter.

6.2.3: Painting Text in the Client Area

Once you obtain a device context (DC) with a call to `w.BeginPaint`, `w.GetDC`, or `w.GetWindowDC`, you may begin drawing in the window. The purpose of this chapter and the next chapter is to introduce you to the various functions you can call to display data in a window. This chapter covers textual output using functions like `w.DrawText` and `w.TextOut`, the next chapter discusses those functions you ll use to draw graphic images in a window.

A little later in this section we ll take a look at the actual functions you can call to place textual data in some window of an application. For now, the important thing to note is that Windows only provides functions that display strings of text. There are no formatting routines like C++ stream I/O (`cut`), C s formatted print (`printf`), or HLA s `stdout.put` that automatically convert various data types to string form upon output. Instead, you must manually convert whatever data you want to display into string form and then write that string to the display. Fortunately, HLA provides a string formatting function whose syntax is nearly identical to `stdout.put` (`str.put`) that lets you easily convert binary data to string form. So to print data in some internal (e.g., integer or floating point) format, you can first call the str.put procedure to convert the data to a string and then call a Win32 API function like `w.DrawText` or `w.TextOut` to display the text in your application s window.

The `str.put` procedure (macro, actually) takes the following form:

str.put(stringVariable, *<<list of items to convert to string form>>*);

If you re familiar with the HLA `stdout.put` procedure, then you ll be right at home with the `str.put` procedure. Except for the presence of a string variable at the beginning of the `str.put` parameter list, you use the `str.put` procedure the same way you use `stdout.put`; the major difference between these two functions is that `str.put` writes its output to the string variable you specify rather than writing it to the standard output device. Here s an example of a typical `str.put` call:

str.put(OutputString, Value of i: , i:4, Value of r: , r:10:2);

The `str.put` procedure converts the items following the first parameter into a string form and then stores the resulting string into the variable you specify as the first parameter in the call to `str.put`. You must ensure that you ve preallocated enough storage for the string to hold whatever data you write to the output string variable. If you re outputting lines of text to the window (one line at a time), then 256 characters is probably a reasonable amount of storage (more than generous, in fact) to allocate for the output string. If you don t have to worry about issues of reentrancy in multi-threaded applications, you can statically allocate a string to hold the output of `str.put` thusly:

```
static
   OutputString: str.strvar( 256 );
```

The `str.init` macro declares `OutputString` as a string variable and initializes it with a pointer to a string buffer capable of holding up to 256 characters. The problem, of course, with using static allocation is that if two threads wind up executing some code that manipulates `OutputString` simultaneously, then your program will produce incorrect output. Although most of the applications you ll write won t be multi-threaded, it s still a good idea to get in the habit of allocating the storage dynamically to avoid problems if you decide to change some existing code to make it multi-threaded in the future.

Of course, the standard way to dynamically allocate storage for a string in HLA is via the `stralloc` function. The only problem with `stralloc` is that it ultimately winds up calling Windows memory management API. While there is nothing fundamentally wrong with doing this, keep in mind two facts: calling Win32 API functions are somewhat expensive (and the memory management calls can be expensive) and we re briefly allocating storage for this string. Once we ve converted our output to string form and displayed it in the window, we don t really need the string data anymore. True, we can use the same string variable for several output operations, but the bottom line is that when our Paint procedure (or whomever is handling the w.WM_PAINT message) returns, that string data is no longer needed and we ll have to deallocate the storage (i.e., another expensive call to the Win32 API via the `strfree` invocation). A more efficient solution is to allocate the string object as local storage within the activation record of the procedure that contains the call to `str.put`. HLA provides two functions to help you do this efficiently: `str.init` and `tstralloc`.

The `str.init` function takes an existing (preallocated) buffer and initializes it for use as a string variable. This function requires two parameters: a buffer and the size (in bytes) of that buffer. This function returns a pointer to the string object it creates within that buffer area and initializes that string to the empty string. You would normally store the return result of this function (in EAX) into an HLA string variable. Here s an example of the use of this function:

```
procedure demoStrInit;
var
   sVar:    string;
   buffer:  char[ 272];
begin demoStrInit;

   str.init( buffer, @size( buffer ));
   mov( eax, sVar );
      .
      .
      .
end demoStrInit;
```

An important thing to note about `str.init` is that the pointer it returns does not contain the address of the first character buffer area whose address you pass as the first parameter. Instead, str.init may skip some bytes at the beginning of the buffer space in order to double-word align the string data, then it sets aside eight bytes for the HLA string s maximum length and current length values (`str.init` also initializes these fields). The `str.init` function returns the address of the first byte beyond these two double-word fields (as per the definition of an HLA string). Once you initialize a buffer for use as a string object via `str.init`, you should only manipulate the data in that buffer using HLA string functions via the string pointer that `str.init` returns (e.g, the sVar variable in this example). One issue of which you should be aware when using the `str.init` function is that the buffer you pass it must contain the maximum number of characters you want to allow in the string *plus sixteen*. That is why buffer in the current example contains 272 characters rather than 256. The `str.init` function uses these extra characters to hold the maximum and current length values, the zero terminating byte, and any padding characters needed to align the string data on a double-word boundary.

One thing nice about the str.init function is that you don t have to worry about deallocating the storage for the string object if both the string variable (where you store the pointer) and the buffer containing the string data are automatic (VAR) variables. The procedure call and return will automatically allocate and deallocate storage for these variables. This scheme is very convenient to use.

Another way to allocate and initialize a string variable is via the `tstralloc` function. This function allocates storage on the stack for a string by dropping the stack point down the necessary number of bytes and initializing the maximum and current length fields of the data it allocates. This function returns a pointer to the string data it creates that you can store into an HLA string variable. Here s an example of the call to the `tstralloc` function:

```
procedure demoTStrAlloc;
var
   s: string;
```

```
begin demoTStrAlloc;

   tstralloc( 256 );
   mov( eax, s );
      .
      .
      .
end demoTStrAlloc;
```

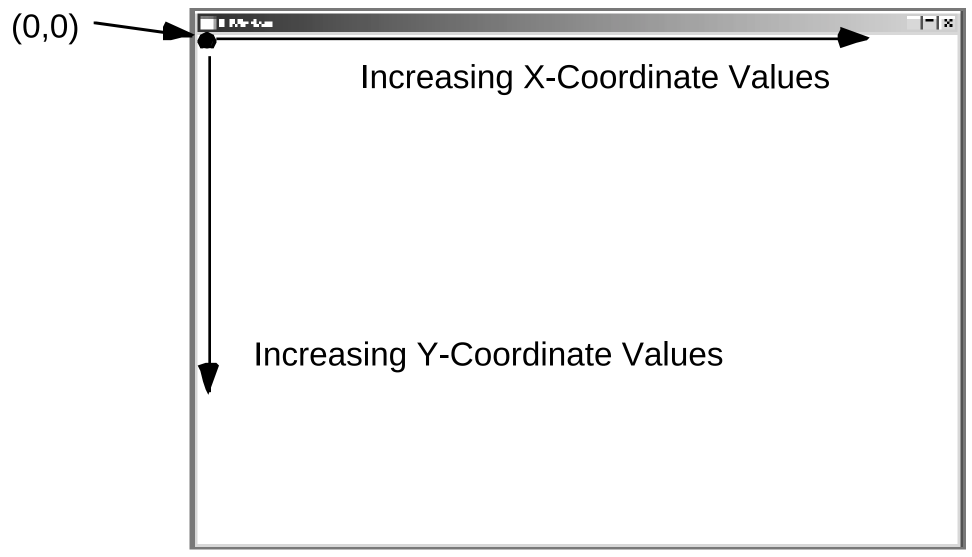
Note that the call to `tstralloc` drops the stack down by as many as 12 bytes beyond the number you specify. In general, you won t know exactly how many bytes by which `tstralloc` will drop the stack. Therefore, it s not a good idea to call this function if you ve got stuff sitting on the stack that you ll need to manipulate prior to returning from the function. Generally, most people call `tstralloc` immediately upon entry into a procedure (after the procedure has built the activation record), before pushing any other data onto the stack. By doing so, the procedure s exit code will automatically deallocate the string storage when it destroys the procedure s activation record. If you ve got some data sitting on the stack prior to calling tstralloc, you ll probably want to save the value in the ESP register prior to calling tstralloc so you can manually deallocate the string storage later (by loading ESP with this value you ve saved).

As you saw in the last chapter, the `w.DrawText` function is capable of rendering textual data in a window. Here s the HLA prototype for this API function:

```
static

   DrawText: procedure
   (
         hDC :dword;
         lpString :string;
         nCount :dword;
      var lpRect :RECT;
         uFormat :dword
   );
   @stdcall;
   @returns( "eax" );
   @external( "__imp__DrawTextA@20" );
```

The `hDC` parameter is a device context handle (that you obtain from the `w.BeginPaint`, or equivalent, function). The `lpString` parameter is a pointer to a zero terminated sequence of characters (e.g., an HLA string) that `w.DrawText` is to display. The `nCount` parameter specifies the length of the string (in characters). If `nCount` contains -1, then `w.DrawText` will determine render all characters up to a zero terminating byte. The `lpRect` parameter is a pointer to a `w.RECT` structure. A `w.RECT` record contains four `int32` fields: `top`, `left`, `bottom`, and `right`. These fields specify the x- and y- coordinates of the top/left hand corner of a rectangular object and the bottom/right hand corner of that object. The `w.DrawText` function will render the text within this rectangle. The coordinates are relative to the client area of the window associated with the device context (see Figure 6-1 concerning the client area). Although Windows supports several different coordinate systems, the default system is what you ll probably expect with coordinate (0,0) appearing in the upper left hand corner of the window with x-coordinate values increasing as you move left and y-coordinate values increasing as you move down (see Figure 6-2).

Figure 6-2: Windows' Default Coordinate System


(0,0)
Increasing X-Coordinate Values
Increasing Y-Coordinate Values

The `w.DrawText` function will format the string within the rectangle the `lpRect` parameter specifies, including wrapping the text if necessary. The `uFormat` parameter is a `uns32` value that specifies the type of formatting that `w.DrawText` is to apply to the text when rendering it. There are a few too many options to list here (see the *User32 API Reference* on the accompanying CD for more details), but a few of the more common formatting options should give you the flavor of what is possible with `w.DrawText`:

¥ w.DT_CENTER: centers the text within the rectangle

¥ w.DT_EXPANDTABS: expands tab characters within the string

¥ w.DT_LEFT: left-aligns the string in the rectangle

¥ w.DT_RIGHT: right-aligns the string in the rectangle

¥ w.DT_SINGLELINE: only allows a single line of text in the rectangle, turns off the processing of carriage returns and line feeds

¥ w.DT_VCENTER: vertically centers the text in the rectangle

¥ w.DT_WORDBREAK: turns on word break - lines are broken on word boundaries when you use this option

These `w.DrawText uFormat` values are bit values that you may combine with the HLA | bitwise-OR operator (as long as the combination makes sense). For example, w.DT_SINGLELINE | w.DT_EXPANDTABS is a perfectly legitimate value to supply for the `uFormat` parameter.

The `w.DrawText` function is very powerful, providing some very powerful formatting options. This formatting capability, however, comes at a cost. Not only is the function a tad bit slow (when it has to do all that formatting), but it is also somewhat inconvenient to use; particularly due to the fact that you must supply a bounding rectangle (which is a pain to set up prior to the call). Without question, the most popular text output routine is the `w.TextOut` function. Here s its prototype:

```
static
```

```
   TextOut:procedure
   (
         hDC       :dword;
         nXStart   :int32;
         nYStart   :int32;
         lpString :string;
         cbString :dword
   );
   @stdcall;
   @returns( "eax" );
   @external( "__imp__TextOutA@20" );
```

As usual, the `hDC` is the device context handle that a function like `w.BeginPaint` returns to specify where `w.TextOut` draws the text. The `nXStart` and `nYStart` values specify where `w.TextOut` will begin placing the text in the window (these coordinates are relative to the client area specified by the `hDC` parameter). The exact nature of these coordinate value depends upon the text-alignment mode maintained by the device context. Generally, these coordinates specify the upper-left-most point of the string when drawn to the window. However, you may change this behavior by calling the `w.SetTextAlign` function and changing this attribute within the device context. The `lpString` parameter is a pointer to a sequence of characters (e.g., an HLA string). Note that you do not have to zero-terminate this string, as you must supply the length of the string (in characters) in the `cbString` parameter. Here s a simple call to the `w.TextOut` API function that display s Hello World near the upper left-hand corner of some window:

w.TextOut(hDC, 10, 10, Hello World , 11);

The `w.TextOut` function does not process any control characters. Instead, it draws them on the screen using various graphic images for each of these character codes. In particular, `w.TextOut` does not recognize tab characters. Windows does provide a variant of `w.TextOut`, `w.TabbedTextOut`, that will expand tab characters. Here s the prototype for this function:

```
static
   TabbedTextOut:
      procedure
      (
               hDC :dword;
               X :dword;
               Y :dword;
               lpString :string;
               nCount :dword;
               nTabPositions :dword;
         var   lpnTabStopPositions :dword;
               nTabOrigin :dword
      );
      @stdcall;
      @returns( "eax" );
      @external( "__imp__TabbedTextOutA@32" );
```

The `hDC, X, Y, lpString`, and `nCount` parameters have the same meaning as the `w.TextOut` parameters (`nCount` is the number of characters in `lpString`). The `nTabPositions` parameter specifies the number of tab positions appearing in the `lpnTabStopPositions` (array) parameter. The `lpnTabStopPositions` parameter is a pointer to the first element of an array of 32-bit unsigned integer values. This array should contain at least `nTabPositions` elements. The array contains the number of pixels to skip to (relative to the value of the last parameter) for each tab stop that `w.TabbedTextOut` encounters in the string. For example, if `nTabPositions` contains 4 and the lpnTabStopPositions array contains 10, 30, 40, and 60, then the tab positions will be at pixels nTabOrigin+10, nTabOrigin+30, nTabOrigin+40, and nTabOrigin+60. As this discussion suggests, the last parameter, nTabOrigin, specifies the pixel position from the start of the string where the tab positions begin.

If the nTabPosition parameter is zero or lpnTabStopPositions pointer is NULL, then w.TabbedTextOut will create a default set of tab stops appearing at an average of eight character positions apart for the current default font. If nTabPosition is one, then w.TabbedTextOut will generate a sequence of tab stops repeating every *n* pixels, where *n* is the value held by the first (or only) element of lpnTabStopPositions. See the following listing for an example of the use of the w.TabbedTextOut function. Figure 6-3 shows the output from this application.

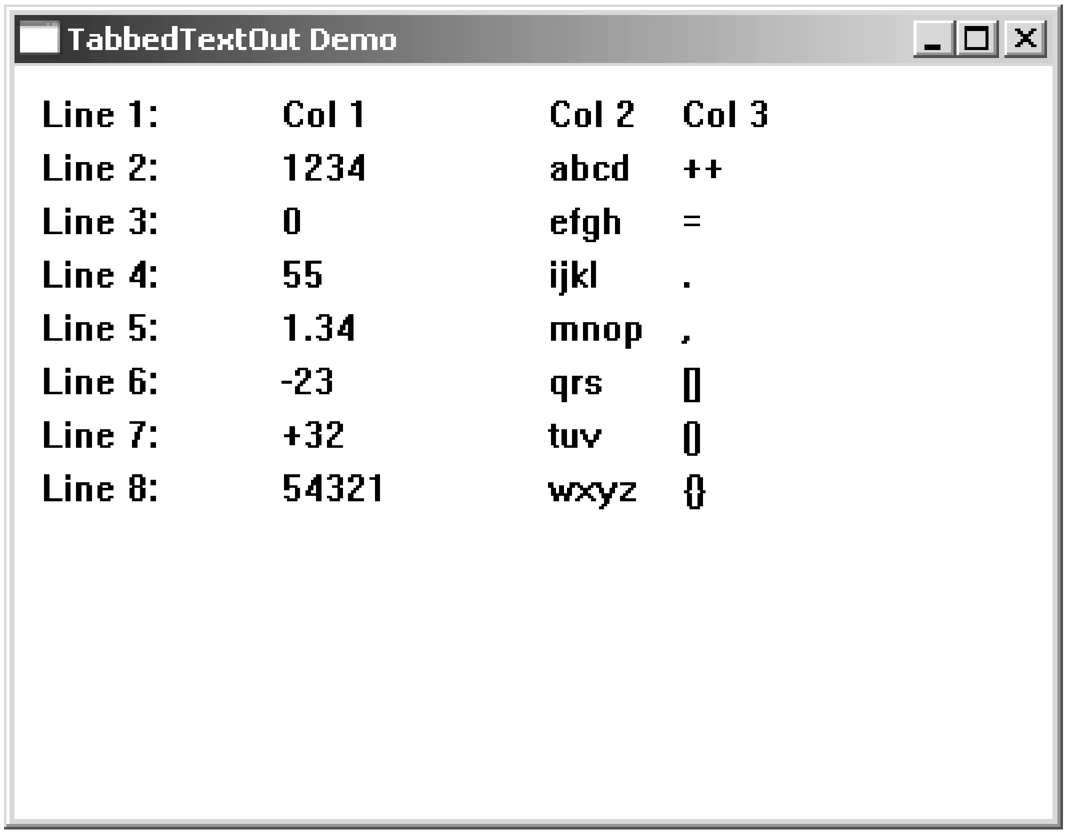
```
// TabbedText.hla:
//
// Demonstrates the use of the w.TabbedTextOut function.

program TabbedText;
#include( "stdlib.hhf" )
#include( "w.hhf" )     // Standard windows stuff.
#include( "wpa.hhf" )   // "Windows Programming in Assembly" specific stuff.
?@nodisplay := true;    // Disable extra code generation in each procedure.
?@nostackalign := true; // Stacks are always aligned, no need for extra code.

const
    tab :text := "#$9"; // Tab character

static
    hInstance:  dword;          // "Instance Handle" supplied by Windows.

    wc:     w.WNDCLASSEX;       // Our "window class" data.
    msg:    w.MSG;              // Windows messages go here.
    hwnd:   dword;              // Handle to our window.


readonly

    ClassName:  string := "TabbedTextWinClass";     // Window Class Name
    AppCaption: string := "TabbedTextOut Demo";     // Caption for Window

// The following data type and DATA declaration
// defines the message handlers for this program.

type
    MsgProc_t:  procedure( hwnd:dword; wParam:dword; lParam:dword );

    MsgProcPtr_t:
        record

            MessageValue:   dword;
            MessageHndlr:   MsgProc_t;

        endrecord;



// The dispatch table:
//
```

```
//  This table is where you add new messages and message handlers
//  to the program.  Each entry in the table must be a tMsgProcPtr
//  record containing two entries: the message value (a constant,
//  typically one of the wm.***** constants found in windows.hhf)
//  and a pointer to a "tMsgProc" procedure that will handle the
//  message.

readonly

    Dispatch:   MsgProcPtr_t; @nostorage;

        MsgProcPtr_t
            MsgProcPtr_t:[  w.WM_DESTROY, &QuitApplication   ],
            MsgProcPtr_t:[  w.WM_PAINT,   &Paint             ],

            // Insert new message handler records here.

            MsgProcPtr_t:[  0, NULL ];   // This marks the end of the list.



/**************************************************************************/
/*          A P P L I C A T I O N   S P E C I F I C   C O D E             */
/**************************************************************************/

// QuitApplication:
//
//  This procedure handles the "wm.Destroy" message.
//  It tells the application to terminate.  This code sends
//  the appropriate message to the main program's message loop
//  that will cause the application to terminate.


procedure QuitApplication( hwnd: dword; wParam:dword; lParam:dword );
@nodisplay;
begin QuitApplication;

    w.PostQuitMessage( 0 );

end QuitApplication;



// Paint:
//
//  This procedure handles the "wm.Paint" message.
//  This procedure displays several lines of text with
//  tab characters embedded in them.

procedure Paint( hwnd: dword; wParam:dword; lParam:dword ); @nodisplay;
const
    LinesOfText := 8;
    NumTabs     := 4;

var
    hdc:    dword;              // Handle to video display device context
    ps:     w.PAINTSTRUCT;      // Used while painting text.
    rect:   w.RECT;             // Used to invalidate client rectangle.
```

```
readonly
    TabStops        :dword[ NumTabs ] := [ 50, 100, 200, 250];
    TextToDisplay   :string[ LinesOfText ] :=
        [
            "Line 1:" tab "Col 1" tab "Col 2" tab "Col 3",
            "Line 2:" tab "1234"  tab "abcd"  tab "++",
            "Line 3:" tab "0"     tab "efgh"  tab "=",
            "Line 4:" tab "55"    tab "ijkl"  tab ".",
            "Line 5:" tab "1.34"  tab "mnop"  tab ",",
            "Line 6:" tab "-23"   tab "qrs"   tab "[]",
            "Line 7:" tab "+32"   tab "tuv"   tab "()",
            "Line 8:" tab "54321" tab "wxyz"  tab "{}"

        ];


begin Paint;

    push( ebx );

    // When Windows requests that we draw the window,
    // fill in the string in the center of the screen.
    // Note that all GDI calls (e.g., w.DrawText) must
    // appear within a BeginPaint..EndPaint pair.

    w.BeginPaint( hwnd, ps );
    mov( eax, hdc );

    for( mov( 0, ebx ); ebx < LinesOfText; inc( ebx )) do

        intmul( 20, ebx, ecx );
        add( 10, ecx );
        w.TabbedTextOut
        (
            hdc,
            10,
            ecx,
            TextToDisplay[ ebx*4 ],
            str.length( TextToDisplay[ ebx*4 ] ),
            NumTabs,
            TabStops,
            0
        );

    endfor;


    w.EndPaint( hwnd, ps );
    pop( ebx );

end Paint;

/**************************************************************************/
/*                  End of Application Specific Code                      */
/**************************************************************************/
```

```
// 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( hwnd:dword; uMsg:uns32; wParam:dword; lParam:dword  );
    @stdcall;
    @nodisplay;
    @noalignstack;

begin WndProc;

    // uMsg contains the current message Windows is passing along to
    // us.  Scan through the "Dispatch" table searching for a handler
    // for this message.  If we find one, then call the associated
    // handler procedure.  If we don't have a specific handler for this
    // message, then call the default window procedure handler function.

    mov( uMsg, eax );
    mov( &Dispatch, edx );
    forever

        mov( (type MsgProcPtr_t [edx]).MessageHndlr, ecx );
        if( ecx = 0 ) then

            // If an unhandled message comes 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.

            w.DefWindowProc( hwnd, uMsg, wParam, lParam );
            exit WndProc;


        elseif( eax = (type MsgProcPtr_t [edx]).MessageValue ) then

            // If the current message matches one of the values
            // in the message dispatch table, then call the
            // appropriate routine.  Note that the routine address
            // is still in ECX from the test above.

            push( hwnd );   // (type tMsgProc ecx)(hwnd, wParam, lParam)
            push( wParam ); //  This calls the associated routine after
            push( lParam ); //  pushing the necessary parameters.
            call( ecx );

            sub( eax, eax ); // Return value for function is zero.
            break;
        endif;
        add( @size( MsgProcPtr_t ), edx );

    endfor;

end WndProc;



// Here's the main program for the application.

begin TabbedText;


    // Set up the window class (wc) object:

    mov( @size( w.WNDCLASSEX ), wc.cbSize );
    mov( w.CS_HREDRAW | w.CS_VREDRAW, wc.style );
    mov( &WndProc, wc.lpfnWndProc );
    mov( NULL, wc.cbClsExtra );
    mov( NULL, wc.cbWndExtra );
    mov( w.COLOR_WINDOW+1, wc.hbrBackground );
    mov( NULL, wc.lpszMenuName );
    mov( ClassName, wc.lpszClassName );

    // Get this process' handle:

    w.GetModuleHandle( NULL );
    mov( eax, hInstance );
    mov( eax, wc.hInstance );

    // Get the icons and cursor for this application:

    w.LoadIcon( NULL, val w.IDI_APPLICATION );
    mov( eax, wc.hIcon );
    mov( eax, wc.hIconSm );

    w.LoadCursor( NULL, val w.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.

    w.RegisterClassEx( wc );

    w.CreateWindowEx
    (
        NULL,
        ClassName,
        AppCaption,
        w.WS_OVERLAPPEDWINDOW,
        w.CW_USEDEFAULT,
        w.CW_USEDEFAULT,
        w.CW_USEDEFAULT,
        w.CW_USEDEFAULT,
        NULL,
        NULL,
        hInstance,
        NULL
    );
    mov( eax, hwnd );

    w.ShowWindow( hwnd, w.SW_SHOWNORMAL );
    w.UpdateWindow( hwnd );

    // Here's the event loop that processes messages
    // sent to our window.  On return from GetMessage,
    // break if EAX contains false and then quit the
    // program.

    forever

        w.GetMessage( msg, NULL, 0, 0 );
        breakif( !eax );
        w.TranslateMessage( msg );
        w.DispatchMessage( msg );

    endfor;

    // The message handling inside Windows has stored
    // the program's return code in the wParam field
    // of the message.  Extract this and return it
    // as the program's return code.

    mov( msg.wParam, eax );
    w.ExitProcess( eax );

end TabbedText;
```

Figure 6-3: TabbedText.HLA Output



```
Line 1:    Col 1      Col 2   Col 3
Line 2:    1234       abcd    ++
Line 3:    0          efgh    =
Line 4:    55         ijkl    .
Line 5:    1.34       mnop    ,
Line 6:    -23        qrs     []
Line 7:    +32        tuv     ()
Line 8:    54321      wxyz    {}
```

The Microsoft Windows API also provides an extended version of the `w.TextOut` function, appropriately named `w.ExtTextOut`. This function uses a couple of additional parameters to specify some additional formatting options. Here s the prototype for the `w.ExtTextOut` function:

```
static
   ExtTextOut: procedure
   (
            hdc          :dword;
            x            :dword;
            y            :dword;
            fuOptions    :dword;
      var   lprc         :RECT;
            lpString     :string;
            cbCount      :dword;
      var   lpDx         :var
   );
   @stdcall;
   @returns( "eax" );
   @external( "__imp__ExtTextOutA@32" );
```

The `hdc, x, y, lpString`, and `cbCount` parameters are compatible to the parameters you supply to the `w.TextOut` function. The `fuOptions` parameter specifies how to use the application-defined `lprc` rectangle. The possible options include clipping the text to the rectangle, specifying an opaque background using the rectangle, and certain non-western alphabet options. Please consult the GDI32 documentation on the accompanying CD-ROM for more details concerning the `fuOptions` and `lprc` parameters. The `lpDx` array is option. If you pass NULL as this parameter s value, then the `w.ExtTextOut` function will use the default character spacing for each of the characters that it draws to the window. If this pointer is non-NULL, then it must point to an array of `uns32` values containing the same number of elements as there are characters in the string. These array elements specify the spacing between the characters in the string. This feature allows you to condense or expand the characters spacing when the `w.ExtTextOut` function draws those characters to the window. This feature is quite useful for word processors and desktop publishing systems that need to take detailed control over the output of characters on the output medium. This book will tend to favor the `w.TextOut` function over the `w.ExtTextOut` function, because the former is easier to use. For more details concerning the `w.ExtTextOut` function, please see the CD-ROM accompanying this book.

As noted in the previous section, the device context maintains certain default values so that you do not have to specify these values on each and every call to various GDI functions. One important attribute for text output is the *text align* attribute. This attribute specifies how functions like `w.TextOut` and `w.ExtTextOut` interpret the coordinate values you pass as parameters. The text align attributes (kept as a bitmap) are the following:

¥ w.TA_BASELINE - The reference point will be on the base line of the text

¥ w.TA_BOTTOM - The reference point will be on the bottom line of the bounding rectangle surrounding the text

¥ w.TA_TOP - The reference point will be on the top line of the bounding rectangle surrounding the text

¥ w.TA_CENTER - The reference point will be aligned with the horizontal center of the bounding rectangle surrounding the text

¥ w.TA_LEFT - The reference point will be on the left edge of the bounding rectangle surrounding the text

¥ w.TA_RIGHT - The reference point will be on the right edge of the bounding rectangle surrounding the text

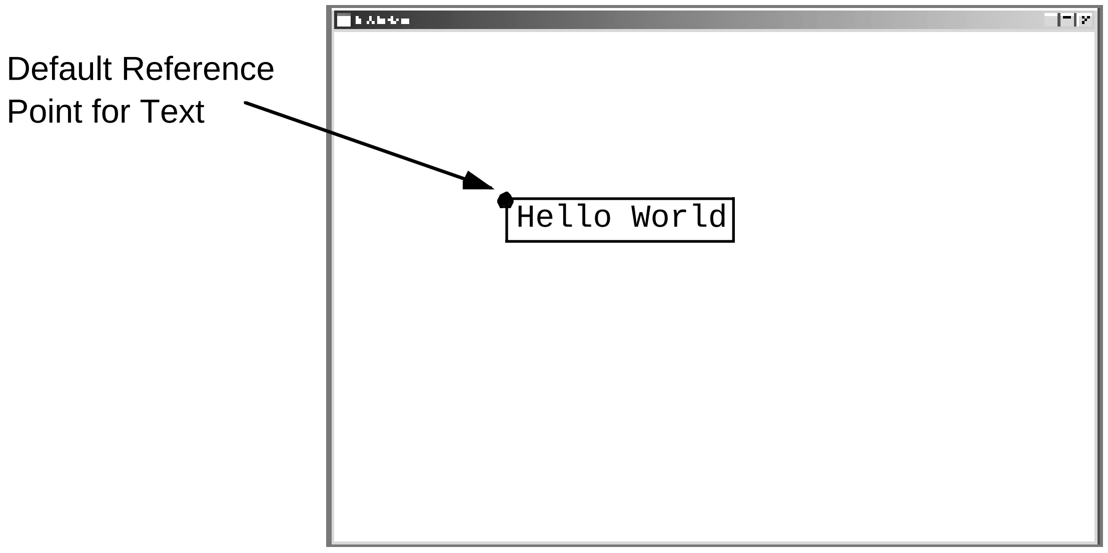
¥ w.TA_NOUPDATECP - The current position is not updated after outputting text to the window

¥ w.TA_RTLREADING - Used only for right-to-left reading text (e.g., middle eastern text)

¥ w.TA_UPDATECP - The output routines update the current position after outputting the text.

These values are all bits in a bit mask and you may combine various options that make sense (e.g., `w.TA_BOTTOM` and `w.TA_TOP` are mutually exclusive). The default values are w.TA_TOP | w.TA_LEFT | w.TA_NOUPDATECP . This means that unless you change the text align attribute value, the x- and y- coordinates you specify in the `w.TextOut` and `w.ExtTextOut` calls supply the top/left coordinate of the bounding rectangle where output is to commence (see Figure 6-4). For right-aligned text, you d normally use the combination `w.TA_RIGHT | w.TA_TOP | w.TA_NOUPDATECP`. Other options are certainly possible, applications that allow the user to align text with other objects would normally make use of these other text alignment options.

Figure 6-4: Default Reference Point for Textual Output



The `w.TA_NOUPDATECP` and `w.TA_UPDATECP` options are fairly interesting. If the w.TA_NOUPDATECP option is set then the application must explicitly state the (x,y) position where Windows begins drawing the text on the display. However, if the `w.TA_UPDATECP` flag is set, then Windows ignores the X- and Y- coordinate values you supply (except on the first call) and Windows automatically updates the cursor position based on the width of the text you write to the display (and any special control characters such as carriage returns and line feeds).

You can retrieve and set the text alignment attribute via the `w.GetTextAlign` and `w.SetTextAlign` API functions. These functions have the following prototypes:

```
static
   GetTextAlign:
      procedure
      (
         hdc :dword
      );
      @stdcall;
      @returns( "eax" );
      @external( "__imp__GetTextAlign@4" );

   SetTextAlign:
      procedure
      (
         hdc :dword;
         fMode :dword
      );
      @stdcall;
      @returns( "eax" );
      @external( "__imp__SetTextAlign@8" );
```

You pass the `w.GetTextAlign` function a device context handle (that you obtain from `w.BeginPaint`) and it returns the attribute bit map value in the EAX register. You pass the `w.SetTextAlign` function the device context handle and the new bitmap value (in the `fMode` parameter). This `fMode` parameter can be any combination of the `w.TA_`*`XXXX`* constants listed earlier; you may combine these constants with the HLA bitwise-OR operator (|), e.g.,

```
// The following statement restores the default text align values:

w.SetTextAlign( hDC, w.TA_TOP | w.TA_LEFT | w.TA_NOUPDATECP );
```

Another set of important text attributes are the text color and background mode attributes. Windows lets you specify a foreground color (that is used to draw the actual characters), a background color (the *background* is a rectangular area surrounding the text) and the *opacity* of the background. You can manipulate these attributes using the following Win32 API functions:

```
static

   GetBkColor:
      procedure
      (
         hdc :dword
      );
      @stdcall;
      @returns( "eax" );
      @external( "__imp__GetBkColor@4" );

   GetTextColor:
      procedure
      (
         hdc :dword
      );
      @stdcall;
      @returns( "eax" );
      @external( "__imp__GetTextColor@4" );

   SetBkColor:
      procedure
      (
         hdc :dword;
         crColor :dword
      );
      @stdcall;
      @returns( "eax" );
      @external( "__imp__SetBkColor@8" );

   SetTextColor:
      procedure
      (
         hdc :dword;
         crColor :COLORREF
      );
      @stdcall;
      @returns( "eax" );
      @external( "__imp__SetTextColor@8" );

   SetBkMode:
      procedure
      (
         hdc :dword;
         iBkMode :dword
      );
      @stdcall;
      @returns( "eax" );
      @external( "__imp__SetBkMode@8" );

   GetBkMode:
      procedure
      (
         hdc :dword
      );
      @stdcall;
      @returns( "eax" );
      @external( "__imp__GetBkMode@4" );
```

As is typical for most GDI (Windows Graphic Device Interface) functions, these functions all take a parameter that is the handle for the current device context (`hdc`); as such, you must only call these functions between a `w.BeginPaint` and `w.EndPaint` sequence (or `w.GetDC/w.GetWindowDC` and `w.ReleaseDC` sequence). The `w.GetXXXX` functions return a color or mode value in EAX. The `w.SetXXXX` functions all take a second dword parameter that specifies the new color or mode attribute.

The color functions (`w.SetTextColor`, `w.SetBkColor`, `w.GetTextColor`, and `w.GetBkColor`) work with 24-bit RGB (red-green-blue) values. An RGB value consists of three eight-bit values specifying shades for red, green, and blue that Windows will use to approximate your desired color on the output device. The blue value appears in the low-order eight bits, the green value appears in bits 8-15 of the 24-bit value, and the red value appears in bits 16-23. The `w.GetXXXX` functions return an RGB value in the EAX register. The `w.SetXXXX` functions pass the 24-bit RGB value as a double word parameter (the high-order eight bits of the double word should contain zero). The *wpa.hhf* header file contains a macro (RGB) that accepts three eight-bit constants and merges them together to produce a 24-bit RGB value. You can use this macro in the `w.SetTextColor` and `w.SetBkColor` function calls thusly:

```
   w.SetTextColor( hdc, RGB( redValue, greenValue, blueValue ));
   w.SetBkColor( hdc, RGB( $FF, 0, 0 ));
```

It is important to remember that RGB is a macro that only accepts constants; you cannot supply variables, registers, or other non-constant values to this macro. It s easy enough to write a function to which you could pass three arbitrary eight-bit values, but merging red-green-blue values into a 24-bit is sufficiently trivial that it s best to simply do this operation in-line, e.g.,

```
   mov( blue, ah );
   mov( 0, al );
   bswap( eax );
   mov( red, al );
   mov( green, ah ); // RGB value is now in EAX.
```

The `w.SetTextColor` API function sets the color that Windows uses when drawing text to the device context. Note that not all devices support a 24-bit color space, Windows will approximate the color (using dithering or other techniques) if the device does not support the actual color you specify.

The `w.SetBkColor` function sets the background that Windows draws when rendering text to the device. Windows draws this background color to a rectangle immediately surrounding the output text. Obviously, there should be a fair amount of contrast between the background color you select and the foreground (text) color or the text will be difficult to read.

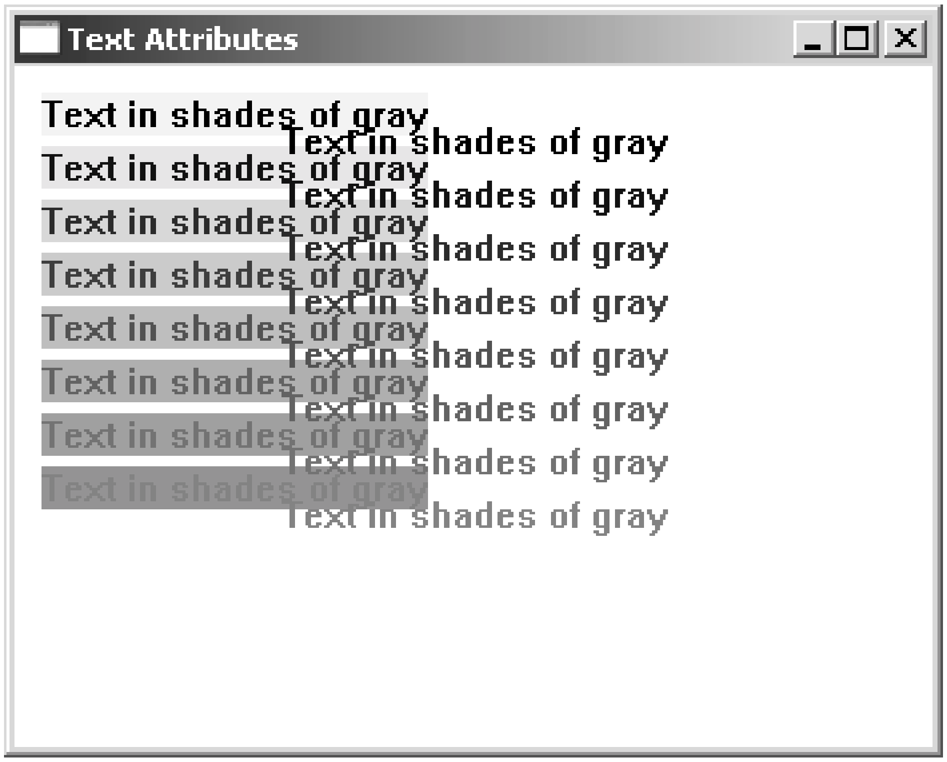
Windows always draws a solid color (or a dithering approximation), never a pattern as the background color for text. By default, Windows always draws the background color (rectangle) prior to rendering the text on the device. However, you can tell Windows to draw only the text without the background color by setting the background mode to transparent . This is achieved via the call to `w.SetBkMode` and passing a device context handle and either the constant `w.TRANSPARENT` (to stop drawing the background rectangle) or `w.OPAQUE` (to start drawing the background rectangle behind the text).

The following listing demonstrates the use of the `w.SetTextColor`, `w.SetBkColor`, and `w.SetBkMode` functions to draw text using various shades of gray. This also demonstrates the `w.OPAQUE` and `w.TRANSPARENT` drawing modes by overlaying some text on the display. The output of this application appears in Figure 6-5.

```
// TextAttr.hla:
//
// Displays text with various colors and attributes.

program TextATtr;
#include( "w.hhf" )      // Standard windows stuff.
#include( "wpa.hhf" )    // "Windows Programming in Assembly" specific stuff.
?@nodisplay := true;     // Disable extra code generation in each procedure.
?@nostackalign := true; // Stacks are always aligned, no need for extra code.

static
    hInstance:  dword;           // "Instance Handle" supplied by Windows.

    wc:     w.WNDCLASSEX;        // Our "window class" data.
    msg:    w.MSG;               // Windows messages go here.
    hwnd:   dword;               // Handle to our window.


readonly

    ClassName:  string := "TextAttrWinClass";       // Window Class Name
    AppCaption: string := "Text Attributes";        // Caption for Window

// The following data type and DATA declaration
// defines the message handlers for this program.

type
    MsgProc_t:  procedure( hwnd:dword; wParam:dword; lParam:dword );

    MsgProcPtr_t:
        record

            MessageValue:   dword;
            MessageHndlr:   MsgProc_t;

        endrecord;



// The dispatch table:
//
//  This table is where you add new messages and message handlers
//  to the program.  Each entry in the table must be a tMsgProcPtr
//  record containing two entries: the message value (a constant,
//  typically one of the wm.***** constants found in windows.hhf)
//  and a pointer to a "tMsgProc" procedure that will handle the
//  message.

readonly

    Dispatch:   MsgProcPtr_t; @nostorage;

        MsgProcPtr_t
            MsgProcPtr_t:[  w.WM_DESTROY, &QuitApplication   ],
            MsgProcPtr_t:[  w.WM_PAINT,   &Paint             ],

            // Insert new message handler records here.

            MsgProcPtr_t:[  0, NULL ];   // This marks the end of the list.



/**************************************************************************/
/*          A P P L I C A T I O N   S P E C I F I C   C O D E            */
/**************************************************************************/

// QuitApplication:
//
//  This procedure handles the "wm.Destroy" message.
//  It tells the application to terminate.  This code sends
//  the appropriate message to the main program's message loop
//  that will cause the application to terminate.


procedure QuitApplication( hwnd: dword; wParam:dword; lParam:dword );
@nodisplay;
begin QuitApplication;

    w.PostQuitMessage( 0 );

end QuitApplication;


// Paint:
//
//  This procedure handles the "wm.Paint" message.
//  This procedure displays several lines of text with
//  different colors.

procedure Paint( hwnd: dword; wParam:dword; lParam:dword ); @nodisplay;
const
    LinesOfText := 8;

    TextToDisplay :string := "Text in shades of gray";
    OverlaidText  :string := "Overlaid Text";

    fgShades :w.RGBTRIPLE[  LinesOfText ]  :=
        [
            w.RGBTRIPLE:[  0, 0, 0],
            w.RGBTRIPLE:[  $10, $10, $10 ],
            w.RGBTRIPLE:[  $20, $20, $20 ],
            w.RGBTRIPLE:[  $30, $30, $30 ],
            w.RGBTRIPLE:[  $40, $40, $40 ],
            w.RGBTRIPLE:[  $50, $50, $50 ],
            w.RGBTRIPLE:[  $60, $60, $60 ],
            w.RGBTRIPLE:[  $70, $70, $70 ]
        ];

    bgShades : w.RGBTRIPLE[  LinesOfText ] :=
        [
            w.RGBTRIPLE:[  $F0, $F0, $F0 ],
            w.RGBTRIPLE:[  $E0, $E0, $E0 ],
            w.RGBTRIPLE:[  $D0, $D0, $D0 ],
            w.RGBTRIPLE:[  $C0, $C0, $C0 ],
            w.RGBTRIPLE:[  $B0, $B0, $B0 ],
            w.RGBTRIPLE:[  $A0, $A0, $A0 ],
            w.RGBTRIPLE:[  $90, $90, $90 ],
            w.RGBTRIPLE:[  $80, $80, $80 ]
        ];


var
    hdc:    dword;              // Handle to video display device context
    ps:     w.PAINTSTRUCT;      // Used while painting text.
    rect:   w.RECT;             // Used to invalidate client rectangle.

begin Paint;

    // When Windows requests that we draw the window,
    // fill in the string in the center of the screen.
    // Note that all GDI calls (e.g., w.DrawText) must
    // appear within a BeginPaint..EndPaint pair.

    w.BeginPaint( hwnd, ps );
    mov( eax, hdc );

    w.SetBkMode( hdc, w.OPAQUE );
    #for( i := 0 to LinesOfText-1 )

        w.SetTextColor
        (
            hdc,
            RGB
            (
                fgShades[ i] .rgbtRed,
                fgShades[ i] .rgbtGreen,
                fgShades[ i] .rgbtBlue
            )
        );

        w.SetBkColor
        (
            hdc,
            RGB
            (
                bgShades[ i] .rgbtRed,
                bgShades[ i].rgbtGreen,
                bgShades[ i].rgbtBlue
            )
        );
        w.TextOut( hdc, 10, i*20+10, TextToDisplay, @length( TextToDisplay ));

    #endfor

    w.SetBkMode( hdc, w.TRANSPARENT );
    #for( i := 0 to LinesOfText-1 )

        w.SetTextColor
        (
            hdc,
            RGB
            (
                fgShades[ i].rgbtRed,
                fgShades[ i].rgbtGreen,
                fgShades[ i].rgbtBlue
            )
        );
        w.TextOut( hdc, 100, i*20+20, TextToDisplay, @length( TextToDisplay ));

    #endfor



    w.EndPaint( hwnd, ps );

end Paint;

/**************************************************************************/
/*                   End of Application Specific Code                     */
/**************************************************************************/




// 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( hwnd:dword; uMsg:uns32; wParam:dword; lParam:dword  );
    @stdcall;
    @nodisplay;
    @noalignstack;

begin WndProc;

    // uMsg contains the current message Windows is passing along to
    // us.  Scan through the "Dispatch" table searching for a handler
    // for this message.  If we find one, then call the associated
    // handler procedure.  If we don't have a specific handler for this
    // message, then call the default window procedure handler function.

    mov( uMsg, eax );
    mov( &Dispatch, edx );
    forever

        mov( (type MsgProcPtr_t [ edx] ).MessageHndlr, ecx );
        if( ecx = 0 ) then

            // If an unhandled message comes 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.

            w.DefWindowProc( hwnd, uMsg, wParam, lParam );
            exit WndProc;


        elseif( eax = (type MsgProcPtr_t [ edx] ).MessageValue ) then

            // If the current message matches one of the values
            // in the message dispatch table, then call the
            // appropriate routine.  Note that the routine address
            // is still in ECX from the test above.

            push( hwnd );   // (type tMsgProc ecx)(hwnd, wParam, lParam)
            push( wParam ); //  This calls the associated routine after
            push( lParam ); //  pushing the necessary parameters.
            call( ecx );

            sub( eax, eax ); // Return value for function is zero.
            break;

        endif;
        add( @size( MsgProcPtr_t ), edx );

    endfor;

end WndProc;



// Here's the main program for the application.

begin TextATtr;


    // Set up the window class (wc) object:

    mov( @size( w.WNDCLASSEX ), wc.cbSize );
    mov( w.CS_HREDRAW | w.CS_VREDRAW, wc.style );
    mov( &WndProc, wc.lpfnWndProc );
    mov( NULL, wc.cbClsExtra );
    mov( NULL, wc.cbWndExtra );
    mov( w.COLOR_WINDOW+1, wc.hbrBackground );
    mov( NULL, wc.lpszMenuName );
    mov( ClassName, wc.lpszClassName );

    // Get this process' handle:

    w.GetModuleHandle( NULL );
    mov( eax, hInstance );
    mov( eax, wc.hInstance );

    // Get the icons and cursor for this application:

    w.LoadIcon( NULL, val w.IDI_APPLICATION );
    mov( eax, wc.hIcon );
    mov( eax, wc.hIconSm );

    w.LoadCursor( NULL, val w.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.

    w.RegisterClassEx( wc );

    w.CreateWindowEx
    (
        NULL,
        ClassName,
        AppCaption,
        w.WS_OVERLAPPEDWINDOW,
        w.CW_USEDEFAULT,
        w.CW_USEDEFAULT,
        w.CW_USEDEFAULT,
        w.CW_USEDEFAULT,
        NULL,
        NULL,
        hInstance,
        NULL
    );
    mov( eax, hwnd );

    w.ShowWindow( hwnd, w.SW_SHOWNORMAL );
    w.UpdateWindow( hwnd );

    // Here's the event loop that processes messages
    // sent to our window.  On return from GetMessage,
    // break if EAX contains false and then quit the
    // program.

    forever

        w.GetMessage( msg, NULL, 0, 0 );
        breakif( !eax );
        w.TranslateMessage( msg );
        w.DispatchMessage( msg );

    endfor;
```

```
    // The message handling inside Windows has stored
    // the program's return code in the wParam field
    // of the message.  Extract this and return it
    // as the program's return code.

    mov( msg.wParam, eax );
    w.ExitProcess( eax );

end TextATtr;
```

Figure 6-5: Output From TextAttr Application



6.2.4: BeginPaint, EndPaint, GetDC, GetWindowDC, and ReleaseDC Macros

One minor issue concerns the use of GDI and Windows User Interface API functions like `w.TextOut` - they are only legal between calls to functions like `w.BeginPaint` and `w.EndPaint` that obtain and release a device context. Furthermore, if an application calls a function like w.BeginPaint, it must ensure that it calls the corresponding function to release the DC (e.g., w.EndPaint). Unfortunately, Windows depends upon programmer discipline to enforce these requirements. Fortunately, this situation is easy to correct in HLA using HLAs *context-free macro* facilities.

An HLA context-free macro invocation always consists of at least two keywords: an initial macro invocation and a corresponding *terminator* invocation. Between these two invocations, HLA maintains a macro context that allows these two macros (as well as other *keyword* macros) to communicate information. The initial invocation and the terminator invocation bracket a sequence of statements, much like the HLA `begin/end while/endwhile, repeat/until`, and `for/endfor` reserved word pairs. An HLA context-free macro declaration takes the following form:

```
#macro macroName(optionalParameters) :<<optional local symbol definitions>>;
   <<Text to expand on macroName invocation>>


   // #keyword is optional, and you may have multiple keyword macros:

   #keyword keywordName(optionalParameters) :<<optional local symbol definitions>>;
      <<Text to expand on keywordName invocation>>

   // Must have exactly one #terminator declaration if this is a context-free
   // macro:

   #terminator terminatorName(optionalParameters) :<<optional local symbols>>;
      <<Text to expand on terminatorName invocation>>
#endmacro
```

For more information about the syntax of an HLA context-free macro declaration, please see the HLA reference manual (it appears on the CD-ROM accompanying this book).

Whenever you invoke a context-free macro, you must also invoke the corresponding terminator associated with that context-free macro. If you fail to do this, HLA will report an error. For example, we can create a simple `BeginPaint` and `EndPaint` context-free macro using the following HLA declaration:

```
#macro BeginPaint( _hwnd, _ps );
   w.BeginPaint( _hwnd, _ps );

#terminator EndPaint( _hwnd, _ps );
   w.EndPaint( _hwnd, _ps );

#endmacro
```

Now you may invoke BeginPaint and EndPaint as follows:

```
BeginPaint( hWnd, ps );
   mov( eax, hdc );
      .
      .
      .
   << code that uses the device context >>
      .
      .
      .
EndPaint( hWnd, ps );
```

Although this macro saves you a small amount of typing (not having to type the w. in front of the `w.BeginPaint` and `w.EndPaint` calls), saving some typing is not the purpose of this macro. Helping you produce correct code is the real purpose behind this macro invocation. As it turns out, HLA will not allow you to invoke the `EndPaint` macro without an earlier invocation of the `BeginPaint` macro. Likewise, if you fail to provide a call to the `EndPaint` macro, HLA will complain that it s missing (much like HLA complains about a missing endif or endwhile). Therefore, it s a little safer to use the `BeginPaint/EndPaint` macros rather than directly calling the `w.BeginPaint` and `w.EndPaint` API functions. This safety is the primary reason for using these macros.

Now saving typing isn t a bad thing, mind you. In fact, with a few minor changes to our macros, we can save a bit of typing. Consider the following macro declaration:

```
#macro BeginPaint( _hwnd, _ps, _hdc );
   w.BeginPaint( _hwnd, _ps );
   mov( eax, _hdc );

#terminator EndPaint;
   w.EndPaint( _hwnd, _ps );

#endmacro;
```

With this macro declaration, you can invoke `BeginPaint` and `EndPaint` thusly:

```
BeginPaint( hWnd, ps, hDC );
       .
       .
       .
   << Code that uses the device context (hDC) >>
       .
       .
       .
EndPaint;
```

Note the convenience that this macro provides - it automatically supplies the parameters for the `w.EndPaint` call (because they re the same parameters you pass to `BeginPaint`, this context-free macro passes those same parameters on through to the `w.EndPaint` call). This is useful because you avoid the mistake of supplying an incorrect parameter to `w.EndPaint` (i.e., you don t have to worry about keeping the parameters passed to `w.BeginPaint` and `w.EndPaint` consistent).

HLAs context-free macros also allow you to create `#keyword` macros. These are macros that you may only invoke between the corresponding initial invocation and the `#terminator` macro invocation. Note that HLA will generate an error if you attempt to invoke one of these macros outside the initial/terminator macro invocation sequence. This works out perfectly for the GDI/User Interface API calls that reference a device context handle because you may only call these functions between a pair of calls that allocate and release a device context (like `w.BeginPaint` and `w.EndPaint`). By defining `#keyword` macros for all these functions, we can ensure that you may only call them between the `BeginPaint` and `EndPaint` macro invocations. A side benefit to defining these API calls this way is that we can drop a parameter (hDC) to each of these functions and have the macro automatically supply this parameter for us. Consider the following sequence that defines macros for the `w.TextOut and` `w.TabbedTextOut` API calls:

```
#macro BeginPaint( _hwnd, _ps, _hdc );
   w.BeginPaint( _hwnd, _ps );
   mov( eax, _hdc );

  #keyword TextOut( _x, _y, _str, _len);
   w.TextOut( _hdc, _x, _y, _str, _len );

  #keyword TabbedTextOut( _x, _y, _str, _len, _tabCnt, _tabs, _offset );
   w.TabbedTextOut( _hdc, _x, _y, _str, _len, _tabCnt, _tabs, _offset );

  #terminator EndPaint;
   w.EndPaint( _hwnd, _ps );
```

```
    #endmacro;
```

The following is the `Paint` procedure from the `TabbedTextDemo` program (appearing earlier in this chapter) that demonstrates the use of the `BeginPaint`, `TabbedTextOut`, and `EndPaint` macro invocations:

```
procedure Paint( hwnd: dword; wParam:dword; lParam:dword ); @nodisplay;
const
    LinesOfText := 8;
    NumTabs     := 4;

var
    hdc:    dword;              // Handle to video display device context
    ps:     w.PAINTSTRUCT;      // Used while painting text.
    rect:   w.RECT;             // Used to invalidate client rectangle.

readonly
    TabStops        :dword[ NumTabs ] := [ 50, 100, 200, 250];
    TextToDisplay   :string[ LinesOfText ] :=
        [
            "Line 1:" tab "Col 1" tab "Col 2" tab "Col 3",
            "Line 2:" tab "1234"  tab "abcd"  tab "++",
            "Line 3:" tab "0"     tab "efgh"  tab "=",
            "Line 4:" tab "55"    tab "ijkl"  tab ".",
            "Line 5:" tab "1.34"  tab "mnop"  tab ",",
            "Line 6:" tab "-23"   tab "qrs"   tab "[]",
            "Line 7:" tab "+32"   tab "tuv"   tab "()",
            "Line 8:" tab "54321" tab "wxyz"  tab "{}"

        ];


begin Paint;

    push( ebx );

    // When Windows requests that we draw the window,
    // fill in the string in the center of the screen.
    // Note that all GDI calls (e.g., TabbedTextOut) must
    // appear within a BeginPaint..EndPaint pair.

    BeginPaint( hwnd, ps, hdc );

        for( mov( 0, ebx ); ebx < LinesOfText; inc( ebx )) do

            intmul( 20, ebx, ecx );
            add( 10, ecx );
            TabbedTextOut
            (
                10,
                ecx,
                TextToDisplay[ ebx*4 ],
                str.length( TextToDisplay[ ebx*4 ] ),
                NumTabs,
                TabStops,
                0
            );

        endfor;


    EndPaint;
    pop( ebx );

end Paint;
```

Note the use of the HLA convention of intending the source code between an initial macro and a terminator macro, much like you would indent text between an `if/endif` or `while/endwhile` pair.

On the CD-ROM accompanying this book, you ll find a *wpa.hhf* header file containing the declarations specific to this book. The `BeginPaint/EndPaint` macro, along with a large percentage of the API calls you d normally make between the calls to `w.BeginPaint` and `w.EndPaint`, appear in that header file. By simply including that header file in your Windows assembly applications, you may use these macros. Also note that this header file supplies context-free macro declarations for `GetDC/ReleaseDC` and `GetWindowsDC/ReleaseDC`.

One thing to keep in mind about the `BeginPaint/EndPaint` macro sequence: you must statically nest the API function calls between a `BeginPaint..EndPaint` sequence. Though this is, by far, the most common way you ll call all these API functions (that is, the call to `w.BeginPaint` appears first, followed in the source code by various API calls that require the device context, and then, finally, the call to `w.EndPaint`), Windows doesn t require this organization at all. Windows only requires that your call `w.BeginPaint` at some point in time before you make any of the other API calls and that you call `w.EndPaint` at some point in time after you make all those other API calls. There is no requirement that this sequence of statement occur in some linear order in your source file, e.g., the following is perfectly legal (though not very reasonable):

```
      jmp doBegin;
doEnd:
      w.EndPaint( hWnd, ps );
      jmp EndOfProc
APICalls:
      w.TextOut( hdc, x, y, stringToPrint, strlen );
      jmp doEnd;

doBegin:
      w.BeginPaint( hWnd, ps );
      mov( eax, hdc );
      jmp APICalls;

EndOfProc:
```

Another thing you can do with the standard Windows API calls is bury the calls to `w.BeginPaint, w.EndPaint`, and any of your API calls requiring the device context inside different procedures. It is only the sequence of the calls that matters to Windows. HLA, on the other hand, requires that the invocations of the initial macro, the keyword macros and the terminator macros occur in a linear fashion within the same function. Fortunately, this isn t much of a restriction because almost all the time this is exactly how you will call these functions.

Because of their safety and convenience, this book will use these context-free macros through the remaining example programs in this book. You should consider using them in your applications for these same reasons. Of course, if you need to write one of these rare applications that needs to call these API functions in a different order (within your source file), there is nothing stopping you from calling the original Windows functions to handle this task.

The *wpa.hhf* header file defines `#keyword` macros for all the GDI/User Interface API functions that this book uses (plus a few others). It may not define all the functions you might want to call between `BeginPaint` and `EndPaint`. However, extending these macros is a trivial process; don t feel afraid to add your own `#keyword` entries to the macros in the *wpa.hhf* header file.

6.3: Device Capabilities

Although Windows attempts to abstract away differences between physical devices so that you may treat all devices the same, the fact is that your applications will have to be aware of many of the underlying characteristics of a given device in order to properly manipulate that device. Examples of important information your applications will need include the size of the device (both the number of displayable pixels and the drawing resolution), the number of colors the device supports, and the aspect ratio the device supports. Windows provides a function named `w.GetDeviceCaps` that returns important device context information for a given device. Here s the prototype for the w.GetDeviceCaps function:

```
static
        GetDeviceCaps: procedure
        (
                hdc     :dword;
                nIndex  :dword
        );
                @stdcall;
                @returns( "eax" );
                @external( "__imp__GetDeviceCaps@8" );
```

The `hdc` parameter is a handle that specifies a device context (and, therefore, indirectly the device); like all other functions that expect a device context handle, you should only call this function between the BeginPaint and EndPaint invocations (or some other device context pair, such as GetDC/ReleaseDC). Note that the `BeginPaint/EndPaint, GetDC/ReleaseDC`, and `GetWindowDC/ReleaseDC` macros in the *wpa.hhf* header file include a `#keyword` macro for `GetDeviceCaps` so you can invoke this macro directly between these pairs of *wpa.hhf* macro invocations.

The `hIndex` parameter is a special Windows value that tells the `w.GetDeviceCaps` function. Table 6-1 lists some of the possible constants that you supply and the type of information this function returns in the EAX register (many of these values we will discuss in the next chapter, so if an entry in the table doesn t make sense to you, ignore it for now).

Table 6-1: Common GetDeviceCaps nIndex Values

Index	Value Returned in EAX
w.HORZSIZE	The horizontal size, in millimeters, of the display (or other device).
w.VERTSIZE	The vertical size, in millimeters, of the display (or other device).
w.HORZRES	The display’s width, in pixels.
w.VERTRES	The display’s height, in pixels.

Index	Value Returned in EAX
w.LOGPIXELSX	The resolution, in pixels per inch, along the displays' X-axis.
w.LOGPIXELSY	The resolution, in pixels per inch, along the displays' Y-axis.
w.BITSPIXEL	The number of bits per pixel (specifying color information).
w.PLANES	The number of color planes.
w.NUMBRUSHES	The number of device-specific brushes available.
w.NUMPENS	The number of device-specific pens available.
w.NUMFONTS	The number of device specific fonts available.
w.NUMCOLORS	Number of entries in the device's color table.
w.ASPECTX	Relative width of a device pixel used for drawing lines.
w.ASPECTY	Relative height of a device pixel used for drawing lines.
w.ASPECTXY	Diagonal width of the device pixel used for line drawing (45 degrees).
w.CLIPCAPS	This is a flag value that indicates whether the device can clip images to a rectangle. The w.GetDeviceCaps function returns one if the device supports clipping, zero otherwise.
w.SIZEPALETTE	Number of entries in the system palette (valid only if the display device uses a palette).
w.NUMRESERVED	Number of system reserved entries in the system palette (valid only if the display device uses a palette).
w.COLORRES	Actual color resolution of the device, in bits per pixel. For device drivers that use palettes.
w.PHYSICALWIDTH	For printing devices, the physical width of the output device in whatever units that device uses.
w.PHYSICALHEIGHT	For printing devices, the physical height of the output device.
w.PHYSICALOFFSETX	For printing devices, the horizontal margin.
w.PHYSICALOFFSETY	For printing devices, the vertical margin.
w.SCALINGFACTORX	For printing devices, the scaling factor along the X-axis.
w.SCALINGFACTORY	For printing devices, the scaling factor along the Y-axis.
w.VREFRESH	For display devices only: the current vertical refresh rate of the device, in Hz.
w.DESKTOPHORZRES	Width, in pixels, of the display device. May be larger than w.HORZRES if the display supports virtual windows or more than one display.
w.DESKTOPVERTRES	Height, in pixels, of the display device. May be larger than w.VERTRES if the display supports virtual windows or more than one display.

Index	Value Returned in EAX
w.BITALIGNMENT	Preferred horizontal drawing alignment. Specifies the drawing alignment, in pixels, for best drawing performance. May be zero if the hardware is accelerated or the alignment doesn’t matter.

The `w.HORZRES` and `w.VERTRES` return values are particularly useful for sizing your windows when your application first starts. You can use these values to determine the size of the display so you can make sure that your window fits entirely within the physical screen dimensions (there are few things more annoying to a user than an application that opens up with the windows borders outside the physical screen area so they cannot easily resize the windows).

The `w.LOGPIXELSX`, `w.HORZSIZE`, `w.LOGPIXELSY`, and `w.VERTSIZE` constants let you request the physical size of the display using real-world units (millimeters and inches; it is interesting that Windows mixes measurement systems here). By querying these values, you can design your software so that information appears on the display at approximately the same size it will print on paper. The return values that Windows supplies via `w.GetDeviceCaps` for these constants are very important for applications that want to display information in a WYSIWYG (what-you-see-is-what-you-get) format.

The `w.BITSPIXEL` `w.PLANES`, `w.NUMCOLORS`, `w.SIZEPALETTE`, `w.NUMRESERVED`, and `w.COLORRES` index values let you query Windows about the number of colors the device can display. Note that not all of these return values make sense for all devices. Some display devices, for example, support a limited number of colors (e.g., 256) and use a pixel s value (e.g., eight-bits) as an index into a look-up table known as a `palette`. The system extracts a 24-bit color value from this lookup table/palette. Other display devices may provide a full 24 bits of color information for each pixel and, therefore, don t use a palette. Obviously, the `w.GetDeviceCaps` queries that return palette information don t return meaningful information on systems whose display card doesn t have a palette.

The `w.NUMBRUSHES`, `w.NUMPENS`, and `w.NUMFONTS` queries request information about the number of GDI objects that the device context currently supports. We ll return to the use of `w.NUMBRUSHES` and `w.NUMPENS` in the next chapter, we ll take a look at the use of `w.NUMFONTS` in the next section.

The `w.ASPECTX` and `w.ASPECTY` queries returns a couple of very important values for your program. These values, taken together, specify the *aspect ratio* of the display. On some displays, the width of a pixel is not equivalent to the height of a pixel (that is, they do not have a 1:1 aspect ratio). If you draw an object on such a display expecting the pixels to have a width that is equivalent to the height, your images will appear squashed or stretched along one of the axes. For example, were you to draw a circle on such a display, it would appear as an oval on the actual display device. By using the `w.GetDeviceCaps` return values for the `w.ASPECTX` and `w.ASPECTY` queries, you can determine by how much you need to squash or stretch the image to undo the distortion the display creates.

The `w.GetDeviceCaps` call returns several other values beyond those mentioned here, we ve just touched on some of the major return values in this section. Because these return values are static (that is, they don t change while the system is operational), you might wonder why Windows doesn t simply return all of these values with a single call (i.e., storing the return values in a record whose address you pass to the `w.GetDeviceCaps` call). The answer is fairly simple: the single-value query method is very easy to expand without breaking existing code. Nevertheless, it s a bit of a shame to call `w.GetDeviceCaps` over and over again, always returning the same values for a given query index value, while your program is running. So although Windows doesn t provide a single API call that reads all of these values, it s not a major problem for you to write your own function that reads the device capabilities your program needs into a single record so you can access those values within your application without making (expensive) calls to Windows. The following program demonstrates just such a function (this application reads a subset of the `w.GetDeviceCaps` values into a record and then displays those values in a Window when the application receives a w.WM_PAINT message, see Figure 6-6 for the output).

```
// GDCaps.hla:
//
// Displays the device capabilities for the video display device.
// Inspired by "DEVCAPS1.C" by Charles Petzold

program GDCaps;
#include( "w.hhf" )       // Standard windows stuff.
#include( "wpa.hhf" )     // "Windows Programming in Assembly" specific stuff.
#include( "memory.hhf" )  // tstralloc is in here
#include( "strings.hhf" ) // str.put is in here

?@nodisplay := true;      // Disable extra code generation in each procedure.
?@nostackalign := true;   // Stacks are always aligned, no need for extra code.

// dct_t and dcTemplate_c are used to maintain the device context data
// structures throughout this program.  Each entry in the dcTemplate_c
// array specifies one of the values we'll obtain from Windows via
// a GetDeviceCap call.  The fieldName entry should contain the name
// of the device capability field to get from Windows. This name must
// exactly match the corresponding w.GetDeviceCap index value with the
// exception of alphabetic case (it doesn't need to be all upper case)
// and you don't need the leading "w."  The desc entry should contain
// a short description of the field.
//
// This is the only place you'll need to modify this code to retrieve
// and display any of the GetDeviceCap values.

type
    dct_t   :record
        fieldName   :string;
        desc        :string;
    endrecord;

const
    dcTemplate_c :dct_t[] :=
    [
        dct_t:[  "HorzSize",      "Width in millimeters" ],
        dct_t:[  "VertSize",      "Height in millimeters" ],
        dct_t:[  "HorzRes",       "Width in pixels" ],
        dct_t:[  "VertRes",       "Height in pixels" ],
        dct_t:[  "BitsPixel",     "Color Bits/pixel" ],
        dct_t:[  "Planes",        "Color planes" ],
        dct_t:[  "NumBrushes",    "Device brushes" ],
        dct_t:[  "NumPens",       "Device pens" ],
        dct_t:[  "NumFonts",      "Device fonts" ],
        dct_t:[  "NumColors",     "Device colors" ],
        dct_t:[  "AspectX",       "X Aspect value" ],
        dct_t:[  "AspectY",       "Y Aspect value" ],
        dct_t:[  "AspectXY",      "Diag Aspect value" ],
        dct_t:[  "LogPixelsX",    "Display pixels (horz)" ],
        dct_t:[  "LogPixelsY",    "Display pixels (vert)" ],
        dct_t:[  "SizePalette",   "Size of palette" ],
        dct_t:[  "NumReserved",   "Reserved palette entries" ],
        dct_t:[ "ColorRes",     "Color resolution" ]
    ];

    DCfields_c := @elements( dcTemplate_c );



// The deviceCapRecord_t and deviceCapabilities_t types are record objects
// that hold the values we're interested in obtaining from the
// w.GetDevCaps API function. The #for loop automatically constructs all
// the fields of the deviceCapRecord_t record from the dcTemplate_c constant
// above.

type
    deviceCapRecord_t :record

        #for( i in dcTemplate_c )

            @text( i.fieldName ) :int32;

        #endfor

    endrecord;

    deviceCapabilities_t :union

        fields      :deviceCapRecord_t;
        elements    :int32[ DCfields_c ];

    endunion;




static
    hInstance:  dword;          // "Instance Handle" supplied by Windows.

    wc:     w.WNDCLASSEX;       // Our "window class" data.
    msg:    w.MSG;              // Windows messages go here.
    hwnd:   dword;              // Handle to our window.

    // Record that holds the device capabilities that this
    // program uses:

    appDevCaps: deviceCapabilities_t;

readonly

    ClassName:  string := "GDCapsWinClass";         // Window Class Name
    AppCaption: string := "Get Device Capabilities";// Caption for Window




// The following data type and DATA declaration
// defines the message handlers for this program.
```

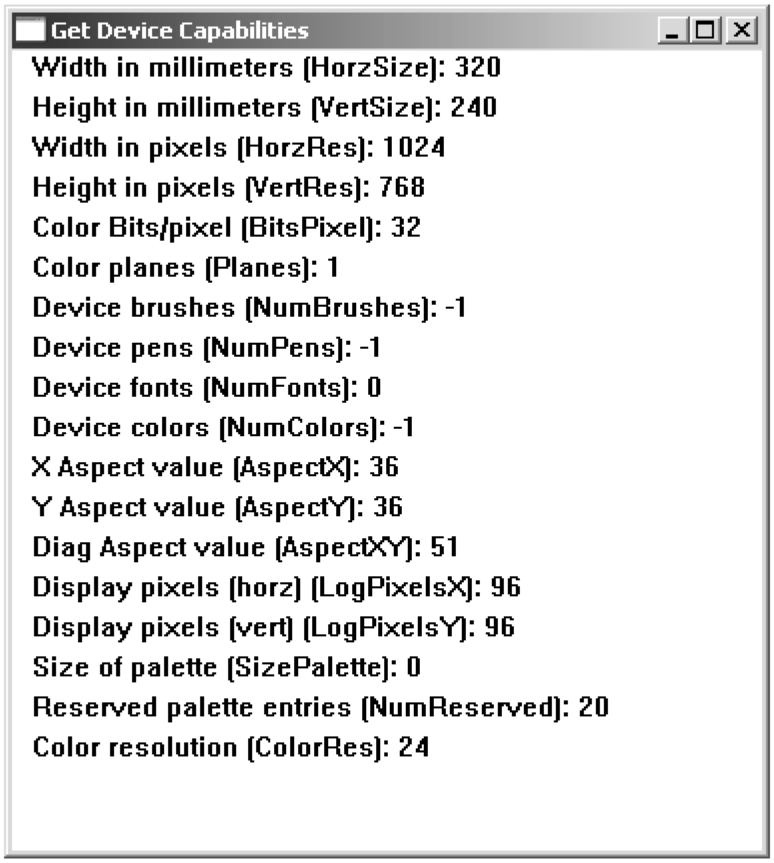
```
type
    MsgProc_t:  procedure( hwnd:dword; wParam:dword; lParam:dword );

    MsgProcPtr_t:
        record

            MessageValue:   dword;
            MessageHndlr:   MsgProc_t;

        endrecord;



// The dispatch table:
//
//  This table is where you add new messages and message handlers
//  to the program.  Each entry in the table must be a tMsgProcPtr
//  record containing two entries: the message value (a constant,
//  typically one of the wm.***** constants found in windows.hhf)
//  and a pointer to a "tMsgProc" procedure that will handle the
//  message.

readonly

    Dispatch:   MsgProcPtr_t; @nostorage;

        MsgProcPtr_t
            MsgProcPtr_t:[  w.WM_DESTROY, &QuitApplication   ],
            MsgProcPtr_t:[  w.WM_CREATE,  &Create            ],
            MsgProcPtr_t:[  w.WM_PAINT,   &Paint             ],

            // Insert new message handler records here.

            MsgProcPtr_t:[  0, NULL ];   // This marks the end of the list.




// Create-
//
//  This function reads an application-specific set of device capabilities
// from Windows using the w.GetDevCaps API function.  This function stores
// the device capabilities into the global appDevCaps record.

procedure Create;
var
    hdc :dword;

begin Create;

    GetDC( hwnd, hdc );

        // Generate a sequence of calls to GetDeviceCaps that
        // take the form:
        //
        //  GetDeviceCaps( w.FIELDNAMEINUPPERCASE );
        //  mov( eax, appDevCaps.FieldName );
        //
        // Where the field names come from the deviceCapabilities_t type.

        #for( field in dcTemplate_c )

            GetDeviceCaps( @text( "w." + @uppercase( field.fieldName, 0 )) );
            mov( eax, @text( "appDevCaps.fields." + field.fieldName ));

        #endfor

    ReleaseDC;

end Create;




// QuitApplication:
//
//  This procedure handles the "wm.Destroy" message.
//  It tells the application to terminate.  This code sends
//  the appropriate message to the main program's message loop
//  that will cause the application to terminate.


procedure QuitApplication( hwnd: dword; wParam:dword; lParam:dword );
begin QuitApplication;

    w.PostQuitMessage( 0 );

end QuitApplication;


// Paint:
//
//  This procedure handles the "wm.Paint" message.
//  This procedure displays several lines of text with
//  different colors.

procedure Paint( hwnd: dword; wParam:dword; lParam:dword ); @nodisplay;
var
    hdc:    dword;              // Handle to video display device context
    ps:     w.PAINTSTRUCT;      // Used while painting text.
    rect:   w.RECT;             // Used to invalidate client rectangle.
    outStr: string;

type
    dclbl_t:    record

        theName     :string;
        desc        :string;

    endrecord;


static
    DClabels    :dct_t[  DCfields_c ]  := dcTemplate_c;
```

```
begin Paint;

    push( ebx );
    push( edi );

    // Allocate temporary storage for a string object
    // (automatically goes away when we return):

    tstralloc( 256 );
    mov( eax, outStr );

    // When Windows requests that we draw the window,
    // fill in the string in the center of the screen.
    // Note that all GDI calls (e.g., DrawText) must
    // appear within a BeginPaint..EndPaint pair.

    BeginPaint( hwnd, ps, hdc );

        for( mov( 0, ebx ); ebx < DCfields_c; inc( ebx )) do

            // Sneaky trick: Although the global appDevCaps is really
            // a structure, this code treats it as an array of
            // dwords (because that's what it turns out to be).

            str.put
            (
                outStr,
                DClabels.desc[ ebx*8 ],
                " (",
                DClabels.fieldName[ ebx*8 ],
                "): ",
                appDevCaps.elements[ ebx*4 ]
            );
            intmul( 20, ebx, edx ); // Compute y-coordinate for output.
            TextOut( 10, edx, outStr, str.length( outStr ) );

        endfor;

    EndPaint;

    pop( edi );
    pop( ebx );

end Paint;





// 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( hwnd:dword; uMsg:uns32; wParam:dword; lParam:dword  );
    @stdcall;
begin WndProc;

    // uMsg contains the current message Windows is passing along to
    // us.  Scan through the "Dispatch" table searching for a handler
    // for this message.  If we find one, then call the associated
    // handler procedure.  If we don't have a specific handler for this
    // message, then call the default window procedure handler function.

    mov( uMsg, eax );
    mov( &Dispatch, edx );
    forever

        mov( (type MsgProcPtr_t [edx]).MessageHndlr, ecx );
        if( ecx = 0 ) then

            // If an unhandled message comes 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.

            w.DefWindowProc( hwnd, uMsg, wParam, lParam );
            exit WndProc;


        elseif( eax = (type MsgProcPtr_t [edx]).MessageValue ) then

            // If the current message matches one of the values
            // in the message dispatch table, then call the
            // appropriate routine.  Note that the routine address
            // is still in ECX from the test above.

            push( hwnd );   // (type tMsgProc ecx)(hwnd, wParam, lParam)
            push( wParam ); //  This calls the associated routine after
            push( lParam ); //  pushing the necessary parameters.
            call( ecx );

            sub( eax, eax ); // Return value for function is zero.
            break;

        endif;
        add( @size( MsgProcPtr_t ), edx );

    endfor;

end WndProc;



// Here's the main program for the application.

begin GDCaps;
```

```
// Set up the window class (wc) object:

mov( @size( w.WNDCLASSEX ), wc.cbSize );
mov( w.CS_HREDRAW | w.CS_VREDRAW, wc.style );
mov( &WndProc, wc.lpfnWndProc );
mov( NULL, wc.cbClsExtra );
mov( NULL, wc.cbWndExtra );
mov( w.COLOR_WINDOW+1, wc.hbrBackground );
mov( NULL, wc.lpszMenuName );
mov( ClassName, wc.lpszClassName );

// Get this process' handle:

w.GetModuleHandle( NULL );
mov( eax, hInstance );
mov( eax, wc.hInstance );

// Get the icons and cursor for this application:

w.LoadIcon( NULL, val w.IDI_APPLICATION );
mov( eax, wc.hIcon );
mov( eax, wc.hIconSm );

w.LoadCursor( NULL, val w.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.

w.RegisterClassEx( wc );

w.CreateWindowEx
(
    NULL,
    ClassName,
    AppCaption,
    w.WS_OVERLAPPEDWINDOW,
    w.CW_USEDEFAULT,
    w.CW_USEDEFAULT,
    w.CW_USEDEFAULT,
    w.CW_USEDEFAULT,
    NULL,
    NULL,
    hInstance,
    NULL
);
mov( eax, hwnd );

w.ShowWindow( hwnd, w.SW_SHOWNORMAL );
w.UpdateWindow( hwnd );

// Here's the event loop that processes messages
// sent to our window.  On return from GetMessage,
// break if EAX contains false and then quit the
// program.
```

```
    forever

        w.GetMessage( msg, NULL, 0, 0 );
        breakif( !eax );
        w.TranslateMessage( msg );
        w.DispatchMessage( msg );

    endfor;

    // The message handling inside Windows has stored
    // the program's return code in the wParam field
    // of the message.  Extract this and return it
    // as the program's return code.

    mov( msg.wParam, eax );
    w.ExitProcess( eax );

end GDCaps;
```

Figure 6-6: GDCaps Output



```
Width in millimeters (HorzSize): 320
Height in millimeters (VertSize): 240
Width in pixels (HorzRes): 1024
Height in pixels (VertRes): 768
Color Bits/pixel (BitsPixel): 32
Color planes (Planes): 1
Device brushes (NumBrushes): -1
Device pens (NumPens): -1
Device fonts (NumFonts): 0
Device colors (NumColors): -1
X Aspect value (AspectX): 36
Y Aspect value (AspectY): 36
Diag Aspect value (AspectXY): 51
Display pixels (horz) (LogPixelsX): 96
Display pixels (vert) (LogPixelsY): 96
Size of palette (SizePalette): 0
Reserved palette entries (NumReserved): 20
Color resolution (ColorRes): 24
```

This program introduces a new Windows message and some novel HLA concepts, so it s worthwhile to spend a small amount of time discussing certain parts of this source code.

Device capability information tends to be static. That is, the capabilities of a device are usually fixed by the manufacturer of that device and these capabilities generally don t change while your application is running. For example, it is rare for the display resolution to change during the execution of your program (possible, but rare; the user, for example, could run the display settings control panel while your application is executing and resize the display). For many applications, it s probably safe to assume that the device capabilities remain static as long as the program doesn t crash or lose data if this assumption turns out to be false (i.e., if the worst effect is that the application s display looks a little funny, you can probably get away with obtaining this information just once rather than querying Windows for the values every time you want to use them). The *GDcaps.hla* program uses this approach - it does not read the device capabilities every time it receives a `w.WM_DRAW` message. Instead, it reads the information into an array that is local to the program and then displays the data from this array every time it repaints the screen. This spares the application from having to make a couple dozen calls to the Win32 `w.GetDeviceCaps` function every time a paint message comes along.

One problem when reading the data only once is when do you read this data? You might be tempted to make the `w.GetDeviceCaps` call from the application s main program, before it starts processing messages. This approach, however, won t work. The `w.GetDeviceCaps` API function requires a device context value, which is only available after you ve created a window. One way to handle this situation is to create a static boolean flag that you ve initialized with false and then initialize the in-memory data structure on the first call to the `Paint` procedure; by testing this flag (and setting it to true once you ve initialized the capability data structure), you can ensure that you only make the calls to `w.GetDeviceCap` once - on the very first call to the `Paint` procedure. There are a couple of problems with this approach. First, it clutters up your Paint procedure with code that really has nothing to do with painting the window. Second, although testing a boolean flag isn t particularly inefficient, it is somewhat unaesthetic (especially in an assembly language program) to have to repeat this test over and over again on each call to the `Paint` procedure even though the initialization of the code takes place exactly once. Yet another problem with this static variable approach is that there are actually some times when you ll want to reinitialize the device capabilities data structure; for example, if the application destroys the window and then creates a new one, you get a new device context and it s possible that the capabilities have changed. To solve these problems (and more), the GDcaps.hla application taps into a different message, `w.WM_CREATE`, in order to initialize the in-memory data structure. Windows will send a `w.WM_CREATE` message to an application s window procedure immediately after it creates a new window. By placing the code to initialize the capabilities data structure in a procedure that responds to this message, you can ensure that you initialize the device capabilities array whenever Windows creates a new window for your application.

The `Create` procedure in the *GDCaps* application uses some novel code that is worth exploring. This code makes multiple calls to the `w.GetDeviceCaps` API function with index values for each of the capabilities this particular program wants to display. After each call to `w.GetDeviceCaps`, the `Create` procedure stores the result result in one of the fields of the `appDevCaps` data structure. The `appDevCaps` variable s type is `deviceCapabilities_t` which is a union of a record and an array type. The record component lets you access each of the device capabilities by (field) name, the array field lets you access all of the fields using a numeric index. Typically, the initialization code will access this object as an array and the rest of the application will access the fields of the record. The record structure is type `deviceCapRecord_t` and it has the following declaration:

```
type
   deviceCapRecord_t :record
      HorzSize       :int32;
      VertSize       :int32;
      HorzRes        :int32;
      BitsPixel      :int32;
      Planes         :int32;
      NumBrushes     :int32;
      NumPens        :int32;
      NumFonts       :int32;
      NumColors      :int32;
      AspectX        :int32;
      AspectY        :int32;
      AspectXY       :int32;
      LogPixelsX     :int32;
      LogPixelsY     :int32;
      SizePalette    :int32;
      NumReserved    :int32;
      ColorRes       :int32;
   endrecord;
```

However, if you look at the source code, you ll discover that the actual record type declaration takes the following form:

```
type
    deviceCapRecord_t :record

        #for( i in dcTemplate_c )

            @text( i.fieldName ) :int32;

        #endfor

    endrecord;
```

This strange looking code constructs all the fields for this record by iterating through the dcTemplate_c constant and extracting the field names from that array of records. The `#for` compile-time loop repeats once for each element of the `dcTemplate_c` (constant) array. Each element of this array is a record whose `fieldName` field contains the name of one of the fields in the `deviceCapRecord_t` type. On each iteration of the `#for` loop, HLA sets the loop control variable (`i`) to the value of each successive array element. Therefore, `i.fieldName` is a character string constant containing the field name for a given iteration of the loop. The @text(i.fieldName) construct tells HLA to substitute the text of the string in place of that string constant, hence HLA replaces each instance of the `@text( i.fieldName )` compile-time function with a field name. That s how this funny looking code expands into the former record declaration.

Now you re probably wondering why this program uses such obtuse code; why not simply type all the field names directly into the record so the code is a little clearer? Well, this was done in order to make the program easier to maintain (believe it or not). As it turns out, the *GDCaps* application references these fields throughout the source code. This means that if you want to modify the code to add or remove fields from this record, you d actually have to make several changes at various points throughout the source code to correctly make the change. Code is much easier to maintain if you only have to make a change (like adding or removing values to print) in one spot in the source code. The *GDCaps* program takes advantage of some of HLA s more sophisticated features to allow the program to collect all the field information into a single data structure: the `dcTemplate_c` array constant. Here s the declaration for this constant and its underlying type:

```
type
    dct_t   :record
        fieldName   :string;
        desc        :string;
    endrecord;

const
    dcTemplate_c :dct_t[] :=
    [
        dct_t:[ "HorzSize",     "Width in millimeters" ],
        dct_t:[  "VertSize",      "Height in millimeters" ],
        dct_t:[  "HorzRes",       "Width in pixels" ],
        dct_t:[  "VertRes",       "Height in pixels" ],
        dct_t:[  "BitsPixel",     "Color Bits/pixel" ],
        dct_t:[  "Planes",        "Color planes" ],
        dct_t:[  "NumBrushes",    "Device brushes" ],
        dct_t:[  "NumPens",       "Device pens" ],
        dct_t:[  "NumFonts",      "Device fonts" ],
        dct_t:[  "NumColors",     "Device colors" ],
        dct_t:[  "AspectX",       "X Aspect value" ],
        dct_t:[  "AspectY",       "Y Aspect value" ],
        dct_t:[  "AspectXY",      "Diag Aspect value" ],
        dct_t:[  "LogPixelsX",    "Display pixels (horz)" ],
        dct_t:[  "LogPixelsY",    "Display pixels (vert)" ],
        dct_t:[  "SizePalette",   "Size of palette" ],
        dct_t:[  "NumReserved",   "Reserved palette entries" ],
        dct_t:[  "ColorRes",      "Color resolution" ]
    ];
```

If you don t want *GDCaps* to display a particular item above, simply remove the entry from this array. If you want *GDCaps* to display a device capability that is not present in this list, simply add the field name and description to the `dcTemplate_c` array. That s all there is to it. The program recomputes all other changes automatically when you recompile the code.

The `dcTemplate_c` array is an array of records with each element containing two fields: a field name/`GetDevCaps` index constant and a description of the field/index constant. The field name you supply must be a string containing the name of the `GetDevCaps` index constant with two exceptions: you don t have to use all upper case characters (HLA will automatically convert the code to upper case later, so you can use more readable names like HorzSize rather than HORZSIZE) and you don t need the w. prefix (HLA will also supply this later). Note that HLA will use whatever field names you supply in the `fieldNames` field as the field names for the record it constructs using the `#for` loop that you saw earlier. To access these values in the application, you d normally use identifiers like `appDevCaps.fields.HorzSize` and `appDevCaps.fields.NumColors`. The second field of each record is a short description of the value that `GetDevCaps` will return. The *GDCaps* program displays this string (along with the field name) in the output window.

Because we can only call `w.BeginPaint` and `w.EndPaint` within a `w.WM_PAINT` message handler, the `Create` procedure (that handles `w.WM_CREATE` messages) will have to call `w.GetDC/w.ReleaseDC` or `w.GetWindowDC/w.ReleaseDC`. Of course, by including the *wpa.hhf* header file, the *GDcaps* application gets to use the `GetDC`, `ReleaseDC`, and `GetDeviceCap` macros to simplify calling these API functions.

Normally, the code to read all these values from Windows and store them into the `appDevCaps` data structure would look something like this:

```
    GetDC( hwnd, hdc );

        GetDeviceCaps( w.HORZSIZE );
        mov( eax, appDevCaps.fields.HorzSize );

        GetDeviceCaps( w.VERTSIZE );
        mov( eax, appDevCaps.fields.VertSize );
            .
            .
            .
        GetDeviceCaps( w.COLORRES );
        mov( eax, appDevCaps.fields.ColorRes );
```

```
   ReleaseDC;
```

One problem with the straight-line code approach is that it s a bit difficult to maintain. If you add or remove fields in the `dcTemplate_c` array, you ll have to manually modify the `Create` procedure to handle those modifications. To avoid this, the *GDcaps* application uses HLA s compile-time `#for` loop to automatically generate all the code necessary to initialize the `appDevCaps` structure. Here s the code that achieves this:

```
        #for( field in dcTemplate_c )

            GetDeviceCaps( @text( "w." + @uppercase( field.fieldName, 0 )) );
            mov( eax, @text( "appDevCaps.fields." + field.fieldName ));

        #endfor
```

Because this code is somewhat sophisticated, it s worthwhile to dissect this code in order to fully understand what s going on.

The first thing to note is that `#for` is a *compile-time* loop, not a run-time loop. This means that the HLA compiler repeats the emission of the text between the `#for` and the corresponding `#endfor` the number of times specified by the `#for` expression. This particular loop repeats once for each field in the `dcTemplate_t` constant array. Therefore, this code emits n copies of the `GetDeviceCaps` call and the `mov` instruction, where n is the number of elements in the `dcTemplate_t` array.

The `#for` loop repeats once for each element of this array, and on each iteration it assigns the current element to the `field` compile-time variable. Therefore, on the first iteration of this loop, the `field` variable will contain the record constant `dct_t:[“HorzSize”, "Width in millimeters" ]`, on the second iteration it will contain `dct_t:[“VertSize”, "Height in millimeters" ]`, on the third iteration it will contain `dct_t:[“HorzRes”, "Width in pixels" ]`, etc. It is important to realize that this array of records and this `field` variable do not exist in your program s object code. They are internal variables that the compiler maintains only while compiling your program. Unless you explicitly tell it to do so, HLA will not place these strings in your object code. Within the body of the loop, the field compile-time variable behaves like an HLA `const` or `val` object.

Within the body of the `#for` loop, the code constructs Windows compatible `w.GetDeviceCaps` index names and `appDevCaps` field names using compile-time string manipulation. The compile-time function @uppercase(field.fieldName, 0) returns a string that is a copy of `field.fieldName`, except that it converts all lowercase alphabetic characters in the string to upper case. For example, on the first iteration of the `#for` loop (when `field` contains HorzSize), this function call returns the string HORZSIZE . The `“w.” + @uppercase( field.fieldName, 0 )` expression concatenates the uppercase version of the field name to a w. string, producing Windows compatible constant names like w.HORZSIZE and w.COLORRES . The `@text` function translates this string data into text data so that HLA will treat the string as part of your source file (rather than as a string literal constant). Therefore, the iterations of the `#for` loop produces the equivalent of the following statements:

```
      GetDeviceCaps( w.HORZSIZE );
      GetDeviceCaps( w.VERTSIZE );
      GetDeviceCaps( w.HORZRES );
         .
         .
         .
      GetDeviceCaps( w.COLORRES );
```

This behavior, by the way, is why the field names in the first string you supply for each dcTemplate_c element must match the GetDeviceCaps constants except for alphabetic case. The #for loop uses those field names to generate the index constant names that it feeds to the GetDeviceCaps API function.

The second instruction in the #for loop copies the data returned by GetDeviceCaps (in EAX) into the appropriate field of the appDevCaps.field data structure. The construction of the field name is almost identical to that used for the GetDeviceCaps call except, of course, the code doesn t do an upper case conversion.

The #for..#endfor loop winds up producing code that looks like the following:

```
    GetDeviceCaps( w.HORZSIZE );
    mov( eax, appDevCaps.fields.HorzSize );
    GetDeviceCaps( w.VERTSIZE );
    mov( eax, appDevCaps.fields.VertSize );
    GetDeviceCaps( w.HORZRES );
    mov( eax, appDevCaps.fields.HorzRes );
        .
        .
        .
    GetDeviceCaps( w.COLORRES );
    mov( eax, appDevCaps.fields.ColorRes );
```

The nice thing about using the #for..#endfor loop, rather than explicitly coding these statements, is that all you have to do to change the device capabilities you retrieve is to modify the dcTemplate_c data type. The next time you compile the source file, the #for loop will automatically generate exactly those statements needed to retrieve the new set of fields for this record type. No other modifications to the code are necessary.

Of course, you could achieve this same result by writing a run-time loop. To do that, you d need an array of index values (at run time) whose elements you pass to the GetDeviceCaps function on each iteration of the loop. The elements of that run-time array need to appear in the same order as the fields in the dcTemplate_c structure (note that you could write some HLA compile-time code to automatically build and initialize this array for you). If you re interested in this approach (which generates a little bit less code), feel free to implement it in your own applications. Of course, the number of fields you ll typically grab from GetDevCaps is usually so small that expanding the code in-line is not usually a problem.

The Paint procedure is responsible for actually displaying the device capabilities information. The procedure is relatively straight-forward - it fetches a list of labels and descriptions from one array and the list of capability values from the array/record read in the Create procedure, turns this information into a sequence of strings and displays these strings in the application s window.

The field names and descriptions are really nothing more than a run-time instantiation of the dcTemplate_c array. The Paint procedure creates this run-time array using the following declaration:

```
static
    DClabels    :dct_t[ DCfields_c ] := dcTemplate_c;
```

This generates a run-time array, DClabels, that contains the same information as the compile-time array constant dcTemplate_c. Remember that at run-time, HLA string objects are four-byte pointers. Therefore, DClabels is really an array of two four-byte pointer values (with one pointer containing the address of a string holding the field name and the second pointer containing the address of a string with the field s description). HLA stores the actual character data in a different section of memory, not directly within the DClabels array.

To print the data to the window, the Paint procedure converts the numeric information in the corresponding appDevCaps array element to a string and then concatenates the two strings in the DClabels array with this numeric string and calls `w.TextOut` to display the whole string. This sounds like a bit of work, but the HLA Standard Library comes to the rescue with the `str.put` function that lets you do all this work with a single function call:

```
str.put
(
    outStr,                         // Store the result here
    DClabels.desc[ ebx*8 ],         // Starts with description string
    " (",                           // Concatenates "(" followed by
    DClabels.fieldName[ ebx*8 ],    // the field name string and ")"
    "): ",                          // Then it concatenates the string
    appDevCaps.elements[ ebx*4 ]    // conversion of the GetDevCaps value.
);
```

Because each element of the `DClabels` array is a pair of 32-bit pointers, this code uses the *8 scaled indexed addressing mode to access elements of that array. The `appDevCaps` array is just an array of `int32` values, so this code uses the *4 scaled indexed addressing mode to access those array elements.

To actually output the text data to the Window, the Paint procedure uses the following two statements:

```
intmul( 20, ebx, edx ); // Compute y-coordinate for output.
TextOut( 10, edx, outStr, str.length( outStr ) );
```

The `intmul` instruction computes the separation between lines (20 pixels) and the `TextOut` call displays the string that the `str.put` call created. A run-time for loop around these statements repeats these statements for each of the fields in the `DClabels` and `appDevCaps` arrays.

Carefully writing this code to make it maintainable may seem like an exorbitant effort for what amounts to a demonstration program. However, this was done on purpose. The `Create` procedure and the related data structures are perfectly general and you can cut and paste this code into other applications. Because most applications won t need exactly the set of `GetDevCaps` values that *GDCaps* uses, spending a little bit of extra effort to make this code easy to modify will help you if you decide to lift portions of this code and place them in a different application.

6.4: Typefaces and Fonts

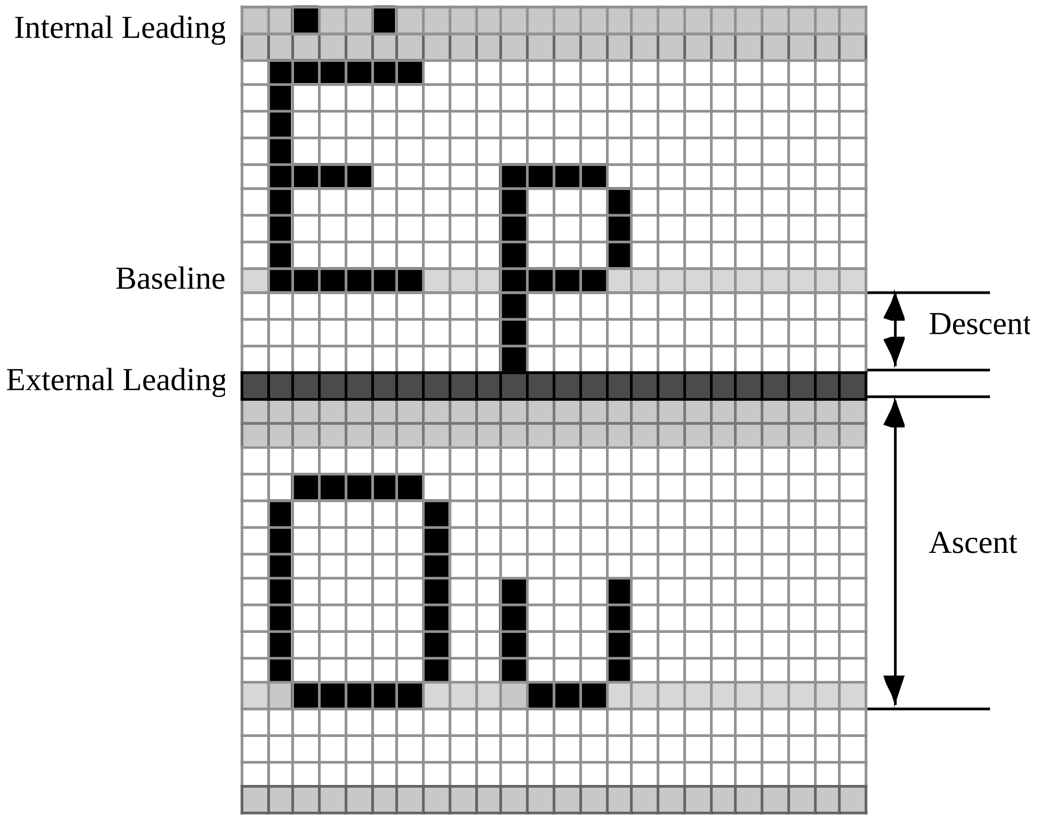
The first major difference between a console application and a GUI application is that GUI apps allow multiple fonts. In this section, we ll explore how to use multiple fonts in a GUI application.

Perhaps the first place to start this discussion is with the definition of a *font*. This side trip is necessary because many people confuse a font with a *typeface*. A *font* is a collection of attributes associated with printed or displayed text. The attributes that define a font include the typeface, the size, and other style attributes such as italic, bold, underlined, and so on.

A *typeface* is a family of type styles that specify the basic shape of the characters in a particular typeface. Examples of typefaces include *Times Roman, Helvetica, Arial,* and *Courier*. Note that *Times Roman* (contrary to popular belief) is not a font; there are many different fonts that use the *Times Roman* typeface (indeed, in theory there are an infinite number of fonts that employ the *Times Roman* typeface, because there are, in theory, an infinite number of sizes one could use with this typeface).

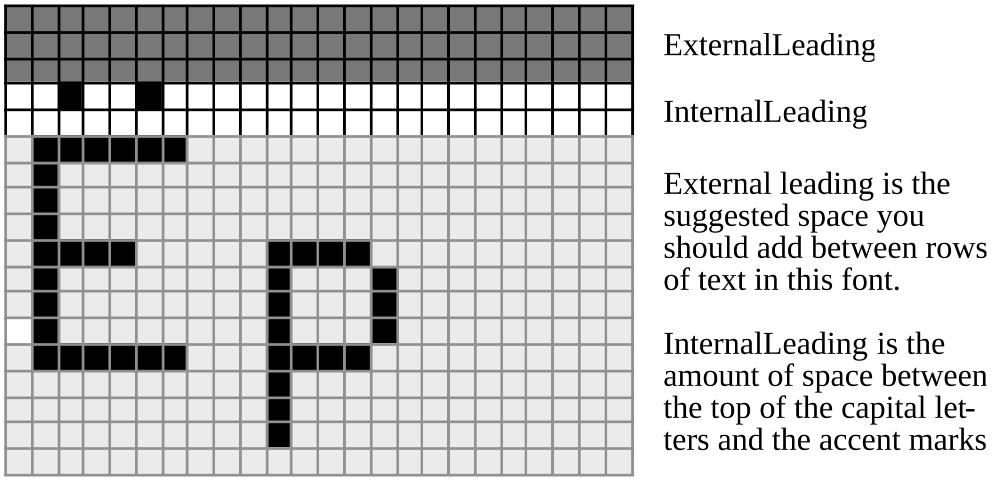
Another attribute of a font is the size of that font. Font sizes are usually specified in *points*. A point is approximately $^1/_{72}$ inches. Therefore, a font with a 72 point size consists of characters that are approximately one inch tall. An important thing to realize is that 72 point characters are not all approximately one inch tall. The size of a font specifies a set of boundaries between which typical characters in that font actually fall. A 72-point lower-case a , for example, is smaller than a 72-point uppercase A even though they both have the same size . To understand how graphic artists measure font sizes, consider Figure 6-7. This figure specifies four terms that graphic artists typically use when discussing the size of a font: leading, baseline, descent, and ascent.

Figure 6-7: Font Sizes



Leading (pronounced led-ing as in the metal lead, not lead-ing as in leading the pack) is the amount of spacing that separates two lines of text. This name comes from the days when graphic artists and printers used lead type in the printing process. Leading referred to thin lead strips inserted between rows of text to spread that text out. Windows actually defines two types of leading: internal leading and external leading (see Figure 6-8). External leading is the amount of space between the upper-most visible portion of any character in the font and the bottom-most visible portion of any character in the font on the previous line of text. Internal leading is the amount of space between the top-most component of any standard character in the font and the top of any accents appearing on characters in the font. Note that the external leading does not count towards the size of a font.

Figure 6-8: Internal Versus External Leading


ExternalLeading
InternalLeading
External leading is the suggested space you should add between rows of text in this font.
InternalLeading is the amount of space between the top of the capital letters and the accent marks

The actual size of a font is measured as the distance from the bottom of the descent region to the top of the ascent region. The position of these two components appears in Figure 6-9 and Figure 6-10. Figure shows how the region that defines the font s size.

Figure 6-9: The Ascent of a Font

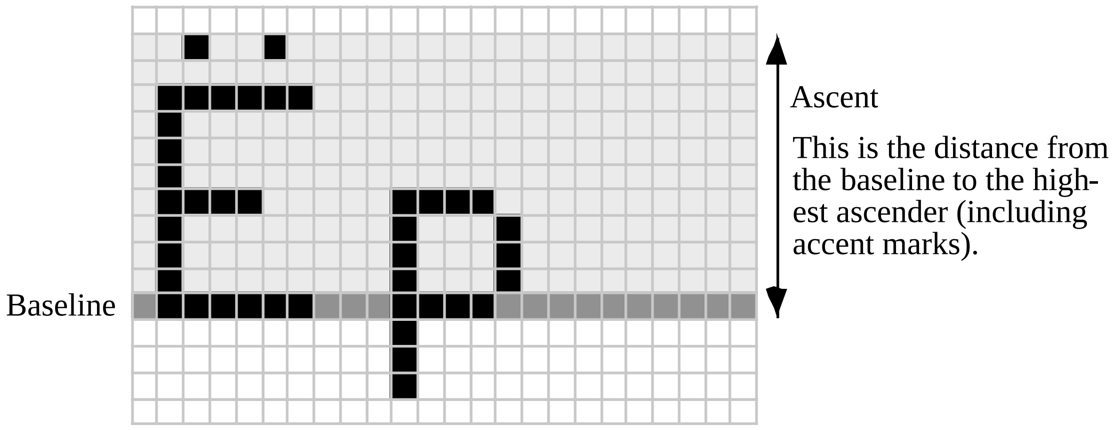

Ascent
This is the distance from the baseline to the highest ascender (including accent marks).
Baseline

Figure 6-10: The Descent of a Font

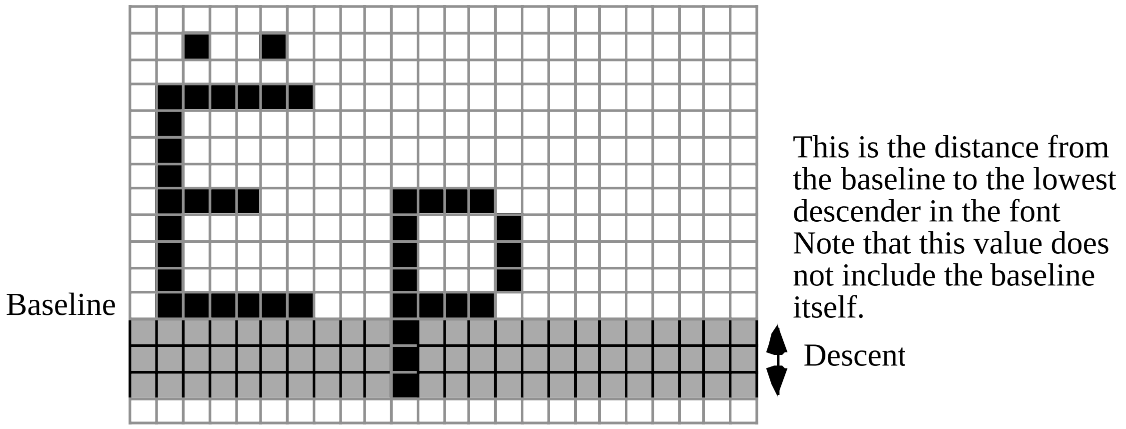
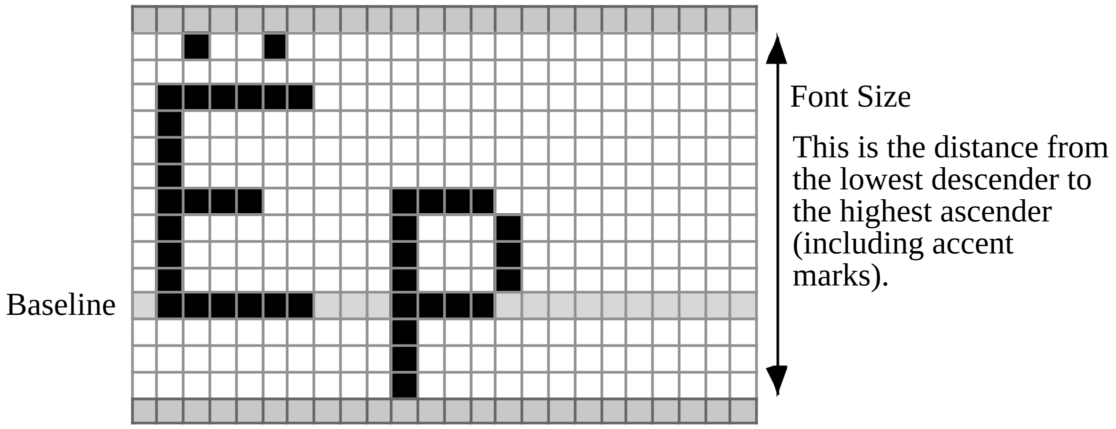

This is the distance from the baseline to the lowest descender in the font Note that this value does not include the baseline itself.
Baseline
Descent

Figure 6-11: The Size of a Font


Font Size
This is the distance from the lowest descender to the highest ascender (including accent marks).
Baseline

There are two important things to note from these figures. First, as already stated, the size of a font does not specify the size of any individual character within the font. Instead, it (generally) provides a maximum range between the bottom-most portions of certain characters in the font and the top-most portions of certain characters within the font. It is very unusual for a single character s height to span this full range. Many characters (e.g., various lower-case characters) are just a little larger than half of the font s specified size. The second thing to note is that the size of a font says absolutely nothing about the width of the characters within that font. True, as the font size increases, the width of the characters tends to increase as well, but the actual width is generally dependent upon the typeface or other font attributes (e.g., certain fonts have a *condensed* attribute that tells a graphic artist that the individual characters are narrower than a standard font of the same size).

The exact width of characters within a font is typically a function of the typeface. However, we can classify fonts into two categories based upon the widths of the characters within the font: proportional fonts and non-proportional (or monospaced) fonts. In a proportional font, the widths of the characters vary according to the particular character and the whim of the font designer. Certain characters, like W require more horizontal space than other characters (e.g., i) in order to look proper within a line of text. As a result, you cannot determine the amount of horizontal space a string of characters will consume by simply counting the number of characters in the string. Instead, you ll have to sum up the widths of each of the individual characters in the string in order to determine the width of the whole string. This makes it difficult to predict the number of characters you can display within a given horizontal distance without actually having the characters to work with.

Some typefaces (e.g., *Courier*) are non-proportional, or monospaced. This means that all the characters in the typeface have the same width. The font designer achieves this by condensing wide characters (like W) and elongating narrow characters (like i). Computing the width of a string of characters rendered with monospaced fonts is easily achieved by counting the number of characters in the string and then multiply this length by the width of a single character.

Windows actually supports three distinct types of fonts: raster fonts, scalable (plotter) fonts, and True Type fonts. Each of these fonts have their own advantages and disadvantages that we ll explore in the following paragraphs. For the most part, this book will assume the use of True Type fonts, but much of the information this chapter discusses is independent of the font technology, so you may use whatever font system you like in your applications.

Raster fonts are bit-mapped images that have two primary attributes: Windows can quickly display raster fonts and raster fonts can be hand-tuned to look very good on a given display device. The disadvantage of the raster font technology is that the output only looks good if the system provides a font in the size that you request. Although Windows can *scale* a given raster font to almost any size, the result is rarely pretty. If your application needs to use a raster font, be sure to use a font size provided by the system to keep the result as legible as possible. Another problem with raster fonts is that you generally need different sets of fonts for different output devices. For example, you ll need one set of raster fonts for display output and one set for printer output (because of the inherent differences in display resolution, you ll need different bitmaps for the two devices).

Scalable fonts are an older font technology that uses line vectors to draw the outlines of the characters in the font. This font technology mainly exists to support certain output devices like pen plotters. Because such output devices are rare in modern computer systems (most plotting that occurs these days takes place on an ink-jet printer rather than a pen plotter), you ll probably not see many scalable fonts in the system. The advantage of scalable fonts over raster plots is that they define their individual characters mathematically via a sequence of points. The pen plotter simply draws lines between the points (in a connect-the-dots fashion). Mathematically, it is very easy to scale such fonts to a larger or smaller size by simply multiplying the coordinates of these points by a fixed value (using a vector multiplication). Therefore, it is easy to produce characters of any reasonable size without a huge quality loss as you get when attempting to scale raster fonts. Because you ll rarely see scalable fonts in use in modern Windows systems, we ll ignore this font technology in this book.

Although scalable fonts solve the problem of adjusting the size of the fonts, the scalable fonts that came with the original versions of Windows were, shall we say, a little less than aesthetic. Fortunately, Apple and Microsoft worked together to develop a high-quality scalable font technology known as *True Type*. Like the Postscript font technology that preceded it, the True Type technology uses a mathematical definition of the outlines for each character in the font. True Type technology uses a simpler, *quadratic*, mathematical definition (versus Postscript s *bezier* outlines) that is more efficient to compute; therefore, Windows can render True Type fonts faster than Postscript fonts. The end result is that Windows can efficiently scale a True Type font to any practical size and produce good-looking results. A raster font, displayed at its native size, can look better than a True Type font scaled to that same size (because raster fonts can be hand-optimized for a given size to look as good as possible); but in general, True Type fonts look better than raster fonts because they look good regardless of the size.

Another advantage of True Type fonts is that they can take advantage of technologies like *anti-aliasing* to improve their legibility on a video display. Anti-aliasing is the process of using various shades of the font s primary color to soften the edges around a particular character to eliminate the appearance of jagged edges between pixels in the character. In theory, it s possible to define anti-aliased raster fonts, but you rarely see such fonts in practice.

Yet another advantage of True Type fonts is the fact that you only have to keep one font definition around in the system. Unlike raster fonts, where you have to store each font (i.e., size) as a separate file, True Type fonts only require a single font outline and the system builds the explicit instances of each font (size) you request from that single mathematical definition. To avoid having to construct each character from its mathematical definition when drawing characters, Windows will convert each character in a given font to its bitmapped representation when you first use each character, and then it will cache those bitmaps (i.e., raster images) away for future reference. As long as you leave that font selected into the device context, Windows uses the bitmapped image it has cached away to rapidly display the characters in the font. Therefore, you only pay a rasterizing penalty on the first use of a given character (or on the first use after a character was removed from the font cache because the cache was full). Generally, if you follow the (graphic arts) rule of having no more than four fonts on a given page, Windows should be able to cache up all the characters you are using without any problems.

Regardless of what font technology you use, whenever you want to display characters in some font on the display, you have to tell Windows to create a font for you from a system font object. If the system font object is a raster font and you re selecting that font s native size, Windows doesn t have to do much with the font data (other than possibly load the font data from disk). If you re using a True Type font, or selecting a non-native raster font size, then Windows will need to create a bitmap of the font it can write to the display prior to displaying any characters from that font. This is the process of font creation and you accomplish it using the Windows API function `w.CreateFontIndirect`. This function has the following prototype:

```
static
        CreateFontIndirect: procedure
        (
                var     lplf:LOGFONT
        );
                @stdcall;
                @returns( "eax" );
                @external( "__imp__CreateFontIndirectA@4" );
```

This function call returns a handle to a font. You must save this handle so you can destroy the font when you are done using it.

The `lplf` parameter in the `w.CreateFontIndirect` call is a pointer to a logical font structure (`w.LOGFONT`). This structure takes the following form:

```
type
   LOGFONT  :record
      lfHeight          :int32;
      lfWidth           :int32;
      lfEscapement      :int32;
      lfOrientation     :int32;
      lfWeight          :int32;
      lfItalic          :byte;
      lfUnderline       :byte;
      lfStrikeOut       :byte;
      lfCharSet         :byte;
      lfOutPrecision    :byte;
      lfClipPrecision   :byte;
      lfQuality         :byte;
      lfPitchAndFamily  :byte;
      lfFaceName        :char[ LF_FACESIZE] ;
   endrecord;
```

The `lfHeight` field is the height of the font using device units . If you pass a zero in this field, then Windows returns the default size for the font. Most of the time, you ll want to specify the exact font size to use, so you ll need to convert sizes in a common measurement system (e.g., points) into device units. To compute the `lfHeight` value in points, you d use the following formula:

lfHeight = -(pointSize * appDevCaps.LogPixelsY) / 72

(This assumes, of course, that you ve read the value of the `w.LOGPIXELSY` device capability into `appDevCaps.LogPixelsY` as was done in the previous section.) For example, if you want to compute the `lfHeight` value for a 12-point type, you d use code like the following:

```
mov( pointSize, eax );
imul( appDevCaps.LogPixelsY, eax ); // pointSize * appDevCaps.LogPixelsY
idiv( 72, edx:eax );                //   / 72
neg( eax );                         // Negate the result.
mov( eax, lfVar.lfHeight );         // Store away in the lfHeight field.
```

The `appDevCaps.LogPixelsY` value specifies the number of logical pixels per inch for the given device context. Multiplying this by the desired point size and dividing by 72 points/inch produces the necessary font size in device units.

The `w.LOGFONT lfWidth` field specifies the *average* character width. If this field contains zero, then Windows will compute the best average width for the font s characters based on the `lfHeight` value. Generally, you ll want to supply a zero for this value. However you d prefer a condensed or elongated font, you can supply a non-zero value here. The computation you d use for this value is the same as for `lfHeight`.

The `lfEscapement` and `lfOrientation` values specify an angle from the baseline (in tenths of degrees) or x-axis that Windows will use to rotate the characters when drawing them. Note that this does not tell Windows to draw your lines of text at the specified angle. Instead, it tells Windows the angle to use when drawing each character on the display (rotating the text within each character cell position).

The `lfWeight` field specifies the boldness of the font using a value between zero and 1000. Here is a list of constants you may use to specify the weight of a character:

¥ w.FW_DONTCARE (0)

¥ w.FW_THIN (100)

¥ w.FW_EXTRALIGHT (200)

¥ w.FW_ULTRALIGHT (200)

¥ w.FW_LIGHT (300)

¥ w.FW_NORMAL (400)

¥ w.FW_MEDIUM (500)

¥ w.FW_SEMIBOLD (600)

¥ w.FW_DEMIBOLD (600)

¥ w.FW_BOLD (700)

¥ w.FW_EXTRABOLD (800)

¥ w.FW_ULTRABOLD (800)

¥ w.FW_HEAVY (900)

¥ w.FW_BLACK (900)

You d normally select a bold font rather than using this field to specify the weight (assuming a bold font is available). If a bold font is not available, then this field provides an acceptable alternative to using a bold font (note that bold fonts are not simply fatter versions of each character in the typeface; the strokes are actually different for bold versus regular characters, increasing the weight of a character is only an approximation of a bold font).

The `lfUnderline, lfStrikeout`, and `lfItalic` fields are boolean variables (true/false or 1/0) that specify whether the font will have underlined characters, strikeout characters (a line through the middle of each character), or italic characters. Generally, you would not use the `lfItalic` flag - you d choose an actual italic font instead. However, if an italic font is not available for a given typeface, you can approximate an italic font by setting the `lfItalic` flag.

The `lfCharSet` field specifies the character set to use. For the purposes of this text, you should always initialize this field with `w.ANSI_CHARSET` or `w.OEM_CHARSET`. For details on internationalization and other character set values that are appropriate for this field, please consult Microsoft s documentation.

The `lfOutPrecision` field tells Windows how closely it must match the requested font if the actual font you specify is not in the system or if there are two or more fonts that have the same name you ve requested. Figure 6-2 lists some common values you d supply for this field.

Table 6-2: lfOutPrecision Values (in w.LOGFONT Records)

Value	Description
w.OUT_DEFAULT_PRECIS	Specifies the default font mapper behavior
w.OUTLINE_PRECIS	This tells Windows to choose from True Type and other outline-based fonts (Win NT, 2K, and later only).
w.OUT_RASTER_PRECIS	If the font subsystem can choose between multiple fonts, this value tells Windows to choose a raster-based font.
w.OUT_TT_ONLY_PRECIS	This value tells the font subsystem to use only True Type fonts. If there are no True Type fonts, then the system uses the default behavior. If there is at least one True Type font, and the system does not contain the requested font, then the system will substitute a True Type font for the missing font (even if they are completely different).
w.OUT_TT_PRECIS	Tells the font subsystem to choose a True Type font over some other technology if there are multiple fonts with the same name.

The `lfClipPrecision` field defines how to clip characters that are partially outside the *clipping region*. A clipping region is that area of the display outside the area which an application is allowed to draw. You should initialize this field with the `w.CLIP_DEFAULT_PRECIS` value.

The `lfQuality` field specifies the output quality of the characters. Windows supports three constant values for this field: `w.DEFAULT_QUALITY, w.DRAFT_QUALITY`, and `w.PROOF_QUALITY`. For display output, you should probably choose `w.DEFAULT_QUALITY`.

The `lfPitchAndFamily` field specifies certain attributes of the font. The low-order two bits should contain one of `w.DEFAULT_PITCH, w.FIXED_PITCH`, or `w.VARIABLE_PITCH`. This specifies whether Windows will use proportional or monospaced fonts (or whether the decision is up to Windows, should you choose w.DEFAULT_PITCH). You may logically OR one of these constants with one of the following values that further specifies the font family:

¥ w.FF_DECORATIVE - use a decorative font (like Old English)

¥ w.FF_DONTCARE - use an arbitrary font (Windows chooses)

¥ w.FF_MODERN - generally specified for monospaced fonts

¥ w.FF_ROMAN - generally specified for proportional, serifed, fonts

¥ w.FF_SCRIPT - specifies a font that uses a cursive (handwritten) style

¥ w.FF_SWISS - specifies a proportional, sans-serifed font

The `lfFaceName` field is a zero-terminated character string that specifies the font name. The font name must not exceed 31 characters (plus a zero terminating byte). If this field contains an empty string, then Windows picks a system font based on the other font attributes appearing in the `w.LOGFONT` structure (e.g., the `lfPitchAndFamily` field). If this field is not an empty string, then the font choice takes precedence over the other attributes appearing in the `w.LOGFONT` record (e.g., if you specify times roman as the font, you ll get a variable pitch roman font, regardless of what value you specify in `lfPtichAndFamily`). If the name doesn t exactly match an existing font in the system, Windows may substitute some other font.

Once you load up a `w.LOGFONT` object with an appropriate set of field values, you can call `w.CreateFontIndirect` to have Windows construct a font that matches the characteristics you ve supplied as closely as possible. An important fact to realize is that Windows may not give you exactly the font you ve requested. For example, if you ask for a Dom Casual but the system doesn t have this typeface available, Windows will substitute some other font. If you ve requested that Windows create a font that is a bit-mapped font, Windows may substitute a font in the same typeface family but of a different size. So never assume that Windows has given you exactly the font you ve asked for. As you ll see in a little bit, you can query Windows to find out the characteristics of the font that Windows is actually using.

Once you create a font with `w.CreateFontIndirect`, Windows does not automatically start using that font. Instead, the `w.CreateFontIndirect` function returns a handle to the font that you can save and rapidly select into the device context as needed. This allows you to create several fonts early on and then rapidly switch between them by simply supplying their handles to the `w.SelectObject` function. The `w.SelectObject` API function lets you attach some GDI object to a device context. The prototype for this function is the following:

```
static
    SelectObject: procedure
    (
        hdc                 :dword;
        hgdiobj             :dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__SelectObject@8" );
```

The `hdc` parameter is a handle to the device context into which you want to activate the font. The `hgdiob` parameter is the handle of the object you want to insert (in this particular case, the font handle that `w.CreateFontIndirect` returns).

Because w.SelectObject requires a device context, you may only call it within a `w.BeginPaint/w.EndPaint`, `w.GetDC/w.ReleaseDC`, or `w.GetWindowDC/w.ReleaseDC` sequence. Of course, the *wpa.hhf* header file contains `#keyword` macro definitions for `SelectObject` so you can call it between a `BeginPaint/EndPaint` sequence thusly:

```
BeginPaint( hwnd, ps, hdc );
       .
```

```
         .
         .
      SelectObject( hFontHandle );
         .
         .
         .
   EndPaint;
```

(note that the `SelectObject` macro only requires a single parameter because the `BeginPaint` macro automatically supplies the device context parameter for you.)

The `w.SelectObject` function (and, therefore, the `SelectObject` macro) returns the handle of the previous font in the EAX register. You should save this handle and then restore the original font (via another `SelectObject` invocation) when you are done using the font, e.g.,

```
   BeginPaint( hwnd, ps, hdc );
         .
         .
         .
      SelectObject( hFontHandle );
      mov( eax, oldFontHandle );
         .
         .
         .
      SelectObject( oldFontHandle );

   EndPaint;
```

If you don t know the characteristics of the font currently selected into the device context, you can call the `w.GetTextMetrics` API/`GetTextMetrics` macro in order to retrieve this information from Windows. The `w.GetTextMetrics` call as the following prototype:

```
static
    GetTextMetrics: procedure
    (
            hdc             :dword;
        var lptm            :TEXTMETRIC
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__GetTextMetricsA@8" );
```

As with most GDI function calls that require a device context, you d call this function between `w.BeginPaint/w.EndPaint`, etc. The *wpa.hhf* header file supplies a `GetTextMetrics` macro (sans the first parameter that `BeginPaint` automatically fills in for you) that you can call within the `BeginPaint..EndPaint` sequence. The lptm parameter is the address of a `w.TEXTMETRIC` object, which takes the following form:

```
type
        TEXTMETRIC: record

            tmHeight            :dword;
            tmAscent            :dword;
            tmDescent           :dword;
            tmInternalLeading   :dword;
            tmExternalLeading    :dword;
            tmAveCharWidth       :dword;
            tmMaxCharWidth       :dword;
            tmWeight             :dword;
            tmOverhang           :dword;
            tmDigitizedAspectX   :dword;
            tmDigitizedAspectY   :dword;
            tmFirstChar          :byte;
            tmLastChar           :byte;
            tmDefaultChar        :byte;
            tmBreakChar          :byte;
            tmItalic             :byte;
            tmUnderlined         :byte;
            tmStruckOut          :byte;
            tmPitchAndFamily     :byte;
            tmCharSet            :byte;

        endrecord;
```

As you can see, many of these fields correspond to the values you pass in the `w.LOGFONT` structure when you create the font in the first place. After creating a font and selecting it into the device context, you can call `GetTextMetrics` to populate this data structure to verify that you ve got a font with the values you expect.

Another important reason for calling `GetTextMetrics` is to obtain the height of the font you re currently using. You can use this height information to determine how far apart to space lines when drawing text to a window. In order to determine the nominal spacing between lines of text for a given font, simply add the values of the `tmHeight` and `tmExternalLeading` fields together. This sum provides the value you should add to the y-coordinate of the current line of text to determine the y-coordinate of the next line on the display. The example code in this book up to this point have always used a fixed distance of 20 pixels or so. While this is sufficient for the system font (and the examples you ve seen), using a fixed distance like this is very poor practice; were the user to select in a larger system font, the lines of text could overlap if you use fixed height values.

When you are done with a font you have to explicitly *destroy* it. Fonts you select into a device context are persistent - that is, they hang around (taking up system resources) once your program terminates unless you explicitly tell Windows to delete those fonts. Failure to delete a font when you re through with it can lead to a *resource leak*. Internally, Windows only supports a limited number of resources for a given device context. If you fail to delete a resource you ve selected into the context, and you lose track of the associated resource (i.e., font) handle, there is no way to recover that resource short of re-booting Windows. Therefore, you should take special care to delete all fonts when your done using them in your application via the `w.DeleteObject` API function. Here s the HLA prototype for the Windows version of this API function:

```
static
    DeleteObject: procedure
    (
        hObject :dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__DeleteObject@4" );
```

The single parameter is the handle of the object you wish to delete. For a font, this would be the font handle that `w.CreateFontIndirect` returns. You will note that the `w.DeleteObject` function does not require a device context parameter value. Therefore, you may call this function anywhere, not just between a `BeginPaint..End-`

Paint (or comparable) sequence. For this same reason, there is no DeleteObject macro that is part of the BeginPaint context-free macro declaration.

One problem with the w.CreateFontIndirect function is that it requires that you know the name of the font you want to create (or you have to be willing to live with a generic font that the system chooses for you). Although all modern (desktop) Windows platforms supply 13 basic True Type fonts, it s perfectly reasonable for the user to have installed additional fonts on the system. There is no reason your applications should limit users to the original set of fonts provided with Windows if they ve installed additional fonts. The only question is: how do you determine those font names so you can supply the name in the w.LOGFONT record that you pass to w.CreateFontIndirect? Well, in modern Windows systems this is actually pretty easy; you bring up a font selection control window (provided by Windows) and you let the Windows code handle all the dirty work for you. We ll talk about this option in the chapter on controls later in this book. The other solution is to enumerate the fonts yourself and then pick a font from the list you ve created.

Windows provides a function, w.EnumFontFamilies, that will iterate through all of the available fonts in the system and provide you with the opportunity to obtain each font name. The w.EnumFontFamilies function has the following prototype:

```
type
   FONTENUMPROC:
      procedure
      (
         var   lpelf    :ENUMLOGFONT;
         var   lpntm    :NEXTEXTMETRIC;
               FontType :dword;
               lParam2  :dword
      );


static
   EnumFontFamilies:
      procedure
      (
               hdc                  :dword;
               lpszFamily           :string;
               lpEnumFontFamProc    :FONTENUMPROC;
         var   lParam               :var
      );
         @stdcall;
         @returns( "eax" );
         @external( "__imp__EnumFontFamiliesA@16" );
```

The w.EnumFontFamilies function requires these parameters:

¥ hdc - The handle for the device context whose fonts you wish to enumerate. Note that w.EnumFontFamilies always iterates over the fonts for a specific device context.

¥ lpszFamily - the address of a zero-terminated (e.g., HLA) string. This string must contain the name of the font family you wish to enumerate, or NULL if you want the function to enumerate all font families in the system.

¥ lpEnumFontFamProc - the address of a call back function. Windows will call this function once for each font it enumerates. Your application must supply this function and Windows will pass information about the font to this callback function. Note that the call function s declaration must exactly match the w.FONTENUMPROC definition.

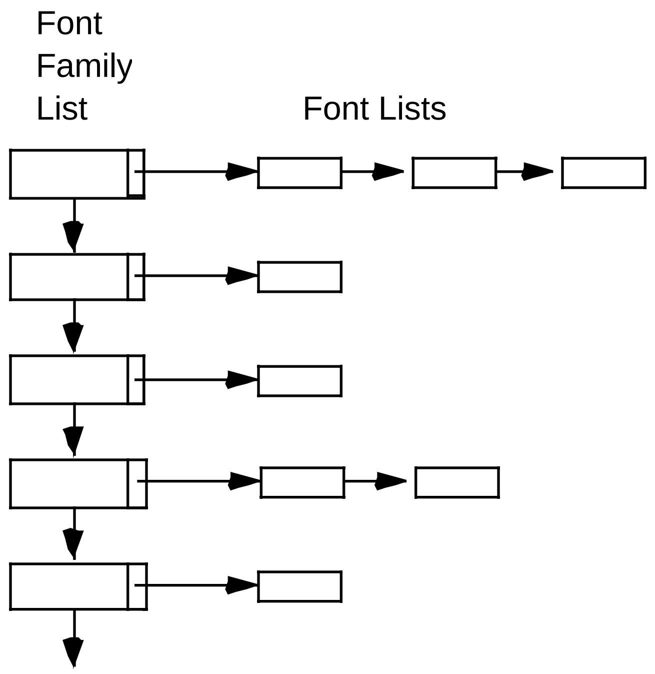
¥ `lParam` - this is a 32-bit application-specific piece of data that the font numeration code passes on to the callback routine (in the lParam2 parameter). Generally, you will pass some application-specific data to the callback routine (such as the address where the callback routine can store the information that Windows passes to it) in this parameter.

For each font or font family present in the system, Windows will call the font enumeration callback routine whose address you pass as the `lpEnumFontFamProc` parameter to `w.EnumFontFamilies`. Windows will pass this callback procedure pointers to `w.LOGFONT` and `w.TEXTMETRIC` data structures that describe the current font. The *fonts.hla* sample program (in the listing that follows) demonstrates the use of the `w.EnumFontFamilies` function to create a list of all the available fonts in the system, as well as display an example of each font.

The *fonts.hla* program captures the `w.WM_CREATE` message for the main window to determine when the window is first created and the program can enumerate all the fonts in the system. The message handle for the create message begins by enumerating all the font families in the system. Then, for each of the font families, it enumerates each font in that family. Because the program doesn t know, beforehand, how many fonts are present in the system, this application uses a *list* data structure that grows dynamically with each font the program enumerates. If you re concerned about linked list algorithms and node insertion or node traversal algorithms, you re in for a pleasant surprise: HLA provides a generic list class that makes the creation and manipulation of lists almost trivial. The *fonts.hla* program takes advantage of this feature of the HLA standard library to reduce the amount of effort needed to create a dynamically sizeable list.

The fonts application actually needs to maintain a two-dimensional list structure. The main list is a list of font families (that is, each node in the list represents a single font family). Each font family also has a list of fonts that are members of that family (see Figure 6-12)

Figure 6-12: Font Family List and Font List Structures



Here s the data structure for the fFamily_t class that maintains the list of font families in the system:

```
type
    fFamily_t:
        class inherits( node );
            var
                familyName  :string;            // Font family name.
                fonts       :pointer to list;   // List of fonts in family.

            override procedure create;
            override method destroy;

        endclass;
```

The `familyName` field points at a string that holds the font family s name. The `fonts` field points at a list of font_t nodes. These nodes have the following structure:

```
type
    font_t:
        class inherits( node );
            var
                tm          :w.TEXTMETRIC;
                lf          :w.LOGFONT;
                fontName    :string;

            override procedure create;
            override method destroy;

        endclass;
```

The `create` procedure and `destroy` method are the conventional class constructors and destructors that allocate storage for objects (create) and deallocate storage when the application is through with them. See the HLA documentation or The Art of Assembly Language Programming for more details on class constructors and destructors.

The *fonts.hla* `Create` procedure, that handles the `w.WM_CREATE` message, takes the following form:

```
// create:
//
//  The procedure responds to the "w.WM_CREATE" message. It enumerates
// the available font families.

procedure Create( hwnd: dword; wParam:dword; lParam:dword ); @returns( "eax" );
var
    hdc :dword;

begin Create;

    push( esi );
    push( edi );
    GetDC( hwnd, hdc );

        // Enumerate the families:

        w.EnumFontFamilies( hdc, NULL, &FontFamilyCallback, NULL );

        // Enumerate the fonts appearing in each family:

        foreach fontFamList.itemInList() do
```

```
            w.EnumFontFamilies
            (
                hdc,
                (type fFamily_t [esi]).familyName,
                &EnumSingleFamily,
                [esi]
            );

        endfor;

    ReleaseDC;
    pop( edi );
    pop( esi );
    mov( 0, eax );  // Return success.

end Create;
```

An interesting thing to note in this procedure is that it uses `GetDC` and `ReleaseDC` to obtain and release a device context (needed by `w.EnumFontFamilies`). Because this procedure is not handling a `w.WM_PAINT` message, we cannot use `BeginPaint` and `EndPaint`. This is a good example of when you need to use `GetDC` and `ReleaseDC`.

The first call to `w.EnumFontFamilies` in this code is responsible for building the fontFamList object that is the list of font families. The fact that the second parameter is NULL (it normally points at a string containing a font family name) tells the `w.EnumFontFamilies` function to enumerate only the families, not the individual fonts within a family. The `w.EnumFontFamilies` function will call the `FontFamilyCallback` function (whose address is passed as the third parameter) once for each font family in the system. It is `FontFamilyCallback's` responsibility to actually build the list of font families. Here s the code that builds this list:

```
// Font callback function that enumerates the font families.
//
//  On each call to this procedure we need to create a new
// node of type fFamily_t, initialize that object with the
// appropriate font information, and append the node to the
// end of the "fontFamList" list.

procedure FontFamilyCallback
(
    var lplf        :w.LOGFONT;
    var lpntm       :w.TEXTMETRIC;
        nFontType   :dword;
        lparam      :dword
);
    @stdcall;
    @returns( "eax" );
var
    curFam  :pointer to fFamily_t;

begin FontFamilyCallback;

    push( esi );
    push( edi );

    // Create a new fFamily_t node to hold this guy:
```

```
    fFamily_t.create();
    mov( esi, curFam );

    // Append node to the end of the font families list:

    fontFamList.append( curFam );

    // Initialize the font family object we just created:

    mov( curFam, esi );

    // Initialize the string containing the font family name:

    mov( lplf, eax );
    str.a_cpyz( (type w.LOGFONT [eax]).lfFaceName );
    mov( eax, (type fFamily_t [esi]).familyName );

    // Create a new list to hold the fonts in this family (initially,
    // this list is empty).

    list.create();
    mov( curFam, edi );
    mov( esi, (type fFamily_t [edi]).fonts );

    // Return success

    mov( 1, eax );

    pop( edi );
    pop( esi );

end FontFamilyCallback;
```

The first thing this function does is create a new `fFamily_t` object (by calling the create procedure for this class) and then appends this new object to the end of the `fontFamList` list. After adding this node to the font family list, the `create` procedure makes a copy of the font family s name and stores this into the object s `familyName` field (the `str.a_cpyz` standard library function converts a zero-terminated string to an HLA string and returns a pointer to that string in EAX, just in case you re wondering). Finally, this constructor creates an empty list to hold the individual fonts (that the application adds later).

Because the `w.EnumFontFamilies` function calls the `FontFamilyCallback` function once for each font family in the system, the `FontFamilyCallback` function winds up create a complete list (`fontFamList`) with all the font families present (because on each call, this function appends a font family object to the end of the `fontFamList`). When the `w.EnumFontFamilies` function returns to the `Create` procedure, therefore, the `fontFamList` contains a list of font family names as well as a set of empty lists ready to hold the individual font information. To fill in these empty font lists, the `Create` procedure simply needs to iterate over the `fontFamList` and call `w.EnumFontFamilies` for each of the individual font families. The `foreach` loop in the `Create` procedure execute s the HLA standard library `itemInList` iterator that steps through each node in the `fontFamList` list[1]. This `foreach` loop calls the `w.EnumFontFamilies` function for each node in the font families list. This call to `w.EnumFontFamilies`, however, passes the current font family name, so it only enumerates those fonts belonging to that specific family. There are two other differences between this call and the earlier call to `w.EnumFont-`

1. For details on the foreach loop and iterators, please consult the HLA documentation or *The Art of Assembly Language*.

`Families`: this call passes the address of the `EnumSingleFamily` procedure and it also passes the address of the current font family list node in the `lparam` parameter (which Windows passes along to `EnumSingleFamily` for each font). The `EnumSingleFamily` needs the address of the parent font family node so it can append a `font_t` object to the end of the `fonts` list present in each font family node. Here s the code for the `EnumSingleFamily` procedure:

```
// Font callback function that enumerates a single font.
// On entry, lparam points at a fFamily_t element whose
// fonts list we append the information to.

procedure EnumSingleFamily
(
    var lplf        :w.LOGFONT;
    var lpntm       :w.TEXTMETRIC;
        nFontType   :dword;
        lparam      :dword
);
    @stdcall;
    @returns( "eax" );

var
    curFont :pointer to font_t;

begin EnumSingleFamily;

    push( esi );
    push( edi );

    // Create a new font_t object to hold this font's information:

    font_t.create();
    mov( esi, curFont );

    // Append the new font to the end of the family list:

    mov( lparam, esi );
    mov( (type fFamily_t [esi]).fonts, esi );
    (type list [esi]).append_last( curFont );

    // Initialize the string containing the font family name:

    mov( curFont, esi );
    mov( lplf, eax );
    str.a_cpyz( (type w.LOGFONT [eax]).lfFaceName );
    mov( eax, (type font_t [esi]).fontName );

    // Copy the parameter information passed to us into the
    // new font_t object:

    lea( edi, (type font_t [esi]).tm );
    mov( lpntm, esi );
    mov( @size( w.TEXTMETRIC), ecx );
    rep.movsb();

    mov( curFont, esi );
    lea( edi, (type font_t [esi]).lf );
    mov( lplf, esi );
    mov( @size( w.LOGFONT ), ecx );
    rep.movsb();

    mov( 1, eax ); // Return success

    pop( edi );
    pop( esi );

end EnumSingleFamily;
```

Like the `fontFamilyCallback` procedure, this procedure begins by create a new object (`font_t` in this case). The `EnumSingleFamily` appends this node to the end of the fonts list that is a member of some font family node (whose address Windows passes into this procedure in the `lparam` parameter). After creating the new node and appending it to the end of a `fonts` list, this procedure initializes the fields of the new object. First, this code creates an HLA string with the font s name (just as the `fontFamilyCallback` function did). This procedure also copies the `w.TEXTMETRIC` and `w.LOGFONT` data passed in as parameters into the `font_t` object (this particular application doesn t actually use most of this information, but this example code copies everything in case you want to cut and paste this code into another application).

When the `foreach` loop in the `Create` procedure finishes execution, the `Create` procedure has managed to build the entire two-dimensional font list data structure. If you ve ever created a complex data structure like this before, you can probably appreciate all the work that the HLA lists class and the Windows `w.EnumFontFamilies` is doing for you. While this code isn t exactly trivial, the amount of code you d have to write to do all this list management on your own is tremendous. The presence of the HLA Standard Library saves considerable effort in this particular application.

Once the `Create` procedure constructs the font lists, the only thing left to do is to display the font information. As you d probably expect by now, the `Paint` procedure handles this task. Just to make things interesting (as well as to demonstrate how to select new fonts into the device context), the paint procedure draws the font family name to the window using the system font (which is readable) and then displays some sample text for each of the fonts using each font to display that information. Here s the `Paint` procedure and the code that pulls this off:

```
// Paint:
//
//  This procedure handles the "w.WM_PAINT" message.

procedure Paint( hwnd: dword; wParam:dword; lParam:dword ); @returns( "eax" );
var
    hdc         :dword;             // Handle to video display device context
    yCoordinate :dword;             // Y-Coordinate for text output.
    newFont     :dword;             // Handle for new fonts we create.
    oldFont     :dword;             // Saves system font while we use new font.
    ps          :w.PAINTSTRUCT;     // Used while painting text.
    outputMsg   :string;            // Holds output text.
    defaultHt   :dword;             // Default font height.
    tm          :w.TEXTMETRIC;      // Default font metrics

begin Paint;

    tstralloc( 256 );       // Allocate string storage on the stack
    mov( eax, outputMsg );  //  for our output string.

    push( edi );
    push( ebx );

    BeginPaint( hwnd, ps, hdc );

        // Get the height of the default font so we can output font family
        // names using the default font (and properly skip past them as
        // we output them):

        w.GetTextMetrics( hdc, tm );
        mov( tm.tmHeight, eax );
        add( tm.tmExternalLeading, eax );
        mov( eax, defaultHt );

        // Initialize the y-coordinate before we draw the font samples:

        mov( -10, yCoordinate );

        // Okay, output a sample of each font:

        push( esi );
        foreach fontFamList.itemInList() do

            // Add in a little extra space for each new font family:

            add( 10, yCoordinate );

            // Write a title line in the system font (because some fonts
            // are unreadable, we want to display the family name using
            // the system font).
            //
            // Begin by computing the number of fonts so we can display
            // that information along with the family title:

            push( esi );
            mov( (type fFamily_t [esi]).fonts, ebx );
            (type list [ebx]).numNodes();
            pop( esi );

            if( eax == 1 ) then

                // Only one font in family, so write "1 font":

                str.put
                (
                    outputMsg,
                    "Font Family: ",
                    (type fFamily_t [esi]).familyName,
                    " (1 font)"
                );

            else

                // Two or more fonts in family, so write "n fonts":

                str.put
                (
                    outputMsg,
                    "Font Family: ",
                    (type fFamily_t [esi]).familyName,
                    " (",
                    (type uns32 eax),
                    " fonts)"
                );

            endif;
            w.TextOut
            (
                hdc,
                10,
                yCoordinate,
                outputMsg,
                str.length(outputMsg)
            );

            // Skip down vertically the equivalent of one line in the current
            // font's size:

            mov( defaultHt, eax );
            add( eax, yCoordinate );

            // For each of the fonts in the current font family,
            // output a sample of that particular font:

            mov( (type fFamily_t [esi]).fonts, ebx );
            foreach (type list [ebx]).itemInList() do

                // Create a new font based on the current font
                // we're processing on this loop iteration:

                w.CreateFontIndirect( (type font_t [esi]).lf );
                mov( eax, newFont );

                // Select the new font into the device context:

                w.SelectObject( hdc, eax );
                mov( eax, oldFont );

                // Compute the font size in points.  This is computed
                // as:
                //
                //  ( <font height> * 72 ) / <font's Y pixels/inch>

                w.GetDeviceCaps( hdc, w.LOGPIXELSY ); // Y pixels/inch
                mov( eax, ecx );
                mov( (type font_t [esi]).lf.lfHeight, eax ); // Font Height
                imul( 72, eax );
                div( ecx, edx:eax );

                // Output the font info:

                str.put
                (
                    outputMsg,
                    (type font_t [esi]).fontName,
                    " (Size in points: ",
                    (type uns32 eax),
                    ')'
                );
                w.TextOut
                (
                    hdc,
                    20,
                    yCoordinate,
                    outputMsg,
                    str.length( outputMsg )
                );

                // Adjust the y-coordinate to skip over the
                // characters we just emitted:

                mov( (type font_t [esi]).tm.tmHeight, eax );
                add( (type font_t [esi]).tm.tmExternalLeading, eax );
                add( eax, yCoordinate );

                // Free the font resource and restore the original font:

                w.SelectObject( hdc, oldFont );
                w.DeleteObject( newFont );

            endfor;

        endfor;
        pop( esi );

    EndPaint;

    pop( ebx );
    pop( edi );
    mov( 0, eax );  // Return success

end Paint;
```

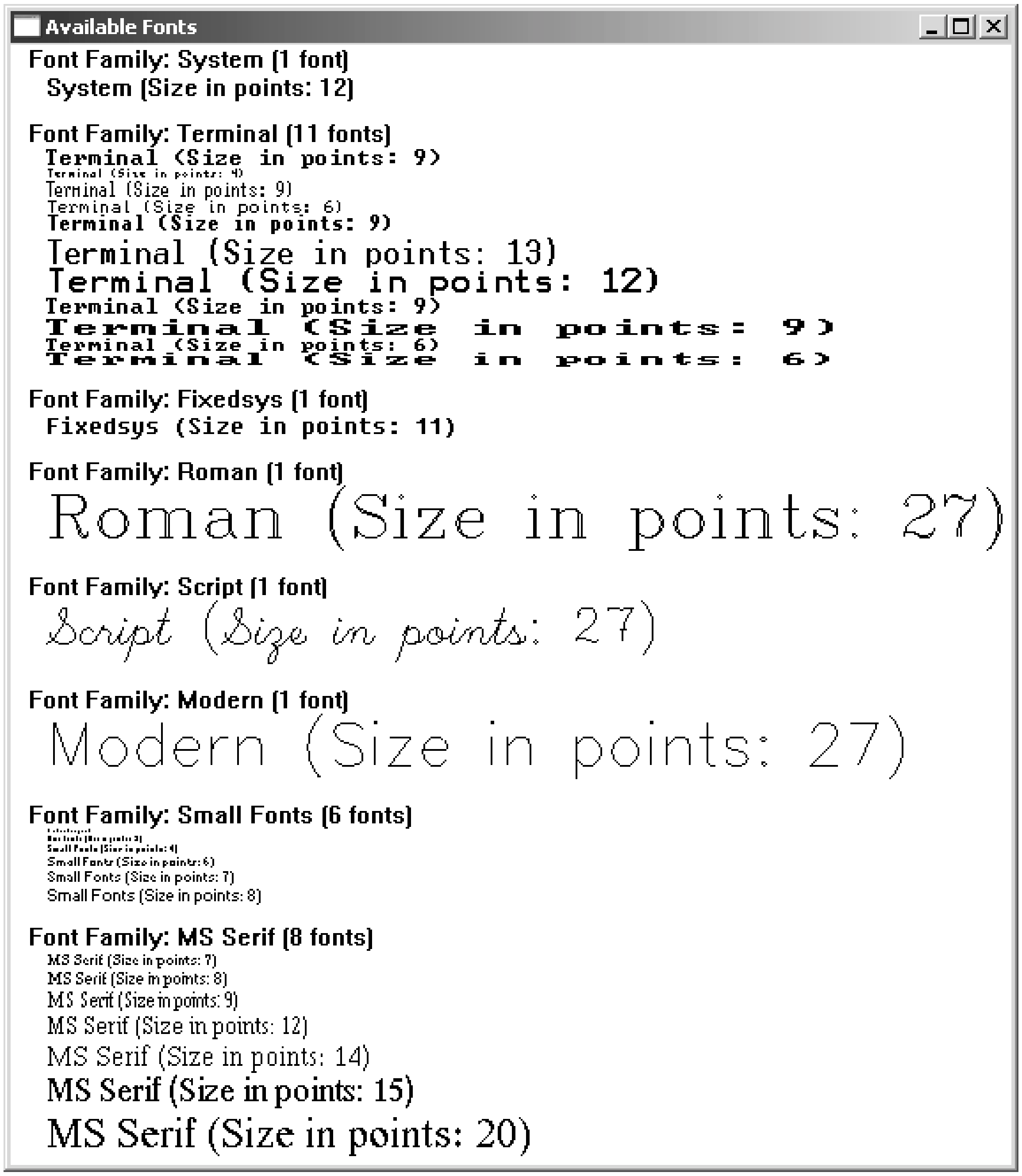
The `Paint` procedure operates using two nested `foreach` loops. The outermost `foreach` loop iterates over each node in the font families list, the inner-most `foreach` loop iterates over each node in the fonts list attached to each of the nodes in the font families list. The action, therefore, is to choose a font family, iterator over each font in that family, move on to the next family, iterate over each font in that new family, and repeat for each font family in the `fontFamList` object.

For each font family, the `Paint` procedure draws a line of text (in the system font) specifying the font family s name. `Paint` tracks the output position (y-coordinate) using the `yCoordinate` local variable. For each line of text that this procedure outputs, it adds the height of the font (plus external leading) to the `yCoordinate` variable so that the next output line will occur below the current output line.

Once the `Paint` procedure outputs the family name (and the number of fonts available in that family), it executes the nested `foreach` loop that displays a sample of each font in the family (see Figure 6-13 for typical output; note that your display may differ depending upon the fonts you ve installed in your system). The `foreach` loop begins by creating a new font using the current `fonts` node s `fontName` string by calling `w.CreateFontIndirect`. Next, the code selects this font into the device context via a `w.SelectObject` call. Finally, just before writing the font sample to the window, this code sequence computes the size of the font (in points) by multiplying the height of the font (in pixels) by the number of points/inch (72) and then divides this product by the number of pixels/inch in the device context. After all this work, `foreach` loop displays the name of that font (in that font) along with the size of the font.

Probably the first thing you ll notice about *font.hla s* output is that it chops off the font listing at the bottom of the window. Fear not, we ll take a look at the solution to this problem (scroll bars) in the very next section.

Figure 6-13: Fonts Output



```
Font Family: System (1 font)
  System (Size in points: 12)

Font Family: Terminal (11 fonts)
  Terminal (Size in points: 9)
  Terminal (Size in points: 4)
  Terminal (Size in points: 9)
  Terminal (Size in points: 6)
  Terminal (Size in points: 9)
  Terminal (Size in points: 13)
  Terminal (Size in points: 12)
  Terminal (Size in points: 9)
  Terminal (Size in points: 9)
  Terminal (Size in points: 6)
  Terminal (Size in points: 6)

Font Family: Fixedsys (1 font)
  Fixedsys (Size in points: 11)

Font Family: Roman (1 font)
  Roman (Size in points: 27)

Font Family: Script (1 font)
  Script (Size in points: 27)

Font Family: Modern (1 font)
  Modern (Size in points: 27)

Font Family: Small Fonts (6 fonts)
  [illegible]
  [illegible]
  Small Fonts (Size in points: 4)
  Small Fonts (Size in points: 6)
  Small Fonts (Size in points: 7)
  Small Fonts (Size in points: 8)

Font Family: MS Serif (8 fonts)
  MS Serif (Size in points: 7)
  MS Serif (Size in points: 8)
  MS Serif (Size in points: 9)
  MS Serif (Size in points: 12)
  MS Serif (Size in points: 14)
  MS Serif (Size in points: 15)
  MS Serif (Size in points: 20)
```

Here s the complete source code to the `fonts.hla` application:

```
// Fonts.hla:
//
```

```
// Displays all the fonts available in the system.

program Fonts;
#include( "w.hhf" )         // Standard windows stuff.
#include( "wpa.hhf" )       // "Windows Programming in Assembly" specific stuff.
#include( "strings.hhf" )   // String functions.
#include( "memory.hhf" )    // tstralloc is in here
#include( "lists.hhf" )     // List abstract data type appears here
?@nodisplay := true;        // Disable extra code generation in each procedure.
?@nostackalign := true;     // Stacks are always aligned, no need for extra code.

type
    // font_t objects are nodes in a list of fonts belonging to a single
    // family.  Such lists appearing in a font family object (class fFamily_t).

    font_t:
        class inherits( node );
            var
                tm          :w.TEXTMETRIC;
                lf          :w.LOGFONT;
                fontName    :string;

            override procedure create;
            override method destroy;

        endclass;


    // fFamily_t objects are nodes in a list of font families. Each node in
    // this list represents a single font family in the system.  Also note
    // that these objects contain a list of fonts that belong to that
    // particular family.

    fFamily_t:
        class inherits( node );
            var
                familyName  :string;            // Font family name.
                fonts       :pointer to list;   // List of fonts in family.

            override procedure create;
            override method destroy;

        endclass;


static
    hInstance   :dword;         // "Instance Handle" supplied by Windows.

    wc          :w.WNDCLASSEX;  // Our "window class" data.
    msg         :w.MSG;         // Windows messages go here.
    hwnd        :dword;         // Handle to our window.

    fontFamList :pointer to list;   // List of font families.


readonly
```

```
    ClassName:  string := "FontsWinClass";      // Window Class Name
    AppCaption: string := "Available Fonts";    // Caption for Window

// The following data type and DATA declaration
// defines the message handlers for this program.

type
    MsgProc_t:  procedure( hwnd:dword; wParam:dword; lParam:dword );

    MsgProcPtr_t:
        record

            MessageValue:   dword;
            MessageHndlr:   MsgProc_t;

        endrecord;



// The dispatch table:
//
//  This table is where you add new messages and message handlers
//  to the program.  Each entry in the table must be a tMsgProcPtr
//  record containing two entries: the message value (a constant,
//  typically one of the wm.***** constants found in windows.hhf)
//  and a pointer to a "tMsgProc" procedure that will handle the
//  message.

readonly

    Dispatch:   MsgProcPtr_t; @nostorage;

        MsgProcPtr_t
            MsgProcPtr_t:[  w.WM_DESTROY, &QuitApplication   ],
            MsgProcPtr_t:[  w.WM_PAINT,   &Paint             ],
            MsgProcPtr_t:[  w.WM_CREATE,  &Create            ],

            // Insert new message handler records here.

            MsgProcPtr_t:[  0, NULL ];   // This marks the end of the list.



/**************************************************************************/
/*          A P P L I C A T I O N   S P E C I F I C   C O D E            */
/**************************************************************************/


// Methods and procedures for the font_t class.
// Remember, ESI contains the "THIS" pointer upon entry to these procedures
// and methods.
//
//
// create-  Constructor for a font_t node in a font list.
//          Note: returns pointer to object in ESI.  Allocates
//          new storage for a node object if ESI contains NULL upon entry.
```

```
procedure font_t.create;
begin create;

    push( eax );
    if( esi == NULL ) then

        // If this is a bare constructor call (font_t.create) then
        // allocate storage for a new node:

        malloc( @size( font_t ) );
        mov( eax, esi );

    endif;
    mov( NULL, this.fontName );
    push( ecx );
    push( edi );

    // Zero out the tm and lf data structures:

    lea( edi, this.tm );
    mov( @size( w.TEXTMETRIC ), ecx );
    xor( eax, eax );
    rep.stosb;

    lea( edi, this.lf );
    mov( @size( w.TEXTMETRIC ), ecx );
    rep.stosb;

    pop( ecx );
    pop( edi );
    pop( eax );

end create;

// font_t.destroy-
//
//  Destructor for a font_t node object.
//  Because this program never frees any item in the list, there
//  really is no purpose for this function; it is required by the
//  node class, hence its presence here.  If this application wanted
//  to free the items in the font lists, it would clear the storage
//  allocated to the fontName field (if non-NULL and on the heap)
//  and it would free the storage associated with the node itself.
//  The following code demonstrates this, even though this program
//  never actually calls this method.

method font_t.destroy;
begin destroy;

    // Free the string name if it was allocated on the heap:

    if( this.fontName <> NULL ) then

        if( strIsInHeap( this.fontName )) then

            strfree( this.fontName );
        endif;

    endif;

    // Free the object if it was allocated on the heap:

    if( isInHeap( esi /* this */ )) then

        free( esi );

    endif;

end destroy;







// Methods and procedures for the fFamily_t class.
// Remember, ESI contains the "THIS" pointer upon entry to these procedures
// and methods.
//
//
// create-  Constructor for a fFamily_t node in a font family list.
//          Note: returns pointer to object in ESI.  Allocates
//          new storage for a node object if ESI contains NULL upon entry.

procedure fFamily_t.create;
begin create;

    push( eax );
    if( esi == NULL ) then

        // If this is a bare constructor call (fFamily_t.create) then
        // allocate storage for a new node:

        malloc( @size( fFamily_t ) );
        mov( eax, esi );

    endif;

    // Initialize the family name to NULL (it will be filled in
    // by whomever is enumerating the family lists):

    mov( NULL, this.familyName );

    // Create a new list to hold the font information for this family:

    push( esi );
    font_t.create();
    mov( esi, eax );
    pop( esi );
    mov( eax, this.fonts );

    pop( eax );
```

```
end create;


// fFamily_t.destroy-
//
//  Destructor for a fFamily_t node object.
//  Because this program never frees any item in the list, there
//  really is no purpose for this function; it is required by the
//  node class, hence its presence here.  If this application wanted
//  to free the items in the font lists, it would clear the storage
//  allocated to the familyName field (if non-NULL and on the heap)
//  and it would free the storage associated with the node itself.
//  The following code demonstrates this, even though this program
//  never actually calls this method.

method fFamily_t.destroy;
begin destroy;

    // Free the string name if it was allocated on the heap:

    if( this.familyName <> NULL ) then

        if( strIsInHeap( this.familyName )) then

            strfree( this.familyName );

        endif;

    endif;

    // Free up the font list:

    push( esi );
    mov( this.fonts, esi );
    (type list [esi]).destroy();
    pop( esi );

    // Free the object if it was allocated on the heap:

    if( isInHeap( esi /* this */ )) then

        free( esi );

    endif;

end destroy;




// Font callback function that enumerates the font families.
//
//  On each call to this procedure we need to create a new
// node of type fFamily_t, initialize that object with the
// appropriate font information, and append the node to the
// end of the "fontFamList" list.

procedure FontFamilyCallback
(
    var lplf        :w.LOGFONT;
    var lpntm       :w.TEXTMETRIC;
        nFontType   :dword;
        lparam      :dword
);
    @stdcall;
    @returns( "eax" );
var
    curFam  :pointer to fFamily_t;

begin FontFamilyCallback;

    push( esi );
    push( edi );

    // Create a new fFamily_t node to hold this guy:

    fFamily_t.create();
    mov( esi, curFam );

    // Append node to the end of the font families list:

    fontFamList.append( curFam );

    // Initialize the font family object we just created:

    mov( curFam, esi );

    // Initialize the string containing the font family name:

    mov( lplf, eax );
    str.a_cpyz( (type w.LOGFONT [ eax] ).lfFaceName );
    mov( eax, (type fFamily_t [ esi] ).familyName );

    // Create a new list to hold the fonts in this family (initially,
    // this list is empty).

    list.create();
    mov( curFam, edi );
    mov( esi, (type fFamily_t [ edi] ).fonts );

    // Return success

    mov( 1, eax );

    pop( edi );
    pop( esi );

end FontFamilyCallback;



// Font callback function that enumerates a single font.
// On entry, lparam points at a fFamily_t element whose
// fonts list we append the information to.

procedure EnumSingleFamily
(
    var lplf        :w.LOGFONT;
    var lpntm       :w.TEXTMETRIC;
        nFontType   :dword;
        lparam      :dword
);
    @stdcall;
    @returns( "eax" );

var
    curFont :pointer to font_t;

begin EnumSingleFamily;

    push( esi );
    push( edi );

    // Create a new font_t object to hold this font's information:

    font_t.create();
    mov( esi, curFont );

    // Append the new font to the end of the family list:

    mov( lparam, esi );
    mov( (type fFamily_t [esi]).fonts, esi );
    (type list [esi]).append_last( curFont );

    // Initialize the string containing the font family name:

    mov( curFont, esi );
    mov( lplf, eax );
    str.a_cpyz( (type w.LOGFONT [eax]).lfFaceName );
    mov( eax, (type font_t [esi]).fontName );

    // Copy the parameter information passed to us into the
    // new font_t object:

    lea( edi, (type font_t [esi]).tm );
    mov( lpntm, esi );
    mov( @size( w.TEXTMETRIC), ecx );
    rep.movsb();

    mov( curFont, esi );
    lea( edi, (type font_t [esi]).lf );
    mov( lplf, esi );
    mov( @size( w.LOGFONT ), ecx );
    rep.movsb();

    mov( 1, eax ); // Return success

    pop( edi );
    pop( esi );
```

```
end EnumSingleFamily;



/**************************************************************************
**
** Message Handling Procedures:
*/

// QuitApplication:
//
//  This procedure handles the "wm.Destroy" message.
//  It tells the application to terminate.  This code sends
//  the appropriate message to the main program's message loop
//  that will cause the application to terminate.


procedure QuitApplication( hwnd: dword; wParam:dword; lParam:dword );
begin QuitApplication;

    w.PostQuitMessage( 0 );

end QuitApplication;


// create:
//
//  The procedure responds to the "w.WM_CREATE" message. It enumerates
// the available font families.

procedure Create( hwnd: dword; wParam:dword; lParam:dword ); @returns( "eax" );
var
    hdc :dword;

begin Create;

    push( esi );
    push( edi );
    GetDC( hwnd, hdc );

        // Enumerate the families:

        w.EnumFontFamilies( hdc, NULL, &FontFamilyCallback, NULL );

        // Enumerate the fonts appearing in each family:

        foreach fontFamList.itemInList() do

            w.EnumFontFamilies
            (
                hdc,
                (type fFamily_t [esi]).familyName,
                &EnumSingleFamily,
                [esi]
            );
        endfor;

    ReleaseDC;
    pop( edi );
    pop( esi );
    mov( 0, eax );  // Return success.

end Create;


// Paint:
//
//  This procedure handles the "w.WM_PAINT" message.

procedure Paint( hwnd: dword; wParam:dword; lParam:dword ); @returns( "eax" );
var
    hdc         :dword;             // Handle to video display device context
    yCoordinate :dword;             // Y-Coordinate for text output.
    newFont     :dword;             // Handle for new fonts we create.
    oldFont     :dword;             // Saves system font while we use new font.
    ps          :w.PAINTSTRUCT;     // Used while painting text.
    outputMsg   :string;            // Holds output text.
    defaultHt   :dword;             // Default font height.
    tm          :w.TEXTMETRIC;      // Default font metrics

begin Paint;

    tstralloc( 256 );       // Allocate string storage on the stack
    mov( eax, outputMsg );  //  for our output string.

    push( edi );
    push( ebx );

    BeginPaint( hwnd, ps, hdc );

        // Get the height of the default font so we can output font family
        // names using the default font (and properly skip past them as
        // we output them):

        w.GetTextMetrics( hdc, tm );
        mov( tm.tmHeight, eax );
        add( tm.tmExternalLeading, eax );
        mov( eax, defaultHt );

        // Initialize the y-coordinate before we draw the font samples:

        mov( -10, yCoordinate );

        // Okay, output a sample of each font:

        push( esi );
        foreach fontFamList.itemInList() do

            // Add in a little extra space for each new font family:

            add( 10, yCoordinate );
```

```
// Write a title line in the system font (because some fonts
// are unreadable, we want to display the family name using
// the system font).
//
// Begin by computing the number of fonts so we can display
// that information along with the family title:

push( esi );
mov( (type fFamily_t [ esi] ).fonts, ebx );
(type list [ ebx] ).numNodes();
pop( esi );

if( eax == 1 ) then

    // Only one font in family, so write "1 font":

    str.put
    (
        outputMsg,
        "Font Family: ",
        (type fFamily_t [ esi] ).familyName,
        " (1 font)"
    );

else

    // Two or more fonts in family, so write "n fonts":

    str.put
    (
        outputMsg,
        "Font Family: ",
        (type fFamily_t [ esi] ).familyName,
        " (",
        (type uns32 eax),
        " fonts)"
    );

endif;
w.TextOut
(
    hdc,
    10,
    yCoordinate,
    outputMsg,
    str.length(outputMsg)
);

// Skip down vertically the equivalent of one line in the current
// font's size:

mov( defaultHt, eax );
add( eax, yCoordinate );

// For each of the fonts in the current font family,
// output a sample of that particular font:
```

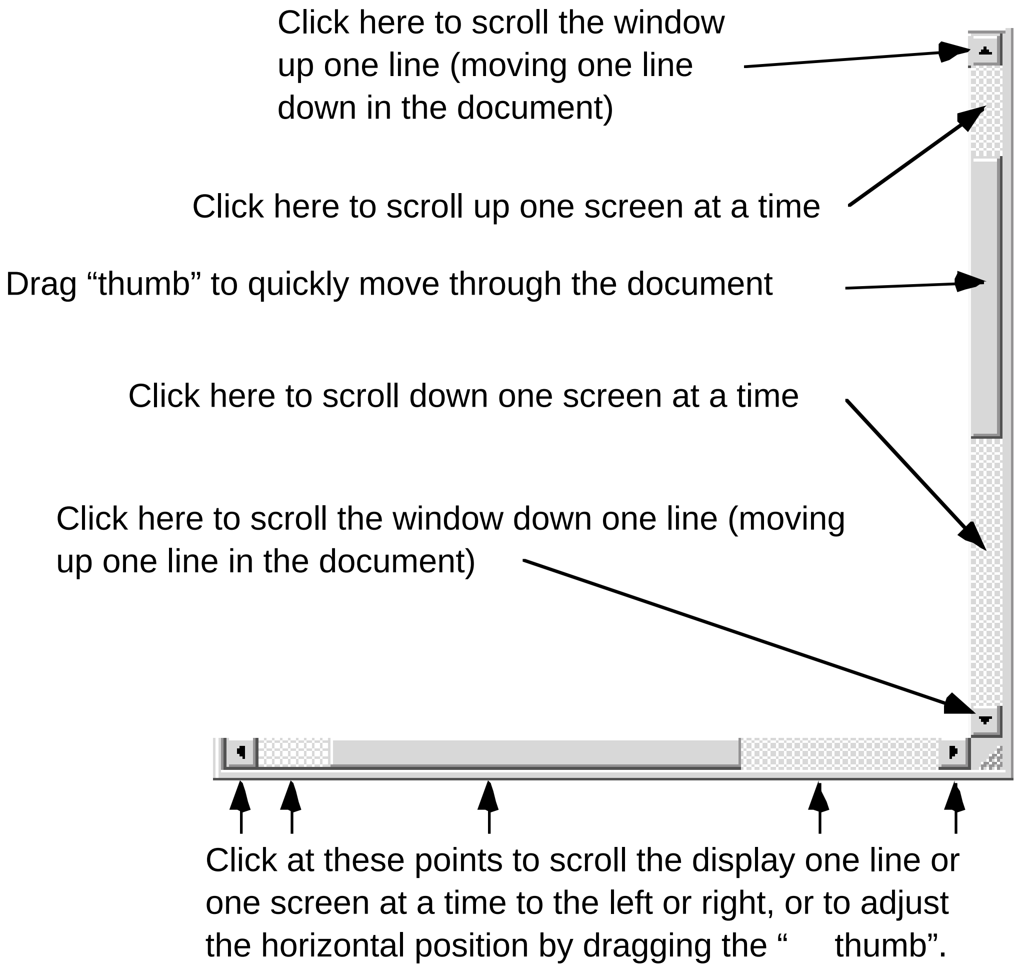
```
mov( (type fFamily_t [ esi] ).fonts, ebx );
foreach (type list [ ebx] ).itemInList() do

    // Create a new font based on the current font
    // we're processing on this loop iteration:

    w.CreateFontIndirect( (type font_t [ esi] ).lf );
    mov( eax, newFont );

    // Select the new font into the device context:

    w.SelectObject( hdc, eax );
    mov( eax, oldFont );

    // Compute the font size in points.  This is computed
    // as:
    //
    //  ( <font height> * 72 ) / <font's Y pixels/inch>

    w.GetDeviceCaps( hdc, w.LOGPIXELSY ); // Y pixels/inch
    mov( eax, ecx );
    mov( (type font_t [ esi] ).lf.lfHeight, eax ); // Font Height
    imul( 72, eax );
    div( ecx, edx:eax );

    // Output the font info:

    str.put
    (
        outputMsg,
        (type font_t [ esi] ).fontName,
        " (Size in points: ",
        (type uns32 eax),
        ')'
    );
    w.TextOut
    (
        hdc,
        20,
        yCoordinate,
        outputMsg,
        str.length( outputMsg )
    );

    // Adjust the y-coordinate to skip over the
    // characters we just emitted:

    mov( (type font_t [ esi] ).tm.tmHeight, eax );
    add( (type font_t [ esi] ).tm.tmExternalLeading, eax );
    add( eax, yCoordinate );

    // Free the font resource and restore the original font:

    w.SelectObject( hdc, oldFont );
    w.DeleteObject( newFont );

endfor;
```

```
        endfor;
        pop( esi );

    EndPaint;

    pop( ebx );
    pop( edi );
    mov( 0, eax );  // Return success

end Paint;

/**************************************************************************/
/*                   End of Application Specific Code                     */
/**************************************************************************/




// 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( hwnd:dword; uMsg:uns32; wParam:dword; lParam:dword  );
    @stdcall;
    @nodisplay;
    @noalignstack;

begin WndProc;

    // uMsg contains the current message Windows is passing along to
    // us.  Scan through the "Dispatch" table searching for a handler
    // for this message.  If we find one, then call the associated
    // handler procedure.  If we don't have a specific handler for this
    // message, then call the default window procedure handler function.

    mov( uMsg, eax );
    mov( &Dispatch, edx );
    forever

        mov( (type MsgProcPtr_t [edx]).MessageHndlr, ecx );
        if( ecx = 0 ) then

            // If an unhandled message comes 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.

            w.DefWindowProc( hwnd, uMsg, wParam, lParam );
            exit WndProc;
        elseif( eax = (type MsgProcPtr_t [ edx] ).MessageValue ) then

            // If the current message matches one of the values
            // in the message dispatch table, then call the
            // appropriate routine.  Note that the routine address
            // is still in ECX from the test above.

            push( hwnd );   // (type tMsgProc ecx)(hwnd, wParam, lParam)
            push( wParam ); //  This calls the associated routine after
            push( lParam ); //  pushing the necessary parameters.
            call( ecx );

            sub( eax, eax ); // Return value for function is zero.
            break;

        endif;
        add( @size( MsgProcPtr_t ), edx );

    endfor;

end WndProc;



// Here's the main program for the application.

begin Fonts;

    // Create the font family list here:

    push( esi );
    list.create();
    mov( esi, fontFamList );
    pop( esi );

    // Set up the window class (wc) object:

    mov( @size( w.WNDCLASSEX ), wc.cbSize );
    mov( w.CS_HREDRAW | w.CS_VREDRAW, wc.style );
    mov( &WndProc, wc.lpfnWndProc );
    mov( NULL, wc.cbClsExtra );
    mov( NULL, wc.cbWndExtra );
    mov( w.COLOR_WINDOW+1, wc.hbrBackground );
    mov( NULL, wc.lpszMenuName );
    mov( ClassName, wc.lpszClassName );

    // Get this process' handle:

    w.GetModuleHandle( NULL );
    mov( eax, hInstance );
    mov( eax, wc.hInstance );

    // Get the icons and cursor for this application:

    w.LoadIcon( NULL, val w.IDI_APPLICATION );
    mov( eax, wc.hIcon );
    mov( eax, wc.hIconSm );

    w.LoadCursor( NULL, val w.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.

    w.RegisterClassEx( wc );

    w.CreateWindowEx
    (
        NULL,
        ClassName,
        AppCaption,
        w.WS_OVERLAPPEDWINDOW,
        w.CW_USEDEFAULT,
        w.CW_USEDEFAULT,
        w.CW_USEDEFAULT,
        w.CW_USEDEFAULT,
        NULL,
        NULL,
        hInstance,
        NULL
    );
    mov( eax, hwnd );

    w.ShowWindow( hwnd, w.SW_SHOWNORMAL );
    w.UpdateWindow( hwnd );

    // Here's the event loop that processes messages
    // sent to our window.  On return from GetMessage,
    // break if EAX contains false and then quit the
    // program.

    forever

        w.GetMessage( msg, NULL, 0, 0 );
        breakif( !eax );
        w.TranslateMessage( msg );
        w.DispatchMessage( msg );

    endfor;

    // The message handling inside Windows has stored
    // the program's return code in the wParam field
    // of the message.  Extract this and return it
    // as the program's return code.

    mov( msg.wParam, eax );
    w.ExitProcess( eax );

end Fonts;
```

6.5: Scroll Bars

One problem that is immediately obvious with the *fonts* program from the previous section is that there is too much information to display in the window at one time. As a result, Windows truncates much of the information when drawing the text to the window. Scroll bars provide the solution for this problem. Scroll bars allow the user to use a window as a window into a larger viewing area (hence the name, window) by selecting the relative coordinate of the upper left-hand corner of the window within the larger viewing area. Applications that need to display more than one window full of information will generally employ a vertical scroll bar (to allow the user to move the display in the window up or down) and a horizontal scroll bar (to allow the user to move the display in the window from side to side). With a properly written application, a user may view any part of the document via the scroll bars (see Figure 6-14).

Figure 6-14: Scroll Bar Actions


Click here to scroll the window up one line (moving one line down in the document)
Click here to scroll up one screen at a time
Drag “thumb” to quickly move through the document
Click here to scroll down one screen at a time
Click here to scroll the window down one line (moving up one line in the document)
Click at these points to scroll the display one line or one screen at a time to the left or right, or to adjust the horizontal position by dragging the “ thumb”.

From the user s perspective, the scroll bars move the document around within the window. From an application s perspective, however, what is really going on is that the scroll bars are repositioning the window over the document. If we think of coordinate (0,0) as being the upper-left hand pixel in the document, then adjust the view using the scroll bars simply defines the coordinate in the document that corresponds to the upper-left hand corner of the window within the document. This is why clicking on the up arrow on the scroll bar actually cause the contents of the window to scroll down. What s really happening is that the user is moving the starting coordinate of the window up on line in the document. Because the top of the window is now displaying one line earlier in the document, the contents of the window shifts down one line. A similar explanation applies to scrolling data left or right in the window via the scroll bars.

Adding scroll bars to your application s windows is very easy. All you ve got to do is supply the `w.WS_VSCROLL` and `w.WS_HSCROLL` constants as part of the window style when calling `w.CreateWindowEx`, e.g.,

```
    w.CreateWindowEx
    (
        NULL,
        ClassName,
        AppCaption,
        w.WS_OVERLAPPEDWINDOW | w.WS_VSCROLL |  w.WS_HSCROLL,
        w.CW_USEDEFAULT,
        w.CW_USEDEFAULT,
        w.CW_USEDEFAULT,
        w.CW_USEDEFAULT,
        NULL,
        NULL,
        hInstance,
        NULL
    );
```

With these constants as part of the window style parameter in the call to `w.CreateWindowEx`, Windows automatically draws the scroll bars (if necessary) and starts sending appropriate messages to your window when the user clicks on a scroll bar or drags the thumb around. Note that although Windows will automatically handle all mouse activities on the scroll bar, Windows does not automatically process keystroke equivalents (e.g., PgUp and PgDn). You will have to handle such keystrokes yourself; we ll discuss how to do that in the chapter on *Event-Driven Input/Output* a little later in this book.

Scroll bars in a window have three important numeric attributes: a minimum range value, a maximum range value, and a current position value. The minimum and maximum range values are two numeric integers that specify the minimum and maximum values that Windows will use as the thumb (scroll box) position within the scroll bar. The current position value is some value, within this range, that represents the current value of the scroll bar s thumb (scroll box). At any time you may query these values using the `w.GetScrollRange` and `w.GetScrollPos` API functions:

```
type
   GetScrollPos: procedure
    (
        hWnd            :dword;
        nBar            :dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__GetScrollPos@8" );

    GetScrollRange: procedure
    (
            hWnd        :dword;
            nBar        :dword;
        var lpMinPos    :dword;
        var lpMaxPos    :dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__GetScrollRange@16" );
```

The `hWnd` parameter to these two functions is either the handle of a window containing a standard horizontal or vertical scroll bar, on the handle of a special scroll bar window you ve created (we ll discuss how to create your own scroll bar windows in the chapter on *Controls, Dialogs, Menus, and Windows* later in this book; for now, we ll just supply the handle of a window that has the `w.WS_VSCROLL` or `w.WS_HSCROLL` window styles).

The nBar parameter in these two functions specifies which scroll bar values to retrieve. This parameter must be one of the following three constants:

¥ `w.SB_CTL` - use this constant if you re reading the value of a custom scroll bar control. When you supply this constant, the `hWnd` parameter must be the handle of the scroll bar control whose value(s) you wish to retrieve.

¥ `w.SB_HORZ` - use this constant if you want to retrieve the position or range values for the horizontal scroll bar in a standard window with the `w.WS_HSCROLL` or `w.WS_VSCROLL` styles.

¥ `w.SB_VERT` - use this constant if you want to retrieve the position or range values for the vertical scroll bar in a standard window with the `w.WS_HSCROLL` or `w.WS_VSCROLL` styles.

The `w.GetScrollPos` function returns the current thumb position in the EAX register. The `w.GetScroll-Range` returns the minimum and maximum positions in the `lpMinPos` and `lpMaxPos` parameters you pass by reference to the function.

Windows scroll bars have a default range of 0..100 (i.e., the scroll position indicates a percentage of the document). The `w.SetScrollRange` API function lets you change the scroll bar range to a value that may be more appropriate for your application. Here s the prototype for this function:

```
type
    SetScrollRange: procedure
    (
        hWnd        :dword;
        nBar        :dword;
        nMinPos     :dword;
        nMaxPos     :dword;
        bRedraw     :boolean
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__SetScrollRange@20" );
```

The `hWnd` and `nBar` parameters have the same meaning as for `w.GetScrollRange`. The `nMinPos` and `nMax-Pos` parameters specify the new minimum and maximum values for the scroll bar s range. Note that `nMinPos` must be less than or equal to `nMaxPos`. If the two values are equal, Windows will remove the scroll bar from the window. These should be unsigned values in the range 0..65535. Technically, Windows allows any unsigned 32-bit value here, but as you ll see in a little bit, it is a bit more efficient to limit the scroll bar positions to 16-bit values. Fortunately, 16-bit resolution is usually sufficient (i.e., you can still scroll a document with 65,536 lines one line at a time when using this resolution). The `bRedraw` parameter (true or false) determines whether Windows will redraw the scroll bar after you call `w.SetScrollRange`. Normally, you d probably want to set this parameter to true unless you re about to call some other function (e.g., `w.SetScrollPos`) that will also redraw the scroll bar; setting bRedraw to false prevents Windows from redrawing the scroll bar twice (which slows down your application and make cause the scroll bar region to flash momentarily).

You may also set the current position of the scroll thumb using the w.SetScrollPos API function:

```
type
    SetScrollPos: procedure
    (
        hWnd        :dword;
        nBar        :dword;
        nPos        :dword;
        bRedraw     :boolean
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__SetScrollPos@16" );
```

The hWnd and nBar parameters have the same meaning as for w.GetScrollPos. The nPos parameter specifies the new thumb position in the scroll bar; this must be a value between the minimum and maximum scroll range values for the scroll bar. The bRedraw parameter has the same meaning as the w.SetScrollRange parameter of the same name: it controls whether Windows will redraw the scroll bar during this function call. Generally, you ll want to redraw the scroll bar at least once at the end of a sequence of modifications to the scroll bar parameters (i.e., range and position). So the last call to w.SetScrollRange or w.SetScrollPos for a given scroll bar should pass true as the bRedraw parameter value.

Although Windows automates much of the work associated with scroll bars, it does not do everything for you. Windows will handle mouse activities on the scroll bar and report such actions to your program. Windows redraws the scroll bar and positions the thumb as the user drags it around. Windows will also send the window procedure (for the window containing the scroll bar) a sequence of messages indicating certain activities within the scroll bar. However, it is your application s responsibility to respond to these messages and actually redraw the window with the new view suggested by the scroll bar values. It is also your application s responsibility to initialize the scroll bar range and update the scroll bar position (as appropriate).

Whenever the user clicks on the scroll bar or drags the thumb, Windows will send a w.WM_HSCROLL or w.WM_VSCROLL message to the window procedure of the window containing the affected scroll bar. The L.O. word of the wParam parameter in the window procedure contains a value that specifies the activity taking place on the scroll bar. This word typically contains one of the values found in Table 6-3.

Table 6-3: wParam Values for a w.WM_HSCROLL or w.WM_VSCROLL Message

Value	Description
w.SB_ENDSCROLL	Indicates that the user has released the mouse button (you can usually ignore this message).
w.SB_LEFT	These two values are actually the same. They indicate scrolling up or to the left (depending on whether the scroll bar is a horizontal or vertical scroll bar, i.e., whether you ve received a w.WM_HSCROLL or w.WM_VSCROLL message.)
w.SB_UP	
w.SB_RIGHT	These two values are actually the same. They indicate scrolling down or to the right (depending on whether the message was a w.WM_VSCROLL or w.VM_HSCROLL.)
w.SB_DOWN	

Value	Description
w.SB_LINEUP	These two values are the same. They indicate scrolling up or left by one unit. Windows passes this value when the user clicks on the up arrow (in a vertical scroll bar) or a left arrow (in a horizontal scroll bar). Your application should scroll the text down one line if this is a w.WM_VSCROLL message, it should scroll the document one position to the right if this is a w.VM_HSCROLL message. Yes, you are scrolling the document in the opposite direction of the scroll message. Remember, the application s perspective of what is happening is opposite of the user s perspective.
w.SB_LINELEFT	
w.SB_LINEDOWN	These two values are the same. They indicate scrolling down or right by one unit. Windows passes this value when the user clicks on the up arrow (in a vertical scroll bar) or a left arrow (in a horizontal scroll bar). Your application should scroll the text down one line if this is a w.WM_VSCROLL message, it should scroll the document one position to the right if this is a w.VM_HSCROLL message. Yes, you are scrolling the document in the opposite direction of the scroll message. Remember, the application s perspective of what is happening is opposite of the user s perspective.
w.SB_LINERGHT	
w.SB_PAGEUP	These two values are the same. They indicate scrolling up or left by one screen. Windows passes this value when the user clicks between the thumb and the up-arrow (in a vertical scroll bar) or between the thumb and the left arrow (in a horizontal scroll bar). Your application should scroll the text down screen (or thereabouts) if this is a w.WM_VSCROLL message, it should scroll the document one screen to the right if this is a w.VM_HSCROLL message. Yes, you are scrolling the document in the opposite direction of the scroll message. Remember, the application s perspective of what is happening is opposite of the user s perspective.
w.SB_PAGELEFT	
w.SB_PAGEDOWN	These two values are the same. They indicate scrolling down or right by one screen. Windows passes this value when the user clicks between the thumb and the down-arrow (in a vertical scroll bar) or between the thumb and the right arrow (in a horizontal scroll bar). Your application should scroll the text up screen (or thereabouts) if this is a w.WM_VSCROLL message, it should scroll the document one screen to the left if this is a w.VM_HSCROLL message. Yes, you are scrolling the document in the opposite direction of the scroll message. Remember, the application s perspective of what is happening is opposite of the user s perspective.
w.SB_PAGERIGHT	
w.SB_THUMBPOSITION	This value indicates that the user has dragged the thumb (scroll box) and has released the mouse button (i.e., this is the end of the drag operation). The H.O. word of wParam indicates the position of the scroll box at the end of the drag operation.
w.SB_THUMBTRACK	This message indicates that the user is currently dragging the thumb in the scroll bar. Windows will send a stream of these messages to the application while the user is dragging the thumb around. The H.O. word of the wParam parameter specifies the current thumb position.

Generally, your applications will need to process the `w.SB_LINEUP`, `w.SB_LINEDOWN`, `w.SB_LINELEFT`, `w.SB_LINERIGHT`, `w.SB_PAGEUP`, `w.SB_PAGEDOWN`, `w.SB_PAGELEFT`, `w.SB_PAGERIGHT`, and `w.SB_THUMBPOSITION`. You have to decide what an appropriate distance is for a line and a screen with respect to these messages. Obviously, if you re creating a text-based application (like a text editor) the concept of line and screen are fairly obvious. However, if you re writing a graphical application, the concept of a line or screen can be somewhat fuzzy. Fortunately, Windows lets you decide how much screen real estate to scroll in response to these messages.

Optionally, you applications may also want to process the `w.SB_THUMBTRACK` messages as well as `w.SB_THUMBPOSITION` messages. The decision of whether to support or ignore `w.SB_THUMBTRACK` messages really depends upon the speed of your application. If you can rapidly redraw the entire screen, then supporting `w.SB_THUMBTRACK` messages provides an extra convenience for your end user. For example, most text editors and word processors support this message so that the user can quickly scan text as it scrolls by while they are dragging the scroll bar thumb around. This is a *very* handy feature to provide if the application can keep up with the user s drag speed. However, if your application cannot instantly (or very close to instantly) redraw the entire screen, then supporting the `w.SB_THUMBTRACK` operation can be an exercise in frustration for your users. There are few programs more frustrating to use than those that process these operational requests but cannot do so instantly. For example, if the user of a drawing program has created a particularly complex drawing that requires several seconds to redraw a single screen full of data, they will become very annoyed if that application processes `w.SB_THUMBTRACK` scrolling messages; when they attempt to drag the scroll bar thumb around, the application will take a few seconds to redraw the screen image, then scroll up a slight amount and take another few seconds to redraw the screen, scroll up a slight amount and take another few seconds... In an extreme case, the application could wind up taking *minutes* to scroll just a few pages through the document. Such applications should simply ignore the `w.SB_THUMBTRACK` scrolling messages and process only `w.SB_THUMBPOSITION` requests. Because Windows only sends a single `w.SB_THUMBPOSITION` message when the user drags the thumb around, the user will only have to sit through one redraw of the window.

You should note that Windows returns a 16-bit thumb position in the H.O. word of wParam in response to a `w.SB_THUMBPOSITION` or `w.SB_THUMBTRACK` message. If you need to obtain a 32-bit position, you can call the Windows API functions `w.GetScrollPos` or `w.GetScrollInfo`[2] to obtain complete information about the current scroll thumb position. To avoid having to make such a call, you should try to limit the range of your scroll bar values to 0..65535 (16-bits).

Armed with this information, it s now possible to correct the problem with the *fonts* program from the previous section. This book, however, will leave it up to you to make the appropriate modifications to that program. In the interests of presenting as many ideas as possible, we ll write a short program, inspired by a comparable program in Petzold s Programming Windows... book, that displays *system metric* values. This program, *sysmet.hla*, displays some useful information about your particular computer system, with one line per value displayed. Unless you ve got a really big video display, you re probably not going to see all of this information on the screen at one time. Even if you ve got a sufficiently large display, you might not want the window to consume so much screen real estate while you re running the *sysmets* application. Therefore, this program employs both vertical and horizontal scroll bars to allow you to make the window as small as is reasonably possible and still be able to view all the information it has to display.

The `w.GetSystemMetrics` API function returns one of approximately 70 different system values. You pass `w.GetSystemMetrics` a value that selects a particular system metric and the function returns this system value (this is very similar to the `w.GetDeviceCaps` function we looked at earlier in this chapter). These index values

2. See the Windows API documentation on the accompanying CD-ROM for details about the `w.GetScrollInfo` function. We will not discuss that function here.

generally have names that begin with w.SM_.... in the *windows.hhf* header file (the SM obviously stands for *system metric*). Here s the prototype for the w.GetSystemMetrics API function:

```
GetSystemMetrics: procedure
    (
        nIndex          :dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__GetSystemMetrics@4" );
```

We re not going to go into the details concerning what all the system metric values mean in this chapter. Fortunately, most of the values are fairly obvious from their corresponding constant (nIndex value) name. A few examples should give you the basic flavor:

¥ w.SM_CXBORDER - width, in pixels, of a window border.

¥ w.SM_CYBORDER - height, in pixels, of a window border.

¥ w.SM_CMOUSEBUTTONS - number of buttons on the mouse.

¥ w.SM_CXFULLSCREEN - width, in pixels, of the client area for a full-screen window on the primary display.

¥ w.SM_CYFULLSCREEN - height, in pixels, of the client area for a full-screen window on the primary display.

¥ w.SM_CXICON - the width, in pixels, of an icon.

¥ w.SM_CYICON - the height, in pixels, of an icon.

¥ w.SM_CXSCREEN - the width, in pixels, of the primary display.

¥ w.SM_CYSCREEN - the height, in pixels, of the primary display.

¥ etc. See the Microsoft documentation for all the possible constants and their meaning.

Rather than attempt to make individual calls to w.GetSystemMetrics for each value to output, the sysmets program executes a loop that retrieves the index value to submit to w.GetSystemMetrics from an array of records. Each element of that array has the following type:

```
MetricRec_t:
        record

            MetConst    :uns32;
            MetStr      :string;
            MetDesc     :string;

        endrecord;
```

The MetConst field holds a constant value like w.SM_CMOUSEBUTTONS that the program will pass on to w.GetSystemMetrics; this constant specifies the system metric to retrieve. The MetStr field is a string specifying the name of the constant (so we can display this constant name in the window). The MetDesc field is a brief English description of this particular system metric. The sysmet program statically initializes this array with the following values:

```
readonly

    MetricData: MetricRec_t[]  :=
    [
        MetricRec_t:[  w.SM_CXSCREEN, "w.SM_CXSCREEN", "Screen width" ],
        MetricRec_t:[  w.SM_CYSCREEN, "w.SM_CYSCREEN", "Screen height" ],
        MetricRec_t:[  w.SM_CXVSCROLL, "w.SM_CXVSCROLL", "Vert scroll arrow width" ],
        MetricRec_t:[  w.SM_CYVSCROLL, "w.SM_CYVSCROLL", "Vert scroll arrow ht" ],
        MetricRec_t:[  w.SM_CXHSCROLL, "w.SM_CXHSCROLL", "Horz scroll arrow width" ],
        MetricRec_t:[  w.SM_CYHSCROLL, "w.SM_CYHSCROLL", "Horz scroll arrow ht" ],
        MetricRec_t:[  w.SM_CYCAPTION, "w.SM_CYCAPTION", "Caption bar ht" ],
        MetricRec_t:[  w.SM_CXBORDER, "w.SM_CXBORDER", "Window border width" ],
        MetricRec_t:[  w.SM_CYBORDER, "w.SM_CYBORDER", "Window border height" ],
        MetricRec_t:[  w.SM_CXDLGFRAME, "w.SM_CXDLGFRAME", "Dialog frame width" ],
        MetricRec_t:[  w.SM_CYDLGFRAME, "w.SM_CYDLGFRAME", "Dialog frame height" ],
        MetricRec_t:[  w.SM_CXHTHUMB, "w.SM_CXHTHUMB", "Horz scroll thumb width" ],
        MetricRec_t:[  w.SM_CYVTHUMB, "w.SM_CYVTHUMB", "Vert scroll thumb width" ],
        MetricRec_t:[  w.SM_CXICON, "w.SM_CXICON", "Icon width" ],
        MetricRec_t:[  w.SM_CYICON, "w.SM_CYICON", "Icon height" ],
        MetricRec_t:[  w.SM_CXCURSOR, "w.SM_CXCURSOR", "Cursor width" ],
        MetricRec_t:[  w.SM_CYCURSOR, "w.SM_CYCURSOR", "Cursor height" ],
        MetricRec_t:[  w.SM_CYMENU,  "w.SM_CYMENU", "Menu bar height" ],
        MetricRec_t:[  w.SM_CXFULLSCREEN, "w.SM_CXFULLSCREEN", "Largest client width" ],
        MetricRec_t:[  w.SM_CYFULLSCREEN, "w.SM_CYFULLSCREEN", "Largets client ht" ],
        MetricRec_t:[  w.SM_DEBUG, "w.SM_CDEBUG", "Debug version flag" ],
        MetricRec_t:[  w.SM_SWAPBUTTON, "w.SM_CSWAPBUTTON", "Mouse buttons swapped" ],
        MetricRec_t:[  w.SM_CXMIN, "w.SM_CXMIN", "Minimum window width" ],
        MetricRec_t:[  w.SM_CYMIN, "w.SM_CYMIN", "Minimum window height" ],
        MetricRec_t:[  w.SM_CXSIZE, "w.SM_CXSIZE", "Minimize/maximize icon width" ],
        MetricRec_t:[  w.SM_CYSIZE, "w.SM_CYSIZE", "Minimize/maximize icon height" ],
        MetricRec_t:[  w.SM_CXFRAME, "w.SM_CXFRAME", "Window frame width" ],
        MetricRec_t:[  w.SM_CYFRAME, "w.SM_CYFRAME", "Window frame height" ],
        MetricRec_t:[  w.SM_CXMINTRACK,  "w.SM_CXMINTRACK", "Minimum tracking width" ],
        MetricRec_t:[  w.SM_CXMAXTRACK,  "w.SM_CXMAXTRACK", "Maximum tracking width" ],
        MetricRec_t:[  w.SM_CYMINTRACK,  "w.SM_CYMINTRACK", "Minimum tracking ht" ],
        MetricRec_t:[  w.SM_CYMAXTRACK,  "w.SM_CYMAXTRACK", "Maximum tracking ht" ],
        MetricRec_t:[  w.SM_CXDOUBLECLK, "w.SM_CXDOUBLECLK", "Dbl-click X tolerance" ],
        MetricRec_t:[  w.SM_CYDOUBLECLK, "w.SM_CYDOUBLECLK", "Dbl-click Y tolerance" ],
        MetricRec_t:[  w.SM_CXICONSPACING, "w.SM_CXICONSPACING", "Horz icon spacing" ],
        MetricRec_t:[  w.SM_CYICONSPACING, "w.SM_CYICONSPACING", "Vert icon spacing" ],
        MetricRec_t:[  w.SM_CMOUSEBUTTONS, "w.SM_CMOUSEBUTTONS", " # of mouse btns" ]
    ];

const
    NumMetrics := @elements( MetricData );
```

With this data structure in place, a simple for loop that executes `NumMetrics` times can sequence through each element of this array, pass the first value to `w.GetSystemMetrics`, and print the second two fields of each element along with the value that `w.GetSystemMetrics` returns.

In addition to the usual `w.WM_DESTROY`, `w.WM_PAINT`, and `w.WM_CREATE` messages we ve see in past programs, the *sysmet* application will need to process the `w.WM_HSCROLL`, `w.WM_VSCROLL`, and `w.WM_SIZE` (window resize) messages. Therefore, the `Dispatch` table will take the following form in *sysmet*:

```
readonly

    Dispatch    :MsgProcPtr_t; @nostorage;
```

```
        MsgProcPtr_t
            MsgProcPtr_t:[  w.WM_DESTROY, &QuitApplication   ],
            MsgProcPtr_t:[  w.WM_PAINT,   &Paint             ],
            MsgProcPtr_t:[  w.WM_CREATE,  &Create            ],
            MsgProcPtr_t:[  w.WM_HSCROLL, &HScroll           ],
            MsgProcPtr_t:[  w.WM_VSCROLL, &VScroll           ],
            MsgProcPtr_t:[  w.WM_SIZE,    &Size              ],

            // Insert new message handler records here.

            MsgProcPtr_t:[  0, NULL ];   // This marks the end of the list.
```

Naturally, we ll have to supply the corresponding `HScroll, VScroll`, and `Size` procedures as well as the window handling procedures we ve written in past applications. We ll return to the discussion of these procedures (as well as the other message handling procedures) momentarily.

In order to handle the actual scrolling, our application is going to need to know how many lines it can display in the window at one time and how many characters it can display on a single line in the current window (on the average, because their widths vary). These calculations depend upon the size (height) of the font we re using, the average character width, and the current size of the window. To facilitate these calculations, the program will store the average character height, average character width, and the average capital character width (which is wider than the average character width) in a set of global variables. Because this program only uses the system font (the default font when the program first begins execution) and never changes the font, the program only needs to calculate these values once. It calculates them in the `Create` procedure when Windows first creates the application s main window. These global variables are the following:

```
static

    AverageCapsWidth    :dword;
    AverageCharWidth    :dword;
    AverageCharHeight   :dword;
    MaxWidth            :int32 := 0;
```

The `Create` procedure (`w.WM_CREATE` message handling procedure) is responsible for initializing the values of these variables. The `Create` procedure calls `w.GetTextMetrics` to get the font s height and average character width. It computes the average capital character width as 1.5 times the average character width if using a proportional font, it simply copies the average character width to the average caps width if using a monospaced font. `MaxWidth` holds the maximum width of a line of text. The computation of this value is based on the assumption that the maximum `MetStr` field is 25 characters long and consists of all capital letters while the `MetDesc` field (and the corresponding value) is a maximum of 40 characters long (mixed case and digits). Hence, each line requires a maximum of `AverageCharWidth*40 + AverageCapsWidth*25` pixels. Here s the complete `Create` procedure that computes these values:

```
// Create-
//
//  This procedure responds to the w.WM_CREATE message.
//  Windows sends this message once when it creates the
//  main window for the application.  We will use this
//  procedure to do any one-time initialization that
//  must take place in a message handler.

procedure Create( hwnd: dword; wParam:dword; lParam:dword );
```

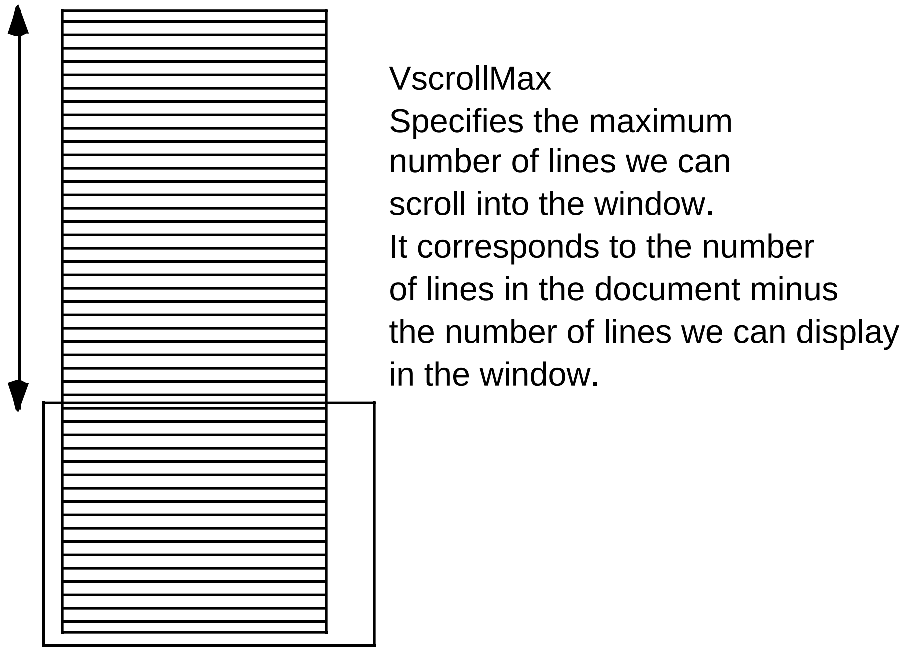
```
var
    hdc:    dword;              // Handle to video display device context
    tm:     w.TEXTMETRIC;
begin Create;

    GetDC( hwnd, hdc );

        // Initialization:
        //
        //  Get the text metric information so we can compute
        //  the average character heights and widths.

        GetTextMetrics( tm );

        mov( tm.tmHeight, eax );
        add( tm.tmExternalLeading, eax );
        mov( eax, AverageCharHeight );

        mov( tm.tmAveCharWidth, eax );
        mov( eax, AverageCharWidth );

        // If bit #0 of tm.tmPitchAndFamily is set, then
        // we've got a proportional font.  In that case
        // set the average capital width value to 1.5 times
        // the average character width.  If bit #0 is clear,
        // then we've got a fixed-pitch font and the average
        // capital letter width is equal to the average
        // character width.

        mov( eax, ebx );
        shl( 1, tm.tmPitchAndFamily );
        if( @c ) then

            shl( 1, ebx );              // 2*AverageCharWidth

        endif;
        add( ebx, eax );                // Computes 2 or 3 times eax.
        shr( 1, eax );                  // Computes 1 or 1.5 times eax.
        mov( eax, AverageCapsWidth );

    ReleaseDC;
    intmul( 40, AverageCharWidth, eax );
    intmul( 25, AverageCapsWidth, ecx );
    add( ecx, eax );
    mov( eax, MaxWidth );

end Create;
```

Whenever windows first creates a window, or whenever the user resizes the window, Windows will send a `w.WM_SIZE` message to the window procedure. Programs we ve written in the past have simply ignored this message. However, once you add scroll bars to your window you need to intercept this message so you can recompute the scroll range values and scroll bar thumb positions. This function computes four important values and saves two other important values. It saves the size of the client window passed to the procedure in H.O. and L.O. words of the `lParam` parameter in the `ClientSizeX` and `ClientSizeY` variables. It then computes the values for the current VscrollPos, VscrollMax, HscrollPos, and HscrollMax variables. These variables have the following declarations in the global data area:

```
static
    ClientSizeX          :int32 := 0;    // Size of the client area
    ClientSizeY          :int32 := 0;    //  where we can paint.

    VscrollPos           :int32 := 0;    // Tracks where we are in the document
    VscrollMax           :int32 := 0;    // Max display position (vertical).
    HscrollPos           :int32 := 0;    // Current Horz position.
    HscrollMax           :int32 := 0;    // Max Horz position.
```

VscrollMax specifies the maximum number of lines we can scroll through the window (see Figure 6-15). This is the number of lines in the document minus the number of lines we can actually display in the window (because we don t want to allow the user to scroll beyond the bottom of the document). In the actual computation in the Size procedure, we ll add two to this value to allow for some padding between the top of the client window and the first line as well as a blank line after the last line in the document.

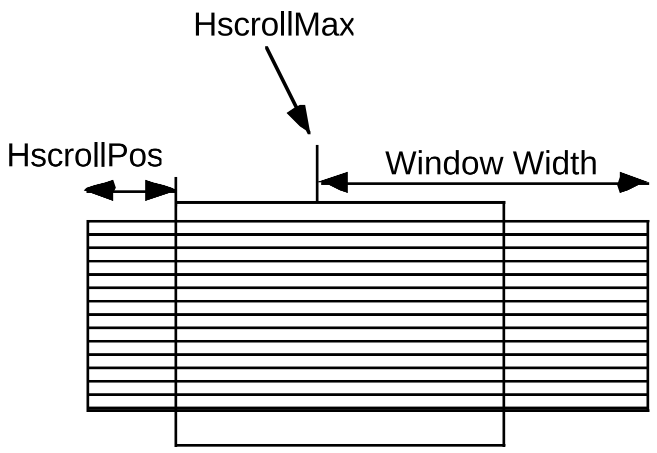
Figure 6-15: VscrollMax Value


VscrollMax
Specifies the maximum
number of lines we can
scroll into the window.
It corresponds to the number
of lines in the document minus
the number of lines we can display
in the window.

VscrollPos specifies the line number into the document that corresponds to the top line currently displayed in the window. Whenever the user resizes the window, we have to make sure that this value does not exceed the new VscrollMax position (that is, if the user makes the window larger and we re already displaying the text at the end of the document, we ll reduce VscrollPos to display more information towards the beginning of the document rather than more information towards the end of the document). The only time that *sysmet* will display a blank area beyond the end of the document is when the user opens up a window that is larger than the amount of data to display.

HscrollPos and HscrollMax are the corresponding values to VscrollMax and VscrollPos for the horizontal direction (see Figure 6-16). HscrollPos specifies how many characters into the text we ve scrolled off the left hand side of the window; HscrollMax specifies the maximum character position we re allowed to scroll (horizontally) to without blank data appearing on the right hand side of the window.

Figure 6-16: HScrollMax and HscrollPos Values



Here s the complete code for the `Size` procedure:

```
// Size-
//
//  This procedure handles the w.WM_SIZE message
//
//  L.O. word of lParam contains the new X Size
//  H.O. word of lParam contains the new Y Size

procedure Size( hwnd: dword; wParam:dword; lParam:dword );
begin Size;

    // Convert new X size to 32 bits and save:

    movzx( (type word lParam), eax );
    mov( eax, ClientSizeX );

    // Convert new Y size to 32 bits and save:

    movzx( (type word lParam[ 2]), eax );
    mov( eax, ClientSizeY );

    // VscrollMax = max( 0, NumMetrics+2 - ClientSizeY/AverageCharHeight )

    cdq();
    idiv( AverageCharHeight );
    mov( NumMetrics+2, ecx );
    sub( eax, ecx );
    if( @s ) then

        xor( ecx, ecx );

    endif;
    mov( ecx, VscrollMax );

    // VscrollPos = min( VscrollPos, VscrollMax )
    if( ecx > VscrollPos ) then

        mov( VscrollPos, ecx );

    endif;
    mov( ecx, VscrollPos );

    w.SetScrollRange( hwnd, w.SB_VERT, 0, VscrollMax, false );
    w.SetScrollPos( hwnd, w.SB_VERT, VscrollPos, true );

    // HscrollMax =
    //  max( 0, 2 + (MaxWidth - ClientSizeX) / AverageCharWidth);

    mov( MaxWidth, eax );
    sub( ClientSizeX, eax );
    cdq();
    idiv( AverageCharWidth );
    add( 2, eax );
    if( @s ) then

        xor( eax, eax );

    endif;
    mov( eax, HscrollMax );

    // HscrollPos = min( HscrollMax, HscrollPos )

    if( eax > HscrollPos ) then

        mov( HscrollPos, eax );

    endif;
    mov( eax, HscrollPos );
    w.SetScrollRange( hwnd, w.SB_HORZ, 0, HscrollMax, false );
    w.SetScrollPos( hwnd, w.SB_HORZ, HscrollPos, true );
    xor( eax, eax ); // return success.


end Size;
```

When Windows sends the application a `w.WM_HSCROLL` message, the L.O. word of the `wParam` parameter holds the type of scroll activity the user has requested. The `HScroll` procedure must interpret this value to determine how to adjust the horizontal position (`HscrollPos`). If the user presses on the left or right arrows on the horizontal scroll bar, then Windows will pass `w.SB_LINELEFT` or `w.SB_LINERIGHT` in the L.O. word of `wParam` and the `HScroll` procedure will scroll the window one (average) character width to the left or right, assuming such an operation would not take you outside the range 0..`Hscrollmax`.

If the user clicks on the scroll bar between the thumb and one of the arrows, then Windows will pass along the constant `w.SB_PAGELEFT` or `w.SB_PAGERIGHT` in the L.O. word of the `wParam` parameter. The *sysmet* application defines a page left or page right operation as a scroll eight average character positions in the appropriate direction. The choice of eight character positions was completely arbitrary. For text-based applications like sysmet, it doesn t make sense to scroll horizontally a whole screen at a time; the application achieves better continuity by scrolling only a few characters at a time. Typically, you d probably want to scroll some percentage of the window s width rather than a fixed amount (like eight character positions). Probably somewhere on the order of 25% to 50% of the window s width would be a decent amount to scroll. Such a modification to the *sysmet* program is rather trivial; feel free to do it as an experiment with this program.

The *sysmet* program ignores `w.SB_THUMBTRACK` messages and processes `w.SB_THUMBPOSITION` messages. For this particular application there is no reason we couldn t process `w.SB_THUMBTRACK` messages as well (and, in fact, the `VScroll` procedure does process those messages). The *sysmet* application only processes `w.SB_THUMBPOSITION` messages on the horizontal scroll bar and `w.SB_THUMBTRACK` messages on the vertical scroll bar so you can compare the feel of these two mechanisms.

The `HScroll` procedure adjusts the value of the global variable `HscrollPos` based upon the type of scrolling activity the user specifies. It also checks to make sure that the scroll position remains in the range 0..`Hscroll-Max`. Once these calculations are out of the way, the `HScroll` procedure calls the `w.ScrollWindow` API function. This function has the following prototype:

```
static
    ScrollWindow: procedure
    (
            hWnd        :dword;
            XAmount     :dword;
            YAmount     :dword;
        var lpRect      :RECT;
        var lpClipRect  :RECT
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__ScrollWindow@20" );
```

The hWnd parameter is the handle of the window whose client area you wish to scroll (which, of course, will be our application s main window). The `XAmount` and `YAmount` parameters specify how many pixels to scroll in the X and Y directions. Negative values scroll to the left or up, positive values scroll to the right or down. The `lpRect` parameter is the address of a `w.RECT` object that specifies the rectangle in the window to be scrolled. If this parameter contains NULL, then Windows scrolls the entire client area of the window. The `lpClipRect` specifies which pixels are to be repainted. If this parameter contains NULL, then Windows repaints all the pixels.

Once `HScroll` scrolls the window in the appropriate horizontal direction, it sets the new scroll position so that Windows will update the thumb position on the scroll bar. This is done with a call to w.SetScrollPos (discussed earlier).

Note that you do not repaint the window within the `HScroll` procedure. Remember, all window updates should take place only within the `Paint` procedure of the *sysmet* application. The call to `w.ScrollWindow` informs Windows that the client area of our application window is now invalid and should be repainted. This means that at some point in the future, Windows will be sending a `w.WM_PAINT` message to *sysmets* window procedure so it can redraw the window.

Here s the complete code for the `HScroll` procedure:

```
// HScroll-
//
//  Handles w.WM_HSCROLL messages.
//  On entry, L.O. word of wParam contains the scroll bar activity.

procedure HScroll( hwnd: dword; wParam:dword; lParam:dword );
begin HScroll;

    // Convert 16-bit value in wParam to 32 bits so we can use the
   // switch macro:

    movzx( (type word wParam), eax );
    switch( eax )

        case( w.SB_LINEUP )

            mov( -1, eax );

        case( w.SB_LINEDOWN )

            mov( 1, eax );

        case( w.SB_PAGEUP )

            mov( -8, eax );

        case( w.SB_PAGEDOWN )

            mov( 8, eax );

        case( w.SB_THUMBPOSITION )

            movzx( (type word wParam[ 2] ), eax );
            sub( HscrollPos, eax );

        default

            xor( eax, eax );

    endswitch;

    // eax =
    //  max( -HscrollPos, min( eax, HscrollMax - HscrollPos ))

    mov( HscrollPos, edx );
    neg( edx );
    mov( HscrollMax, ecx );
    add( edx, ecx );
    if( eax > (type int32 ecx) ) then

        mov( ecx, eax );

    endif;
    if( eax < (type int32 edx )) then

        mov( edx, eax );

    endif;
    if( eax <> 0 ) then

        add( eax, HscrollPos );
        imul( AverageCharWidth, eax );
        neg( eax );
        w.ScrollWindow( hwnd, eax, 0, NULL, NULL );
        w.SetScrollPos( hwnd, w.SB_HORZ, HscrollPos, true );
```

```
    endif;
    xor( eax, eax ); // return success

end HScroll;
```

The `VScroll` procedure is very similar in operation to the `HScroll` procedure; therefore, we don t need quite as detailed a discussion of its operation. However, there are a couple of important differences that we do need to discuss. As noted earlier, the `VScroll` procedure processes `w.WM_VSCROLL/w.SB_THUMBTRACK` messages rather than the `w.WM_VSCROLL/w.SB_THUMBPOSITION` messages that `HScroll` handles. There is one very big impact that this has on the execution of `VScroll` - `VScroll` cannot depend upon Windows issuing a `w.WM_PAINT` message in a timely manner. Windows `w.WM_PAINT` messages are very low priority and Windows holds them back in the message queue while your application processes other messages (e.g., the stream of `w.WM_VSCROLL/ w.SB_THUMBTRACK` messages that are screaming through the system). This may create a time lag between the movement of the scroll bar thumb and the corresponding update on the display, which is unacceptable. To overcome this problem, the `VScroll` procedure calls the `w.UpdateWindow` procedure. `w.UpdateWindow` tells Windows to immediately send a `w.WM_PAINT` message through the message queue (and make it high priority, so that it will be the next message that Windows sends to your window procedure). Therefore, when `w.UpdateWindows` returns, Windows will have already updated the display. This makes the behavior of the w.SB_THUMBTRACK request seem very fluid and efficient.

Here s the complete `VScroll` procedure:

```
// VScroll-
//
//  Handles the w.WM_VSCROLL messages from Windows.
//  The L.O. word of wParam contains the action/command to be taken.
//  The H.O. word of wParam contains a distance for the w.SB_THUMBTRACK
//  message.

procedure VScroll( hwnd: dword; wParam:dword; lParam:dword );
begin VScroll;

    movzx( (type word wParam), eax );
    switch( eax )

        case( w.SB_TOP )

            mov( VscrollPos, eax );
            neg( eax );

        case( w.SB_BOTTOM )

            mov( VscrollMax, eax );
            sub( VscrollPos, eax );

        case( w.SB_LINEUP )

            mov( -1, eax );

        case( w.SB_LINEDOWN )

            mov( 1, eax );
        case( w.SB_PAGEUP )

            mov( ClientSizeY, eax );
            cdq();
            idiv( AverageCharHeight );
            neg( eax );
            if( (type int32 eax) > -1 ) then

                mov( -1, eax );

            endif;

        case( w.SB_PAGEDOWN )

            mov( ClientSizeY, eax );
            cdq();
            idiv( AverageCharHeight );
            if( (type int32 eax) < 1 ) then

                mov( 1, eax );

            endif;

        case( w.SB_THUMBTRACK )

            movzx( (type word wParam[ 2]), eax );
            sub( VscrollPos, eax );

        default

            xor( eax, eax );

    endswitch;

    // eax = max( -VscrollPos, min( eax, VscrollMax - VscrollPos ))

    mov( VscrollPos, edx );
    neg( edx );
    mov( VscrollMax, ecx );
    add( edx, ecx );
    if( eax > (type int32 ecx) ) then

        mov( ecx, eax );

    endif;
    if( eax < (type int32 edx)) then

        mov( edx, eax );

    endif;

    if( eax <> 0 ) then

        add( eax, VscrollPos );
        intmul( AverageCharHeight, eax );
        neg( eax );
        w.ScrollWindow( hwnd, 0, eax, NULL, NULL );
```

```
        w.SetScrollPos( hwnd, w.SB_VERT, VscrollPos, true );
        w.UpdateWindow( hwnd );

    endif;
    xor( eax, eax ); // return success.


end VScroll;
```

The last procedure of interest is the `Paint` procedure, that actually draws the system metric information to the display. Technically, we could just set up the Windows paint structure so that the clipping rectangle (the area that Windows allows us to draw into) only includes the new region we want to draw, and then draw the entire document. Windows will clip (not draw) all data outside the clipping region, so this is a cheap way to do scrolling - just adjust the clipping region and then draw the entire document. Unfortunately, this scheme is too inefficient to even consider for most applications. Suppose you ve got a word processor application and the user has typed 100 pages into the word processor. Redrawing 100 pages every time the user scrolls one line (or worse yet, drags the thumb with `w.SB_THUMBTRACK` processing going on) would be incredibly slow. Therefore, the `Paint` procedure needs to be a little smarter about what it attempts to draw to the window.

The *sysmet* `Paint` procedure uses the `VscrollPos` variable to calculate the starting line to redraw in the window. In general, painting text is sufficiently fast on modern machines that this is all that would normally be necessary when painting the screen - just paint starting at `VscrollPos` for the number of lines of text that will fit in the window. When scrolling the entire window, you re going to wind up repainting the entire window anyway. However, there are many times when you don t actually need to repaint the entire window. For example, when a portion of *sysmet s* window is covered by some other window and the user closes that other window, Windows will only request that you redraw that portion of the client area that was originally covered by the closed window (that is, the invalid region). Although drawing text is relatively efficient, the *sysmet* application only redraws those lines of text that fall into the invalid region of the window. This will improve response time by a fair amount if painting the window is a complex operation. Once again, *sysmet s* painting isn t very complex but sysmet demonstrates this mechanism so you can see how to employ it in other applications.

As you may recall, the `w.BeginPaint` (i.e., the `BeginPaint` macro in *wpa.hhf*) has a parameter of type `w.PAINTSTRUCT` that windows initializes when you call `w.BeginPaint`. This object contains a field, `rcPaint`, of type `w.RECT`, that specifies the invalid region you must repaint. The *sysmet* `ps.rcPaint.top` and `ps.rcPaint.bottom` variables specify the starting vertical position in the client area and the ending vertical position in the client area that the `Paint` procedure must redraw. The `Paint` procedure divides these two values by the average character height to determine how many lines it can skip drawing at the top and bottom of the window. By combining the value of `VscrollPos` and `ps.rcPaint.top` the `Paint` procedure calculates `first-Met` - the index into the `MetricData` array where output is to begin. The `Paint` procedure uses a similar calculation based on `NumMetrics, ps.rcPaint.bottom`, and `VscrollPos` to determine the last line it will draw in the window from the `MetricData` array.

Here is the complete *sysmet* program, including the `Paint` procedure we ve just discussed:

```
// Sysmet.hla-
//
//  System metrics display program.

program systemMetrics;
#include( "conv.hhf" )
#include( "strings.hhf" )
#include( "memory.hhf" )
#include( "hll.hhf" )
#include( "w.hhf" )
```

```
#include( "wpa.hhf" )

?NoDisplay := true;
?NoStackAlign := true;


type
    // Data type for the system metrics data array:

    MetricRec_t:
        record

            MetConst    :uns32;
            MetStr      :string;
            MetDesc     :string;

        endrecord;


    // Message and dispatch table related definitions:

    MsgProc_t:  procedure( hwnd:dword; wParam:dword; lParam:dword );

    MsgProcPtr_t:
        record

            MessageValue    :dword;
            MessageHndlr    :MsgProc_t;

        endrecord;




static
    hInstance           :dword;         // "Instance Handle" Windows supplies.

    wc                  :w.WNDCLASSEX;  // Our "window class" data.
    msg                 :w.MSG;         // Windows messages go here.
    hwnd                :dword;         // Handle to our window.

    AverageCapsWidth    :dword;         // Font metric values.
    AverageCharWidth    :dword;
    AverageCharHeight   :dword;

    ClientSizeX         :int32 := 0;    // Size of the client area
    ClientSizeY         :int32 := 0;    //  where we can paint.
    MaxWidth            :int32 := 0;    // Maximum output width
    VscrollPos          :int32 := 0;    // Tracks where we are in the document
    VscrollMax          :int32 := 0;    // Max display position (vertical).
    HscrollPos          :int32 := 0;    // Current Horz position.
    HscrollMax          :int32 := 0;    // Max Horz position.


readonly

    ClassName   :string := "SMWinClass";                // Window Class Name
    AppCaption  :string := "System Metrics Program";    // Caption for Window


    // The dispatch table:
    //
    //  This table is where you add new messages and message handlers
    //  to the program.  Each entry in the table must be a MsgProcPtr_t
    //  record containing two entries: the message value (a constant,
    //  typically one of the w.WM_***** constants found in windows.hhf)
    //  and a pointer to a "MsgProcPtr_t" procedure that will handle the
    //  message.


    Dispatch    :MsgProcPtr_t; @nostorage;

        MsgProcPtr_t
            MsgProcPtr_t:[  w.WM_DESTROY, &QuitApplication   ],
            MsgProcPtr_t:[  w.WM_PAINT,   &Paint             ],
            MsgProcPtr_t:[  w.WM_CREATE,  &Create            ],
            MsgProcPtr_t:[  w.WM_HSCROLL, &HScroll           ],
            MsgProcPtr_t:[  w.WM_VSCROLL, &VScroll           ],
            MsgProcPtr_t:[  w.WM_SIZE,    &Size              ],

            // Insert new message handler records here.

            MsgProcPtr_t:[  0, NULL ];   // This marks the end of the list.



readonly

    MetricData: MetricRec_t[] :=
    [
        MetricRec_t:[  w.SM_CXSCREEN, "w.SM_CXSCREEN", "Screen width" ],
        MetricRec_t:[  w.SM_CYSCREEN, "w.SM_CYSCREEN", "Screen height" ],
        MetricRec_t:[  w.SM_CXVSCROLL, "w.SM_CXVSCROLL", "Vert scroll arrow width" ],
        MetricRec_t:[  w.SM_CYVSCROLL, "w.SM_CYVSCROLL", "Vert scroll arrow ht" ],
        MetricRec_t:[  w.SM_CXHSCROLL, "w.SM_CXHSCROLL", "Horz scroll arrow width" ],
        MetricRec_t:[  w.SM_CYHSCROLL, "w.SM_CYHSCROLL", "Horz scroll arrow ht" ],
        MetricRec_t:[  w.SM_CYCAPTION, "w.SM_CYCAPTION", "Caption bar ht" ],
        MetricRec_t:[  w.SM_CXBORDER, "w.SM_CXBORDER", "Window border width" ],
        MetricRec_t:[  w.SM_CYBORDER, "w.SM_CYBORDER", "Window border height" ],
        MetricRec_t:[  w.SM_CXDLGFRAME, "w.SM_CXDLGFRAME", "Dialog frame width" ],
        MetricRec_t:[  w.SM_CYDLGFRAME, "w.SM_CYDLGFRAME", "Dialog frame height" ],
        MetricRec_t:[  w.SM_CXHTHUMB, "w.SM_CXHTHUMB", "Horz scroll thumb width" ],
        MetricRec_t:[  w.SM_CYVTHUMB, "w.SM_CYVTHUMB", "Vert scroll thumb width" ],
        MetricRec_t:[  w.SM_CXICON, "w.SM_CXICON", "Icon width" ],
        MetricRec_t:[  w.SM_CYICON, "w.SM_CYICON", "Icon height" ],
        MetricRec_t:[  w.SM_CXCURSOR, "w.SM_CXCURSOR", "Cursor width" ],
        MetricRec_t:[  w.SM_CYCURSOR, "w.SM_CYCURSOR", "Cursor height" ],
        MetricRec_t:[  w.SM_CYMENU,  "w.SM_CYMENU", "Menu bar height" ],
        MetricRec_t:[  w.SM_CXFULLSCREEN, "w.SM_CXFULLSCREEN", "Largest client width" ],
        MetricRec_t:[  w.SM_CYFULLSCREEN, "w.SM_CYFULLSCREEN", "Largets client ht" ],
        MetricRec_t:[  w.SM_DEBUG, "w.SM_CDEBUG", "Debug version flag" ],
        MetricRec_t:[  w.SM_SWAPBUTTON, "w.SM_CSWAPBUTTON", "Mouse buttons swapped" ],
        MetricRec_t:[  w.SM_CXMIN, "w.SM_CXMIN", "Minimum window width" ],
        MetricRec_t:[  w.SM_CYMIN, "w.SM_CYMIN", "Minimum window height" ],
        MetricRec_t:[  w.SM_CXSIZE, "w.SM_CXSIZE", "Minimize/maximize icon width" ],
        MetricRec_t:[  w.SM_CYSIZE, "w.SM_CYSIZE", "Minimize/maximize icon height" ],
        MetricRec_t:[  w.SM_CXFRAME, "w.SM_CXFRAME", "Window frame width" ],
        MetricRec_t:[  w.SM_CYFRAME, "w.SM_CYFRAME", "Window frame height" ],
        MetricRec_t:[  w.SM_CXMINTRACK,  "w.SM_CXMINTRACK", "Minimum tracking width" ],
        MetricRec_t:[  w.SM_CXMAXTRACK,  "w.SM_CXMAXTRACK", "Maximum tracking width" ],
        MetricRec_t:[  w.SM_CYMINTRACK,  "w.SM_CYMINTRACK", "Minimum tracking ht" ],
        MetricRec_t:[  w.SM_CYMAXTRACK,  "w.SM_CYMAXTRACK", "Maximum tracking ht" ],
        MetricRec_t:[  w.SM_CXDOUBLECLK, "w.SM_CXDOUBLECLK", "Dbl-click X tolerance" ],
        MetricRec_t:[  w.SM_CYDOUBLECLK, "w.SM_CYDOUBLECLK", "Dbl-click Y tolerance" ],
        MetricRec_t:[  w.SM_CXICONSPACING, "w.SM_CXICONSPACING", "Horz icon spacing" ],
        MetricRec_t:[  w.SM_CYICONSPACING, "w.SM_CYICONSPACING", "Vert icon spacing" ],
        MetricRec_t:[  w.SM_CMOUSEBUTTONS, "w.SM_CMOUSEBUTTONS", " # of mouse btns" ]
    ];

const
    NumMetrics := @elements( MetricData );


/**************************************************************************/
/*          A P P L I C A T I O N   S P E C I F I C   C O D E             */
/**************************************************************************/

// QuitApplication:
//
//  This procedure handles the w.WM_DESTROY message.
//  It tells the application to terminate.  This code sends
//  the appropriate message to the main program's message loop
//  that will cause the application to terminate.

procedure QuitApplication( hwnd: dword; wParam:dword; lParam:dword );
begin QuitApplication;

    w.PostQuitMessage( 0 );

end QuitApplication;


// Create-
//
//  This procedure responds to the w.WM_CREATE message.
//  Windows sends this message once when it creates the
//  main window for the application.  We will use this
//  procedure to do any one-time initialization that
//  must take place in a message handler.

procedure Create( hwnd: dword; wParam:dword; lParam:dword );
var
    hdc:    dword;              // Handle to video display device context
    tm:     w.TEXTMETRIC;
begin Create;

    GetDC( hwnd, hdc );

        // Initialization:
        //
        //  Get the text metric information so we can compute
        //  the average character heights and widths.

        GetTextMetrics( tm );

        mov( tm.tmHeight, eax );
        add( tm.tmExternalLeading, eax );
        mov( eax, AverageCharHeight );

        mov( tm.tmAveCharWidth, eax );
        mov( eax, AverageCharWidth );

        // If bit #0 of tm.tmPitchAndFamily is set, then
        // we've got a proportional font.  In that case
        // set the average capital width value to 1.5 times
        // the average character width.  If bit #0 is clear,
        // then we've got a fixed-pitch font and the average
        // capital letter width is equal to the average
        // character width.

        mov( eax, ebx );
        shl( 1, tm.tmPitchAndFamily );
        if( @c ) then

            shl( 1, ebx );              // 2*AverageCharWidth

        endif;
        add( ebx, eax );                // Computes 2 or 3 times eax.
        shr( 1, eax );                  // Computes 1 or 1.5 times eax.
        mov( eax, AverageCapsWidth );

    ReleaseDC;
    intmul( 40, AverageCharWidth, eax );
    intmul( 25, AverageCapsWidth, ecx );
    add( ecx, eax );
    mov( eax, MaxWidth );

end Create;


// Paint:
//
//  This procedure handles the w.WM_PAINT message.
//  For this System Metrics program, the Paint procedure
//  displays three columns of text in the main window.
//  This procedure computes and displays the appropriate text.

procedure Paint( hwnd: dword; wParam:dword; lParam:dword );
var
    x           :int32;         // x-coordinate of start of output str.
    y           :int32;         // y-coordinate of start of output str.

    CurVar      :string;        // Current system metrics variable name.
    CVlen       :uns32;         // Length of CurVar string.

    CurDesc     :string;        // Current system metrics description.
    CDlen       :string;        // Length of the above.
    CDx         :int32;         // X position for CurDesc string.
    value       :string;
    valData     :char[ 32];
    CVx         :int32;          // X position for value string.
    vallen      :uns32;          // Length of value string.

    firstMet    :int32;          // Starting metric to begin drawing
    lastMet     :int32;          // Ending metric index to draw.

    hdc         :dword;          // Handle to video display device context
    ps          :w.PAINTSTRUCT; // Used while painting text.

begin Paint;

    // Message handlers must preserve EBX, ESI, and EDI.
    // (They've also got to preserve EBP, but HLA's procedure
    // entry code already does that.)

    push( ebx );
    push( esi );
    push( edi );


    // Initialize the value->valData string object:

    mov( str.init( (type char valData), 32 ), value );

    // When Windows requests that we draw the window,
    // fill in the string in the center of the screen.
    // Note that all GDI calls (e.g., w.DrawText) must
    // appear within a BeginPaint..EndPaint pair.

    BeginPaint( hwnd, ps, hdc );

        // Figure out which metric we should start drawing
        // (firstMet =
        //      max( 0, VscrollPos + ps.rcPaint.top/AverageCharHeight - 1)):

        mov( ps.rcPaint.top, eax );
        cdq();
        idiv( AverageCharHeight );
        add( VscrollPos, eax );
        dec( eax );
        if( (type int32 eax) < 0 ) then

            xor( eax, eax );

        endif;
        mov( eax, firstMet );


        // Figure out the last metric we should be drawing
        // ( lastMet =
        //      min( NumMetrics,
        //           VscrollPos + ps.rcPaint.bottom/AverageCharHeight )):

        mov( ps.rcPaint.bottom, eax );
        cdq();
        idiv( AverageCharHeight );
        add( VscrollPos, eax );
        if( (type int32 eax) > NumMetrics ) then

            mov( NumMetrics, eax );

        endif;
        mov( eax, lastMet );


        // The following loop processes each entry in the
        // MetricData array.  The loop control variable (EDI)
        // also determines the Y-coordinate where this code
        // will display each line of text in the window.
        // Note that this loop counts on the fact that Windows
        // API calls preserve the EDI register.

        for( mov( firstMet, edi ); edi < lastMet; inc( edi )) do

            // Before making any Windows API calls (which have
            // a nasty habit of wiping out registers), compute
            // all the values we will need for these calls
            // and save those values in local variables.
            //
            //  A typical "high level language solution" would
            // be to compute these values as needed, immediately
            // before each Windows API calls.  By moving this
            // code here, we can take advantage of values previously
            // computed in registers without having to worry about
            // Windows wiping out the values in those registers.

            // Compute index into MetricData:

            intmul( @size( MetricRec_t ), edi, esi );

            // Grab the string from the current MetricData element:

            mov( MetricData.MetStr[ esi ], eax );
            mov( eax, CurVar );
            mov( (type str.strRec [eax]).length, eax );
            mov( eax, CVlen );

            mov( MetricData.MetDesc[ esi ], eax );
            mov( eax, CurDesc );
            mov( (type str.strRec [eax]).length, eax );
            mov( eax, CDlen );

            // Column one begins at X-position AverageCharWidth (ACW).
            // Col 2 begins at ACW + 25*AverageCapsWidth.
            // Col 3 begins at ACW + 25*AverageCapsWidth + 40*ACW.
            // Compute the Col 2 and Col 3 values here.

            mov( 1, eax );
            sub( HscrollPos, eax );
            intmul( AverageCharWidth, eax );
            mov( eax, x );
            intmul( 25, AverageCapsWidth, eax );
            add( x, eax );
            mov( eax, CDx );

            intmul( 40, AverageCharWidth, ecx );
            add( ecx, eax );
            mov( eax, CVx );

            // The Y-coordinate for the line of text we're writing
            // is computed as AverageCharHeight * (1-VscrollPos+edi).
            // Compute that value here:

            mov( 1, eax );
            sub( VscrollPos, eax );
            add( edi, eax );
            intmul( AverageCharHeight, eax );
            mov( eax, y );


            // Now generate the string we're going to print
            // as the value for the current metric variable:

            w.GetSystemMetrics( MetricData.MetConst[ esi ] );
            conv.i32ToStr( eax, 0, ' ', value );
            mov( str.length( value ), vallen );


            // First two columns have left-aligned text:

            SetTextAlign( w.TA_LEFT | w.TA_TOP );

            // Output the name of the metric variable:

            TextOut( x, y, CurVar, CVlen );

            // Output the description of the metric variable:

            TextOut( CDx, y, CurDesc, CDlen );

            // Output the metric's value in the third column.  This is
            // a numeric value, so we'll right align this data.

            SetTextAlign( w.TA_RIGHT | w.TA_TOP );
            TextOut( CVx, y, value, vallen );

            // Although not strictly necessary for this program,
            // it's a good idea to always restore the alignment
            // back to the default (top/left) after you done using
            // some other alignment.

            SetTextAlign( w.TA_LEFT | w.TA_TOP );


        endfor;
    EndPaint;

    pop( edi );
    pop( esi );
    pop( ebx );

end Paint;



// Size-
//
//  This procedure handles the w.WM_SIZE message
//
//  L.O. word of lParam contains the X Size
//  H.O. word of lParam contains the Y Size

procedure Size( hwnd: dword; wParam:dword; lParam:dword );
begin Size;

    movzx( (type word lParam), eax );
    mov( eax, ClientSizeX );

    movzx( (type word lParam[ 2]), eax );
    mov( eax, ClientSizeY );

    // VscrollMax = max( 0, NumMetrics+2 - ClientSizeY/AverageCharHeight )

    cdq();
    idiv( AverageCharHeight );
    mov( NumMetrics+2, ecx );
    sub( eax, ecx );
    if( @s ) then

        xor( ecx, ecx );

    endif;
    mov( ecx, VscrollMax );

    // VscrollPos = min( VscrollPos, VscrollMax )

    if( ecx > VscrollPos ) then

        mov( VscrollPos, ecx );

    endif;
    mov( ecx, VscrollPos );

    w.SetScrollRange( hwnd, w.SB_VERT, 0, VscrollMax, false );
    w.SetScrollPos( hwnd, w.SB_VERT, VscrollPos, true );

    // HscrollMax =
    //  max( 0, 2 + (MaxWidth - ClientSizeX) / AverageCharWidth);

    mov( MaxWidth, eax );
    sub( ClientSizeX, eax );
    cdq();
    idiv( AverageCharWidth );
    add( 2, eax );
    if( @s ) then

        xor( eax, eax );

    endif;
    mov( eax, HscrollMax );

    // HscrollPos = min( HscrollMax, HscrollPos )

    if( eax > HscrollPos ) then

        mov( HscrollPos, eax );

    endif;
    mov( eax, HscrollPos );
    w.SetScrollRange( hwnd, w.SB_HORZ, 0, HscrollMax, false );
    w.SetScrollPos( hwnd, w.SB_HORZ, HscrollPos, true );
    xor( eax, eax ); // return success.


end Size;



// HScroll-
//
//  Handles w.WM_HSCROLL messages.
//  On entry, L.O. word of wParam contains the scroll bar activity.

procedure HScroll( hwnd: dword; wParam:dword; lParam:dword );
begin HScroll;

    // Convert 16-bit value in wParam to 32 bits so we can use the
   // switch macro:

    movzx( (type word wParam), eax );
    switch( eax )

        case( w.SB_LINEUP )

            mov( -1, eax );

        case( w.SB_LINEDOWN )

            mov( 1, eax );

        case( w.SB_PAGEUP )

            mov( -8, eax );

        case( w.SB_PAGEDOWN )

            mov( 8, eax );

        case( w.SB_THUMBPOSITION )
```

```
            movzx( (type word wParam[ 2] ), eax );
            sub( HscrollPos, eax );

        default

            xor( eax, eax );

    endswitch;

    // eax =
    //  max( -HscrollPos, min( eax, HscrollMax - HscrollPos ))

    mov( HscrollPos, edx );
    neg( edx );
    mov( HscrollMax, ecx );
    add( edx, ecx );
    if( eax > (type int32 ecx) ) then

        mov( ecx, eax );

    endif;
    if( eax < (type int32 edx )) then

        mov( edx, eax );

    endif;
    if( eax <> 0 ) then

        add( eax, HscrollPos );
        imul( AverageCharWidth, eax );
        neg( eax );
        w.ScrollWindow( hwnd, eax, 0, NULL, NULL );
        w.SetScrollPos( hwnd, w.SB_HORZ, HscrollPos, true );

    endif;
    xor( eax, eax ); // return success

end HScroll;



// VScroll-
//
//  Handles the w.WM_VSCROLL messages from Windows.
//  The L.O. word of wParam contains the action/command to be taken.
//  The H.O. word of wParam contains a distance for the w.SB_THUMBTRACK
//  message.

procedure VScroll( hwnd: dword; wParam:dword; lParam:dword );
begin VScroll;

    movzx( (type word wParam), eax );
    switch( eax )

        case( w.SB_TOP )
            mov( VscrollPos, eax );
            neg( eax );

        case( w.SB_BOTTOM )

            mov( VscrollMax, eax );
            sub( VscrollPos, eax );

        case( w.SB_LINEUP )

            mov( -1, eax );

        case( w.SB_LINEDOWN )

            mov( 1, eax );

        case( w.SB_PAGEUP )

            mov( ClientSizeY, eax );
            cdq();
            idiv( AverageCharHeight );
            neg( eax );
            if( (type int32 eax) > -1 ) then

                mov( -1, eax );

            endif;

        case( w.SB_PAGEDOWN )

            mov( ClientSizeY, eax );
            cdq();
            idiv( AverageCharHeight );
            if( (type int32 eax) < 1 ) then

                mov( 1, eax );

            endif;

        case( w.SB_THUMBTRACK )

            movzx( (type word wParam[2]), eax );
            sub( VscrollPos, eax );

        default

            xor( eax, eax );

    endswitch;

    // eax = max( -VscrollPos, min( eax, VscrollMax - VscrollPos ))

    mov( VscrollPos, edx );
    neg( edx );
    mov( VscrollMax, ecx );
    add( edx, ecx );
    if( eax > (type int32 ecx) ) then
```

```
        mov( ecx, eax );

    endif;
    if( eax < (type int32 edx)) then

        mov( edx, eax );

    endif;

    if( eax <> 0 ) then

        add( eax, VscrollPos );
        intmul( AverageCharHeight, eax );
        neg( eax );
        w.ScrollWindow( hwnd, 0, eax, NULL, NULL );
        w.SetScrollPos( hwnd, w.SB_VERT, VscrollPos, true );
        w.UpdateWindow( hwnd );

    endif;
    xor( eax, eax ); // return success.


end VScroll;



/**************************************************************************/
/*                   End of Application Specific Code                    */
/**************************************************************************/




// The window procedure.
// 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( hwnd:dword; uMsg:uns32; wParam:dword; lParam:dword );
    @stdcall;

begin WndProc;

    // uMsg contains the current message Windows is passing along to
    // us.  Scan through the "Dispatch" table searching for a handler
    // for this message.  If we find one, then call the associated
    // handler procedure.  If we don't have a specific handler for this
    // message, then call the default window procedure handler function.

    mov( uMsg, eax );
    mov( &Dispatch, edx );
    forever

        mov( (type MsgProcPtr_t [edx]).MessageHndlr, ecx );
        if( ecx = 0 ) then
```

```
            // If an unhandled message comes 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.

            w.DefWindowProc( hwnd, uMsg, wParam, lParam );
            exit WndProc;


        elseif( eax = (type MsgProcPtr_t [ edx] ).MessageValue ) then

            // If the current message matches one of the values
            // in the message dispatch table, then call the
            // appropriate routine.  Note that the routine address
            // is still in ECX from the test above.

            push( hwnd );   // (type tMsgProc ecx)(hwnd, wParam, lParam)
            push( wParam ); //  This calls the associated routine after
            push( lParam ); //  pushing the necessary parameters.
            call( ecx );

            sub( eax, eax ); // Return value for function is zero.
            break;

        endif;
        add( @size( MsgProcPtr_t ), edx );

    endfor;

end WndProc;



// Here's the main program for the application.

begin systemMetrics;

    // Get this process' handle:

    w.GetModuleHandle( NULL );
    mov( eax, hInstance );


    // Set up the window class (wc) object:

    mov( @size( w.WNDCLASSEX ), wc.cbSize );
    mov( w.CS_HREDRAW | w.CS_VREDRAW, wc.style );
    mov( &WndProc, wc.lpfnWndProc );
    mov( NULL, wc.cbClsExtra );
    mov( NULL, wc.cbWndExtra );
    mov( w.COLOR_WINDOW+1, wc.hbrBackground );
    mov( NULL, wc.lpszMenuName );
    mov( ClassName, wc.lpszClassName );
    mov( hInstance, wc.hInstance );

    // Get the icons and cursor for this application:
```

```
w.LoadIcon( NULL, val w.IDI_APPLICATION );
mov( eax, wc.hIcon );
mov( eax, wc.hIconSm );

w.LoadCursor( NULL, val w.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.

w.RegisterClassEx( wc );

w.CreateWindowEx
(
    NULL,
    ClassName,
    AppCaption,
    w.WS_OVERLAPPEDWINDOW | w.WS_VSCROLL |  w.WS_HSCROLL,
    w.CW_USEDEFAULT,
    w.CW_USEDEFAULT,
    w.CW_USEDEFAULT,
    w.CW_USEDEFAULT,
    NULL,
    NULL,
    hInstance,
    NULL
);
mov( eax, hwnd );

w.ShowWindow( hwnd, w.SW_SHOWNORMAL );
w.UpdateWindow( hwnd );

// Here's the event loop that processes messages
// sent to our window.  On return from GetMessage,
// break if EAX contains false and then quit the
// program.

forever

    w.GetMessage( msg, NULL, 0, 0 );
    breakif( !eax );
    w.TranslateMessage( msg );
    w.DispatchMessage( msg );

endfor;

// The message handling inside Windows has stored
// the program's return code in the wParam field
// of the message.  Extract this and return it
// as the program's return code.

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

```
end systemMetrics;
```

6.6: The DebugWindow Application

When moving from console applications to GUI applications, an important feature is lost to the application programmer - the ability to print a string of text from an arbitrary point in the program that display s the program s status and other debugging information. In this section we will explore how to write a simple terminal emulator application that displays text much like console applications process text. This simple application will demonstrate how to create console-like applications within a GUI environment. One feature of this particular application, however, is that it displays textual data sent to it from other applications. This allows you to write a GUI application that can send text messages to this simple console emulator application, allowing you to display debug (and other status messages) from a GUI application that doesn t normally support console-like output.

Writing a *DebugWindow* application that accepts messages from other applications makes use of some advanced *message passing* features in Windows. Few books on Windows programming would present this topic so early in the book. This book makes an exception to this rule for a couple of reasons. First, though the whole concept of message passing and multitasking applications is a bit advanced, you don t have to learn everything there is to know about this subject area to create the *DebugWindow* application. This section will only present a few of the API calls that you will need to implement this advanced form of message passing, so this shouldn t be a problem. Another reason for presenting this application here is that *DebugWindow* is, intrinsically, a text-based application. So describing the operation of this application in a chapter on text processing in Windows seems like a natural fit. Finally, one really good reason for presenting this application here is because you ll find it extremely useful and the sooner you have this application available, the sooner you ll be able to employ it in your own Windows projects. So the sooner the better...

6.6.1: Message Passing Under Windows

Windows communicates with your applications by passing them *messages*. Your window procedure is the code that Windows calls when it sends your application a message. Window procedures in code we ve written up to this point have processed messages like `w.WM_PAINT`, `w.WM_CREATE`, and `w.WM_DESTROY` that Windows has passed to these applications. Although Windows is the most common source of messages to your applications, it s also possible for your application to send messages to itself or send messages to other processes in the system. For example, it s perfectly possible for you to send a `w.WM_PAINT` message to your program to force the window to repaint itself. Indeed, it s reasonable to manually send almost any message Windows sends to your window procedure. Messages provide a form of deferred procedure call that you can use to call the message handling procedures that your window procedure calls.

In addition to the stock messages that Windows defines, Windows also predefines a couple of sets of user-definable messages. These user-definable messages let your application create its own set of procedures that can be called via the message passing mechanism. Windows defines three ranges of user-definable messages. The first set of user-definable messages have values in the range `w.WM_USER`..$7FFF. Windows reserves messages in this range for private use by a window procedure. Note, however, that Window messages in this range are only meaningful within a single window class, they are not unique throughout an application. For example, some stock controls that Windows provides will use message values in this range. However, if you are sending a message to a specific window (which is a member of some specific window class), then you can define private messages that your window procedure will respond to using the values in this range.

If you need a range of messages that are unique throughout an application (e.g., you re going to broadcast a message to all active windows within a given application), then you ll want to use the message values in the range w.WM_APP..$BFFF. Windows guarantees that no system messages use these values, so if you broadcast a message whose message number is in this range, then none of the Windows system window procedures (e.g., button, text edit boxes, list boxes, etc.) will inadvertently respond to messages in this range.

If two different processes want to pass messages between themselves, allocating hard-coded message numbers is not a good idea (because it s possible that some third process could accidentally intercept such messages if it also uses the same message number for inter-process communication). Therefore, Windows defines a final range of message values ($C000..$FFFF) that Windows explicitly manages via the w.RegisterWindowMessage API call. Here s the prototype for this API function:

```
static
    RegisterWindowMessage: procedure
    (
        lpString        :string
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__RegisterWindowMessageA@4" );
```

The parameter is an HLA string that uniquely identifies the message value to create. Windows will search for this string in an internal database and return whatever value is associated with that string. If no such string exists internal to Windows, then Windows will allocate a unique message number in the range $C000..$FFFF and return that value (w.RegisterWindowMessage returns the message value in the EAX register). The message number remains active until all programs that have registered it terminate execution. Note that unlike the other message values, that you can represent via constants, you must store message values that w.RegisterWindow-Message returns in a variable so you can test for these message values in your window procedure. The coding examples up to this point have always put their dispatch table in a readonly section. If you want to be able to process a message number that you obtain from w.RegisterWindowMessage, you ll have to put the dispatch table in a static section or modify the window procedure to test the message number that w.RegisterWindowMessage returns outside the normal loop that it uses to process the dispatch table entries. E.g.,

```
static

    Dispatch:   MsgProcPtr_t; @nostorage;

        MsgProcPtr_t
            MsgProcPtr_t:[  w.WM_DESTROY, &QuitApplication   ],
            MsgProcPtr_t:[  w.WM_CREATE,  &Create            ],
            MsgProcPtr_t:[  w.WM_PAINT,   &Paint             ],

    MyMsg: MsgProcPtr_t; @nostorage;
            MsgProcPtr_t:[  -1, &MyMsgHandler ],  // -1 is a dummy value.

            // Insert new message handler records here.

            MsgProcPtr_t:[  0, NULL ];   // This marks the end of the list.


      .
      .
      .
    // Get a unique message value from Windows and overwrite the dummy "-1" value
    // we're currently using as MyMsg's message value:
```

```
    w.RegisterWindowMessage( “MyMsg_Message_Value” );
    mov( eax, MyMsg.MessageValue );
```

Sending a message is really nothing more than a fancy way of calling a window procedure. Windows provides many different ways to send a message to a window procedure, but a very common API function that does this is `w.SendMessage`:

```
static
    SendMessage: procedure
    (
        hWnd            :dword;
        _Msg            :dword;
        _wParam         :dword;
        _lParam         :dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__SendMessageA@16" );
```

The `hWnd` parameter specifies the handle of the window whose window procedure will receive the message. The `_Msg` parameter specifies the value of the message you want to pass to the window procedure (e.g., `w.WM_PAINT` or `w.WM_USER`). The `_wParam` and `_lParam` parameters are arbitrary 32-bit values that Windows passes on through to the window procedure; these parameters usually supply parameter data for the specific message. As you ll recall, the declaration of a window procedure takes the following form:

procedure WndProc(hwnd:dword; uMsg:uns32; wParam:dword; lParam:dword);

As you can probably tell, Windows just passes the four `w.SendMessage` parameters on through to the appropriate Window procedure.

When an application is send a message to its own window procedure, knowing the window procedures handle is no big deal; Windows returns this handle when the application calls `w.RegisterWindowEx` from the main program. The application can save this value and use it when calling the window procedure via `w.SendMessage`. However, if an application wants to send a message to some other process in the system, the application will need to know the handle value for that other process window procedure. To obtain the window handle for a window belonging to some other process in the system, an application can call the Windows `w.FindWindow` function:

```
static
    FindWindow: procedure
    (
        lpClassName     :string;
        lpWindowName    :string
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__FindWindowA@8" );
```

The `lpClassName` parameter is a string containing the window procedure s class name, that is, the string that was registered with Windows in the `w.RegisterClassEx` call[3]. If you only have one instance of a given window

3. Windows also allows you to pass a small integer value, known as an *atom*, as this parameter s value. We will not consider atoms here.

class active in the system, then you can pass NULL as the value of the second parameter. However, if you ve instantiated several different copies of a window, you can use the second parameter (which is the window title string) to further differentiate the windows. If you re going to be passing messages between different processes in the system, it s a real good idea to create a unique window class for the receiving process and only run one instance of that class at a time. That s the approach this book will take; doing so allows us to simply pass NULL as the value of the second parameter to `w.FindWindow`.

The `w.FindWindow` function returns the handle of the window you specify in the EAX register, assuming that window is presently active in the system. If the window class you specify is not available in the system, then `w.FindWindow` will return NULL (zero) in the EAX register. You can use this NULL return value to determine that there is no process waiting to receive the messages you want to send it, and take corrective action, as appropriate.

As a short example, suppose that you have the System Metrics program from the previous section running on your system. You can send this application a message from a different process and tell it to resize its window using a code sequence like the following:

```
    w.FindWindow( “SMWinClass”, NULL );  // Get Sysmet’s window handle in eax.
    w.SendMessage( eax, w.WM_SIZE, 0, 480 << 16 + 640 );  // 640X480 resize operation.
```

The `w.SendMessage` API function is *synchronous*. This means that it doesn t return to the caller until the window procedure it invokes finishes whatever it does with the message and returns control back to Windows. The only time `w.SendMessage` returns without the other process completing its task is when you pass `w.SendMessage` an illegal handle number. For example, if the process containing the window procedure you re invoking has quit (or was not executing in the first place), then the handle you re passing `w.SendMessage` is invalid. Otherwise, it s up to the message destination s window procedure to retrieve and process the message before whomever calls `w.SendMessage` will continue execution. If the target of the message never retrieves the message, or hangs up while processing the message, then the application that sent the message will hang up as well.

Synchronous behavior is perfect semantics for an intra-process message (that is, a message that an application sends to itself). This is very similar to a procedure call, which most programmers are comfortable using. Unfortunately, this behavior is not entirely appropriate for inter-process calls because the activities of that other process (e.g., whether or not that other process has hung up) have a direct impact on the execution of the current process. To write more robust message passing applications, we need to relax the semantics of a synchronous call somewhat to avoid problems.

Windows provides several different API functions you can call to send a message through the message-handling system. Some of these additional functions address the problems with `w.SendMessage` hanging up if the receiving window procedure doesn t properly process the message. Examples of these API functions include `w.PostMessage`, `w.SendMessageCallback`, and `w.SendMessageTimeout`. These particular API functions have the following prototypes:

```
static
    PostMessage: procedure
    (
        hWnd            :dword;
        _Msg            :dword;
        wParam          :dword;
        lParam          :dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__PostMessageA@16" );
    SendMessageCallback: procedure
    (
        hWnd            :dword;
        _Msg            :dword;
        _wParam         :dword;
        _lParam         :dword;
        lpCallBack      :SENDASYNCPROC;
        dwData          :dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__SendMessageCallbackA@24" );

    SendMessageTimeout: procedure
    (
            hWnd        :dword;
            _Msg        :dword;
            _wParam     :dword;
            _lParam     :dword;
            fuFlags     :dword;
            uTimeout    :dword;
        var lpdwResult  :dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__SendMessageTimeoutA@28" );
```

The `w.PostMessage` function sends a message to some window procedure and immediately returns to the caller without waiting for the receiver to process the message. The parameters are identical to those for `w.SendMessage`. Although `w.PostMessage` avoids the problem with an application hanging up because the recipient of the message doesn t respond, there is a major drawback to using `w.PostMessage` - the caller doesn t know whether the target window procedure has actually received and processed the message. Therefore, if the caller requires some sort of proof of completion of the message, `w.PostMessage` is not an appropriate function to call. On the other hand, if the loss of the message (or the activity that the message causes) is no big deal, then `w.PostMessage` is a safe way to communicate with another window procedure[4].

To receive confirmation that the target window procedure has processed (or has not processed) a message, without hanging up the calling process, Windows provides the `w.SendMessageCallback` function and the `w.SendMessageTimeout` function. These two functions take completely different approaches to dealing with the problems of synchronous message passing.

The `w.SendMessageCallback` API function behaves like `w.PostMessage` insofar as it immediately returns after sending the message to the target window procedure (that is, it doesn t wait for the window procedure to actually process the message). To receive notification that message processing has taken place, you pass `w.SendMessageCallback` the address of a *callback function*. Windows will call this function when the target window procedure completes execution. In the meantime, your application has continued execution. Because the call to the callback function is asynchronous, there are many issues involving concurrent programming that you must consider when using this technique. The issues of concurrent programming are a bit beyond the scope of this chapter, so we ll ignore this API function for the time being. See the Windows documentation for more details concerning the use of this function.

4. Actually, we ll see another big problem with `w.PostMessage` in the next section. But for most purposes, `w.PostMessage` is just fine if the application calling w.PostMessage can live with lost messages.

The second mechanism is to use a timeout to return control back to whomever calls `w.SendMessageTimeout` if the send message operation is taking too long (implying that the target window procedure is hung or is otherwise not going to process the message we sent it). Under normal circumstances, `w.SendMessageTimeout` behaves just like `w.SendMessage` - that is, the caller blocks (halts) until the recipient of the message processes the message and returns. The `w.SendMessageTimeout` function, however, will go ahead and return if the message s receiver does not respond within some time period (that you specify).

The `w.SendMessageTimeout` API function has three additional parameters (above and beyond those you supply to `w.SendMessage`): `fuFlags, uTimeout`, and `lpdwResult`. The `fuFlags` parameter should be one of the following values:

¥ `w.SMTO_ABORTIFHUNG` - This tells `w.SendMessageTimeout` to immediately return (with a timeout error) if the receiving process appears to be hung. To Windows, a process is hung if it has not attempted to read a message from its message queue during the past five seconds.

¥ `w.SMTO_BLOCK` - This value tells `w.SendMessageTimeout` to wait until the message returns (or times out). Windows will not send any other messages to the current process window procedure when you use this value.

¥ `w.SMTO_NORMAL` - The caller can process other messages that are sent to the window procedure while waiting for the `w.SendMessageTimeout` call to return.

¥ `w.SMTO_NOTIMEOUTIFNOTHUNG` - (Windows 2000 and later) Does not return when the time-out period expires if the process receiving the message does not appear to be hung (that is, it is still processing messages on a regular basis).

Most of the time you ll want to pass `w.SMTO_BLOCK` or `w.SMTO_NORMAL` as the `fuFlags` parameter value.

The `uTimeout` parameter specifies the timeout period in milliseconds. Choosing a time-out period can be tricky. Too long a value may cause your application to pause for extended periods of time while waiting for a hung receiver, to short a timeout period may cause Windows to prematurely return a time-out error to your program because the message s receiver hasn t gotten around to processing the message. Because Windows defines a hung program as one that doesn t respond to any messages within five seconds, a five second timeout period is probably a good starting point unless you need much faster response from `w.SendMessageTimeout`. Presumably, if the system is functioning correctly and the target process is running, you should get a response within a couple of milliseconds.

6.6.2: Memory Protection and Messages

A Windows message provides only a 64-bit data payload (i.e., the values in the `wParam` and `lParam` parameters). If someone needs to pass more data via a message, the standard convention is to pass a pointer to a data structure containing that data through the `lParam` parameter. The window procedure then casts the `lParam` parameter as a pointer to that data structure and accesses the data indirectly. By limiting the data payload to 64 bits, Windows keeps the message passing mechanism lean and efficient. Unfortunately, this scheme creates some problems when we want to pass data from one process to another via a message.

The problem with passing information between two processes under (modern versions of) Windows is that each process runs in a separate *address space*. This means that Windows sets aside a four-gigabyte memory range for each running program that is independent of the four-gigabyte address space for other executing programs. The benefit of this scheme is that one process cannot inadvertently modify code or data in another process. The drawback is that two processes cannot easily share data in memory by passing pointers between themselves. Address $40_1200 in one process will not reference the same data as address $40_1200 in a second process. Therefore, passing data directly through memory from one process to another isn t going to work.

In order to copy a large amount of data (that is, anything beyond the 64-bit data payload that wParam and lParam provide), Windows has to make a copy of the data in the source process address space in the address space of the destination process. For messages that Windows knows about (e.g., a w.WM_GETTEXT message that returns text associated with certain controls, like a text edit box), Windows handles all the details of copying data to and from buffers (typically pointed at by lParam) between two different processes. For user-defined messages, however, Windows has no clue whether the bit pattern in the lParam parameter to w.SendMessage is a pointer to some block of data in memory or a simple integer value. To overcome this problem, Windows provides a special message specifically geared towards passing a block of data via the w.SendMessage API function: the w.WM_COPYDATA message. A call to w.SendMessage passing the w.WM_COPYDATA message value takes the following form:

```
w.SendMessage
(
   hDestWindow,   // Handle of destination window
   w.WM_COPYDATA, // The copy data message command
   hWnd,          // Handle of the window passing the data
   cds            // (address of) a w.COPYDATASTRUCT object
);
```

The w.COPYDATASTRUCT data structure takes the following form:

```
type
   COPYDATASTRUCT:
      record
         dwData    :dword;  // generic 32-bit value passed to receiver.
         cbData    :uns32;  // Number of bytes in block to transfer.
         lpData    :dword;  // Pointer to block of data to transfer.
      endrecord;
```

The dwData field is a generic 32-bit value that Windows will pass along to the application receiving the block of data. The primary purpose of this field is to pass along a user-defined message number (or command) to the receiver. Windows only provides a single w.WM_COPYDATA message; if you need to send multiple messages with large data payloads to some other process, you will have to send each of these messages using the single w.WM_COPYDATA message. To differentiate these messages, you can use the cds.dwData field to hold the actual message type. If your receiving application only needs to process a single user-defined message involving a block of data, it can switch off the w.WM_COPYDATA message directly and use the dwData field for any purpose it chooses.

The cbData field in the w.COPYDATASTRUCT data type specifies the size of the data block that is being transferred from the source application to the destination window. If you re passing a zero-terminated string from the source to the target application, don t forget to include the zero terminating byte in your length (i.e., don t simply pass the string length as the cbData value). If you re passing some other data structure, just take the size of that structure and pass it in the cbData field.

The lpData field in the w.COPYDATASTRUCT object is a pointer to the block of data to copy between the applications. This block of data must not contain any pointers to objects within the source application s address space (even if those pointers reference other objects in the data block you re copying). There is no guarantee that Windows will copy the block of data to the same address in the target address space and Windows will only copy data that is part of the data block (it will not copy data that pointers within the block reference, unless that data is also within the block). Therefore, any pointers appearing in the lpData area will be invalid once Windows copies this data into the address space of the target window procedure.

Given the complexity and overhead associated with passing a block of data from one application to another via `w.SendMessage`, you might question why we would want to use this scheme to create the *DebugWindow* application. After all, it s perfectly possible to open up a second window in an existing application and send all of your debugging output to that window. However, the problem with sending debug messages to an application s own debug window is that if the program crashes or hangs, that application s debug window may be destroyed (so you can t see the last output messages sent to the display just prior to the crash) or the program may freeze (so you can scroll through the messages sent to the debug window). Therefore, it s much better to have all the debug messages sent to a separate application whose browsing ability isn t tied to the execution of the application you re testing.

6.6.3: Coding the *DebugWindow* Application

Like any system involving inter-process communication, the *DebugWindow* application is one of two (or more) applications that have to work together. *DebugWindow* is responsible for displaying debug and status messages that other applications send. However, for *DebugWindow* to actually do something useful, you ve actually got to have some other application send a debug message to the *DebugWindow* program. In this section we ll discuss the receiver end of this partnership (that is, the *DebugWindow* application), in the next section we ll discuss the transmitter (that is, the modifications necessary to applications that transmit debug messages).

The *DebugWindow* application itself will be a relatively straight-forward *dumb terminal emulation*. It will display text sent to it and it will buffer up some number of lines of text allowing the user to review those lines. By default, the version of *DebugWindow* we will develop in this section will save up to 1,024 lines of text, each line up to 255 characters long (though these values are easy enough to change in the source code). *DebugWindow* will make use of the vertical and horizontal scroll bars to allow the user to view all lines and columns of text displayable in the window. Once the transmitting application(s) send more than the maximum number of lines of text, *DebugWindow* will begin throwing away the oldest lines of text in the system. This keeps the program from chewing up valuable system resources (memory) if an application produces a large amount of output. The number of lines of text that *DebugWindow* remembers is controlled by the `MaxLines_c` constant appearing at the beginning of the source file:

```
const
MaxLines_c  := 1024;                // Maximum number of lines we will
                                    //  save up for review.
```

If you want to allow fewer or more lines of saved text, feel free to change this value.

DebugWindow will be capable of processing a small set of control characters embedded in the strings sent to it. Specifically, it will handle *carriage returns, line feeds*, and *tab* characters as special characters. All other characters it will physically write to the display window. Though it s fairly easy to extend *DebugWindow* to handle other control characters, there really is no need to do so. Most other control characters that have a widely respected meaning (e.g., backspace and formfeed) have a destructive nature and we don t want garbage output to the debugging window to accidentally erase any existing data (that could be used to determine why we got the garbage output).

DebugWindow ignores carriage returns and treats linefeeds as a new line character. Normal console applications under Windows use carriage return to move the cursor to the beginning of the current line. However, keeping in mind that we don t want *DebugWindow* to wipe out any existing data, we ll choose to quietly ignore carriage returns that appear in the character stream. The reason for considering carriage returns at all is because HLA s `nl` constant (newline) is the character sequence <carriage return><line feed> so it s quite likely that carriage returns will appear in the debug output because programmers are used to using the `nl` constant. Whenever a linefeed comes along, *DebugWindow* will emit the current line of text to the next output line in the output window and move the cursor to the beginning of the next line in the output window (*DebugWindow* will also save the line in an internal memory buffer so the user can review the line later, using the vertical scroll bar).

DebugWindow will interpret tab characters to adjust the output cursor so that it moves to the next tab stop on the output line. By default, *DebugWindow* sets tab stops every eight character positions. In order for tab positions to make any sense at all, *DebugWindow* uses a monospaced font (Fixedsys) for character output. If a tab character appears in the data stream, *DebugWindow* moves the cursor to the next column that is an even multiple of eight, plus one. I.e., columns 9, 17, 25, 33, 41, etc., are tab positions. A tab character moves the cursor (blank filling) to the closest tab stop whose column number is greater than the current column position. *DebugWindow* uses a default of eight-column tab positions (this is a standard output for terminal output). You may, however, change this by modifying the `tabCols` constant at the beginning of the source file:

```
const
    tabCols     := 8;                   // Columns per tabstop position.
```

Like many of the applications you ve seen in this chapter, *DebugWindow* uses the `w.WM_CREATE` message to determine when it can do program initialization of various global objects. The initialization that the Create procedure handles includes the following:

¥ Initialization of the array of strings that maintain the output lines for later review by the user (i.e., sets all of these strings to the NULL pointer).

¥ Selection of the Fixedsys font and determining font metrics so the application knows the height and width of characters in this font (so *DebugWindows* can determine how many characters will fit in the output window as well as deal with vertical and horizontal scrolling).

¥ Output of an initial debug message to inform the user that*DebugWindows* is active.

Here s the code for the Create procedure that handles these tasks:

```
// Create-
//
//  This procedure responds to the w.WM_CREATE message.
//  Windows sends this message once when it creates the
//  main window for the application.  We will use this
//  procedure to do any one-time initialization that
//  must take place in a message handler.

procedure Create( hwnd: dword; wParam:dword; lParam:dword );
readonly
    s           :string :=
                    "Debug Windows begin execution:" nl nl;

    FixedsysName:byte; @nostorage;
                 byte "Fixedsys", 0;

static
    hDC         :dword;
    hOldFont    :dword;
    Fixedsys    :w.LOGFONT;
    tm          :w.TEXTMETRIC;
    cds         :w.COPYDATASTRUCT;

begin Create;
```

```
push( edi );


// Zero out our line index, count, and the array of pointers:

xor( eax, eax );
mov( eax, StartLine );
mov( eax, LineCnt );
mov( MaxLines_c, ecx );
mov( &Lines, edi );
rep.stosd();

// Zero out the FONT structure:

mov( @size( w.LOGFONT ), ecx );
mov( &Fixedsys, edi );
rep.stosb();

// Create the font using the system's "Fixedsys" font and select
// it into the device context:

str.zcpy( FixedsysName, Fixedsys.lfFaceName );
w.CreateFontIndirect( Fixedsys );
mov( eax, FixedsysFontHandle );     // Save, so we can free this later.
GetDC( hwnd, hDC );

    // Select in the fixed system font:

    SelectObject( FixedsysFontHandle );
    mov( eax, hOldFont );

    // Determine the sizes we need for the fixed system font:

    GetTextMetrics( tm );


    mov( tm.tmHeight, eax );
    add( tm.tmExternalLeading, eax );
    inc( eax );
    mov( eax, CharHeight );

    mov( tm.tmAveCharWidth, eax );
    mov( eax, CharWidth );

    SelectObject( hOldFont );

ReleaseDC;

// Just for fun, let's send ourselves a message to print the
// first line in the debug window:

mov( DebugMsg_c, cds.dwData );
mov( s, eax );
mov( eax, cds.lpData );
mov( (type str.strRec [eax]).length, eax );
inc( eax ); // Count zero byte, too!
```

```
    mov( eax, cds.cbData );
    w.SendMessage( hwnd, w.WM_COPYDATA, hwnd, &cds );

    pop( edi );

end Create;
```

The *DebugWindow* application will process several Windows messages and one user message. Here s the dispatch table for *DebugWindow*:

```
static
    Dispatch    :MsgProcPtr_t; @nostorage;

        MsgProcPtr_t
            MsgProcPtr_t:[  w.WM_PAINT,       &Paint              ],
            MsgProcPtr_t:[  w.WM_COPYDATA,    &RcvMsg             ],
            MsgProcPtr_t:[  w.WM_CREATE,      &Create             ],
            MsgProcPtr_t:[  w.WM_HSCROLL,     &HScroll            ],
            MsgProcPtr_t:[  w.WM_VSCROLL,     &VScroll            ],
            MsgProcPtr_t:[  w.WM_SIZE,        &Size               ],
            MsgProcPtr_t:[  w.WM_DESTROY,     &QuitApplication    ],

            // Insert new message handler records here.

            MsgProcPtr_t:[  0, NULL ];   // This marks the end of the list.
```

The user message will actually be the `w.WM_COPYDATA` system message. This is because the one user-defined message that *DebugWindow* accepts is a print message that contains a zero-terminated string to display in the debug output window. The `RcvMsg` procedure (which handles the `w.WM_COPYDATA` message) defines the three `w.COPYDATASTRUCT` fields as follows:

¥ `dwData` - This field contains the four bytes dbug (d is in the H.O. byte). *DebugWindow* only processes a single user message, so there is no need to use this field to differentiate messages. However, *DebugWindow* does check the value of this field just as a sanity check to avoid display garbage data.

¥ `cbData` - This field contains the length of the string data (including the zero byte). *DebugWindow* actually ignores this information and simply processes the string data until it encounters a zero terminating byte.

¥ `lpData` - This is a pointer to a zero-terminated string (note: this is *not* an HLA string!) that contains the data to display in the output window.

Note that the string at which `lpData` points isn t necessarily a single line of text. This string may contain multiple new line sequences (<carriage return><line feed>) that result in the display of several lines of text in the output window. Therefore, *DebugWindow* cannot simply copy this string to an element of the array of strings that maintain the lines of text in the display window. Instead, the `RcvMsg` procedure has to process each character in the message sent to *DebugWindows* and deal with the control characters (tabs, carriage returns, and line feeds) and construct one or more strings from the message of the text to place in the internal string list. Here s the `RcvMsg` procedure that does this processing:

```
// RcvMsg-
//
//  Receives a message from some other process and prints
// the zstring passed as the data payload for that message.
// Note that lParam points at a w.COPYDATASTRUCT object. The
// dwData field of that structure must contain DebugMsg_c if
// we are to process this message.

procedure RcvMsg( hwnd: dword; wParam:dword; lParam:dword );
var
    tabPosn :uns32;
    line    :char[ 256];

   // Note: addLine procedure actually goes here...

begin RcvMsg;

    push( esi );
    push( edi );

    // Process the incoming zero-terminated string.
    // Break it into separate lines (based on newline
    // sequences found in the string) and expand tabs.

    mov( lParam, esi );
    if( (type w.COPYDATASTRUCT [ esi] ).dwData = DebugMsg_c ) then

        // Okay, we've got the w.COPYDATASTRUCT type with a valid debug
        // message. Extract the data and print it.

        mov( (type w.COPYDATASTRUCT [ esi] ).lpData, esi );
        mov( 0, tabPosn );
        lea( edi, line );
        while( (type char [ esi] ) <> #0 ) do

            mov( [ esi] , al );

            // Ignore carriage returns:

            if( al <> stdio.cr ) then

                if( al = stdio.tab ) then

                    // Compute the number of spaces until the
                    // next tab stop position:

                    mov( tabPosn, eax );
                    cdq();
                    div( tabCols, edx:eax );
                    neg( edx );
                    add( tabCols, edx );

                    // Emit spaces up to the next tab stop:

                    repeat

                        mov( ' ', (type char [ edi] ) );
                        inc( edi );
                        inc( tabPosn );
                        if( tabPosn >= 255 ) then

                            dec( edi );
                        endif;
                        dec( edx );

                    until( @z );

                elseif( al = stdio.lf ) then

                    // We've just hit a new line character.
                    // Emit the line up to this point:

                    mov( #0, (type char [edi]) ); // Zero terminate.
                    lea( edi, line );       // Resets edi for next loop iteration.
                    str.a_cpyz( [edi] );    // Build HLA string from zstring.
                    addLine( eax );         // Add to our list of lines.

                    // Reset the column counter back to zero since
                    // we're starting a new line:

                    mov( 0, tabPosn );

                else

                    // If it's not a special control character we process,
                    // then add the character to the string:

                    mov( al, [edi] );
                    inc( edi );
                    inc( tabPosn );

                    // Don't allow more than 255 characters:

                    if( tabPosn >= 255 ) then

                        dec( edi );

                    endif;

                endif;

            endif;

            // Move on to next character in source string:

            inc( esi );

        endwhile;

        // If edi is not pointing at the beginning of "line", then we've
        // got some characters in the "line" string that need to be added
        // to our list of lines.

        lea( esi, line );
        if( edi <> esi ) then

            mov( #0, (type char [edi]) );   // Zero terminate the string.
            str.a_cpyz( [esi] );            // Make a copy of this zstring.
```

```
                addLine( eax );

            endif;

            // Because adding and removing lines can affect the
            // maximum line length, recompute the width and horizontal
            // scroll bar stuff here:

            ComputeWidth( hwnd );

            // Ditto for the vertical scroll bars:

            ComputeHeight( hwnd );

            // Tell Windows to tell us to repaint the screen:

            w.InvalidateRect( hwnd, NULL, true );
            w.UpdateWindow( hwnd );

        endif;

        pop( edi );
        pop( esi );

    end RcvMsg;
```

This procedure begins by checking the `dwData` field of the `w.COPYDATASTRUCT` parameter passed by reference via the `lParam` parameter. If this field contains the constant `DebugMsg_c` (dbug) then this procedure continues processing the data, otherwise it completely ignores the message (i.e., this is the sanity check to see if the message is valid).

If the message passes the sanity check, then the `RcvMsg` procedure begins processing the characters from the message one at a time, copying these characters to the local `line` character array if the characters aren t carriage returns, tabs, line feeds, or the zero terminating byte. If the individual character turns out to be a tab character, then `RcvMsg` expands the tab by writing the appropriate number of spaces to the `line` array. If it s a carriage return, then `RcvMsg` quietly ignores the character. If it s a line feed, then `RcvMsg` converts the characters in the `line` array to an HLA string and calls its local procedure `addLine` to add the current line of text to the list of lines that *DebugWindow* displays. If it s any other character besides a zero-terminating byte, then `RcvMsg` just appends the character to the end of the `line` array.

When `RcvMsg` encounters a zero-terminating byte it checks to see if there are any characters in the line array. If so, it converts those characters to an HLA string and adds them to the list of strings that *DebugWindow* will display. If there are no (more) characters in the input string, then RcvMsg doesn t bother creating another string. This process ensures that we don t lose any strings that consist of a sequence of characters ending with a new line sequence (on the other hand, it also means that `RcvMsg` automatically appends a new line sequence to the end of the message text if it doesn t end with this character sequence).

After adding the string(s) to the string list, the `RcvMsg` procedure calls the ComputeWidth and ComputeHeight functions (see the source code a little later for details). These functions compute the new maximum width of the file (the width of the list of strings could have gotten wider or narrowing depending on the lines we ve added or delete) and the new height information (because we ve added lines). These functions also update the horizontal and vertical scroll bars in an appropriate fashion.

The last thing that RcvMsg does is call the w.InvalidateRect and w.UpdateWindow API functions to force Windows to send a w.WM_PAINT message to the application. This causes *DebugWindow* to redraw the screen and display the text appearing in the debug window.

The RcvMsg procedure has a short local procedure (addLine) that adds HLA strings to the list of strings it maintains. To understand how this procedure operates, we need to take a quick look at the data structures that *DebugWindows* uses to keep track of the strings it has displayed. Here are the pertinent variables:

```
static
    LineAtTopOfScrn     :uns32 := 0;    // Tracks where we are in the document
    DisplayLines        :uns32 := 2;    // # of lines we can display.

    StartLine   :uns32;                 // Starting index into "Lines" array.
    LineCnt     :uns32;                 // Number of valid lines in "Lines".
    Lines       :string[ MaxLines_c] ;  // Holds the text sent to us.
```

The fundamental data structure that keeps track of the text in the debug output window is an array of strings named Lines. This array holds up to MaxLines_c strings. Once this array fills up, *DisplayWindows* reuses the oldest strings in the array to hold incoming data. The StartLine and LineCnt variables maintain this circular queue of strings. LineCnt specifies how many strings are present in the Lines array. This variable starts with the value zero when *DebugWindow* first runs and increments by one with each incoming line of text until it reaches MaxLines_c. Once LineCnt reaches MaxLines_c the *DebugWindows* program stops adding new lines to the array and it starts reusing the existing entries in the lines array. The StartLine variable is what *DebugWindows* uses to maintain the circular buffer of strings. StartLine specifies the first array element of Lines to begin drawing in the debug window (assuming you re scrolled all the way to the top of the document). This variable will contain zero as long as there are fewer than MaxLines_c lines in the array. Once LineCnt hits MaxLines_c, however, *DisplayWindow* stops adding new lines to the Lines array and it no longer increments the LineCnt variable. Instead, it will first free the storage associated with the string at Lines[StartLine], it will store the new (incoming) string into Lines[StartLine], and then it will increment StartLine by one (wrapping back to zero whenever StartLine reaches MaxLines_c). This has the effect of reusing the oldest line still in the Lines array and setting the second oldest line to become the new oldest line.

The LineAtTopOfScrn is an integer index into the Lines array that specifies the line that the Paint procedure will begin drawing at the top of the window. DisplayLines is a variable that holds the maximum number of lines that the program can display in the window at one time. Once the value of LineCnt exceeds the value of DisplayLines, the program will increment LineAtTopOfScrn with each incoming line of text so that the screen will scroll up with each incoming line (as you d normally expect for a terminal).

In addition to these variables, there are a complementary set that control horizontal scrolling. The addLine procedure, however, doesn t use them so we ll ignore them for now.

Here s the actual addLine procedure (which is local to the RcvMsg procedure):

```
    procedure addLine( lineToAdd:string in eax ); @nodisplay; @noframe;
    begin addLine;

        if( LineCnt >= MaxLines_c ) then

            mov( StartLine, ecx );
            strfree( Lines[  ecx*4 ] ); // Free the oldest line of text.
            inc( StartLine );
            if( StartLine >= MaxLines_c ) then

                mov( 0, StartLine );
            endif;

        else

            mov( LineCnt, ecx );
            inc( LineCnt );

        endif;
        mov( eax, Lines[ ecx*4 ] );

        // If we've got more than "DisplayLines" lines in the
        // output, bump "LineAtTopOfScrn" to scroll the window up:

        mov( LineCnt, ecx );
        if( ecx >= DisplayLines ) then

            inc( LineAtTopOfScrn );

        endif;
        ret();

    end addLine;
```

The DebugWindow application also processes `w.WM_SIZE`, `w.WM_VSCROLL`, and `w.WM_HSCROLL` messages. However, the logic their corresponding message handling procedures use is nearly identical to that used by the *Sysmets* application given earlier in this chapter, so we won t rehash the description of these routines. The major differences between *Sysmets* and *DebugWindow* are mainly in the horizontal scrolling code. DebugWindow uses a monospaced font, so it can more easily handle horizontal scrolling as a function of some number of characters. Another difference is that horizontal scrolling in *DebugWindows* scrolls $^1/_4$ of the screen when you do a page left or page right operation.

The `Paint` procedure in *DebugWindow* is also fairly straight-forward. A global variable (`ColAtLeftOfScrn`) tracks the amount of horizontal scrolling that takes place; *DebugWindow* uses this as an index into each line of text that it displays.

Here s the complete code to the *DebugWindow* application:

```
// DebugWindow.hla-
//
//  This program accepts "Debug Print" statements from other processes
// and displays that data in a "console-like" text window.

program DebugWindow;
#include( "stdio.hhf" )
#include( "conv.hhf" )
#include( "strings.hhf" )
#include( "memory.hhf" )
#include( "hll.hhf" )
#include( "w.hhf" )
#include( "wpa.hhf" )
#include( "excepts.hhf" )

?@NoDisplay := true;
?@NoStackAlign := true;
```

```
const
    tabCols     := 8;                   // Columns per tabstop position.
    MaxLines_c  := 1024;                // Maximum number of lines we will
                                        //  save up for review.



static
    hInstance           :dword;         // "Instance Handle" Windows supplies.

    wc                  :w.WNDCLASSEX;  // Our "window class" data.
    msg                 :w.MSG;         // Windows messages go here.
    hwnd                :dword;         // Handle to our window.
    FixedsysFontHandle  :dword;         // Save this so we can free it later.

    WM_DebugPrint       :dword;         // Message number sent to us.

    CharWidth           :dword;
    CharHeight          :dword;

    ClientSizeX         :int32 := 0;    // Size of the client area
    ClientSizeY         :int32 := 0;    //  where we can paint.

    LineAtTopOfScrn     :uns32 := 0;    // Tracks where we are in the document
    MaxLnAtTOS          :uns32 := 0;    // Max display position (vertical).
    DisplayLines        :uns32 := 2;    // # of lines we can display.

    ColAtLeftOfScrn     :uns32 := 0;    // Current Horz position.
    MaxColAtLeft        :uns32 := 0;    // Max Horz position.
    MaxWidth            :uns32 := 40;   // Maximum columns seen thus far


    StartLine   :uns32;                 // Starting index into "Lines" array.
    LineCnt     :uns32;                 // Number of valid lines in "Lines".
    Lines       :string[ MaxLines_c] ;  // Holds the text sent to us.


// The following data type and DATA declaration
// defines the message handlers for this program.

type
    MsgProc_t:  procedure( hwnd:dword; wParam:dword; lParam:dword );

    MsgProcPtr_t:
        record

            MessageValue:   dword;
            MessageHndlr:   MsgProc_t;

        endrecord;


readonly

    ClassName   :string := "DebugWindowClass";  // Window Class Name
    AppCaption  :string := "Debug Window";      // Caption for Window
    // The dispatch table:
    //
    //  This table is where you add new messages and message handlers
    //  to the program.  Each entry in the table must be a MsgProcPtr_t
    //  record containing two entries: the message value (a constant,
    //  typically one of the w.WM_***** constants found in windows.hhf)
    //  and a pointer to a "MsgProcPtr_t" procedure that will handle the
    //  message.
    //
    // Note: the RcvMsg entry *must* be first as the Create code will
    // patch the entry for the message number (this is not a constant,
    // the message number gets assigned by the system).  The current
    // message number value for this entry is bogus; it must not, however,
    // be zero.


static
    Dispatch    :MsgProcPtr_t; @nostorage;

        MsgProcPtr_t
            MsgProcPtr_t:[  w.WM_PAINT,      &Paint              ],
            MsgProcPtr_t:[  w.WM_COPYDATA,   &RcvMsg             ],
            MsgProcPtr_t:[  w.WM_CREATE,     &Create             ],
            MsgProcPtr_t:[  w.WM_HSCROLL,    &HScroll            ],
            MsgProcPtr_t:[  w.WM_VSCROLL,    &VScroll            ],
            MsgProcPtr_t:[  w.WM_SIZE,       &Size               ],
            MsgProcPtr_t:[  w.WM_DESTROY,    &QuitApplication    ],

            // Insert new message handler records here.

            MsgProcPtr_t:[  0, NULL ];   // This marks the end of the list.




/**************************************************************************/
/*          A P P L I C A T I O N   S P E C I F I C   C O D E            */
/**************************************************************************/

// ComputeWidth-
//
//  This procedure scans through all the lines of text we've saved
// up and finds the longest line in the list.  From this, it computes
// the maximum width we have and sets up the horizontal scroll bar
// accordingly.  This function also sets up the global variables
// MaxWidth,
// MaxColAtLeft, and
// ColAtLeftOfScrn.

procedure ComputeWidth( hwnd:dword );
begin ComputeWidth;

    push( eax );
    push( ecx );
    push( edx );

    // Need to scan through all the lines we've saved up and
    // find the maximum width of all the lines:

    mov( 0, ecx );
    for( mov( 0, edx ); edx < LineCnt; inc( edx )) do

        mov( Lines[ edx*4 ], eax );
        mov( (type str.strRec [eax]).length, eax );
        if( eax > ecx ) then

            mov( eax, ecx );

        endif;

    endfor;
    mov( ecx, MaxWidth );

    // MaxColAtLeft =
    //  max( 0, MaxWidth+1 - ClientSizeX / CharWidth);

    mov( ClientSizeX, eax );
    cdq();
    idiv( CharWidth );
    neg( eax );
    add( MaxWidth, eax );
    inc( eax );
    if( @s ) then

        xor( eax, eax );

    endif;
    mov( eax, MaxColAtLeft );

    // ColAtLeftOfScrn = min( MaxColAtLeft, ColAtLeftOfScrn )

    if( eax > ColAtLeftOfScrn ) then

        mov( ColAtLeftOfScrn, eax );

    endif;
    mov( eax, ColAtLeftOfScrn );

    w.SetScrollRange( hwnd, w.SB_HORZ, 0, MaxColAtLeft, false );
    w.SetScrollPos( hwnd, w.SB_HORZ, ColAtLeftOfScrn, true );

    pop( edx );
    pop( ecx );
    pop( eax );

end ComputeWidth;


// ComputeHeight-
//
// Computes the values for the following global variables:
```

```
//
//  DisplayLines,
//  MaxLnAtTOS, and
//  LineAtTopOfScrn
//
//  This procedure also redraws the vertical scroll bars, as necessary.

procedure ComputeHeight( hwnd:dword );
begin ComputeHeight;

    push( eax );
    push( ebx );
    push( ecx );

    // DisplayLines = ClientSizeY/CharHeight:

    mov( ClientSizeY, eax );
    cdq();
    idiv( CharHeight );
    mov( eax, DisplayLines );

    // MaxLnAtTOS = max( 0, LineCnt - DisplayLines )

    mov( LineCnt, ecx );
    sub( eax, ecx );
    if( @s ) then

        xor( ecx, ecx );

    endif;
    mov( ecx, MaxLnAtTOS );

    if( edx <> 0 ) then // EDX is remainder from ClientSizeY/CharHeight

        // If we can display a partial line, bump up the
        // DisplayLine value by one to display the partial line.

        inc( DisplayLines );

    endif;


    // LineAtTopOfScrn = min( LineAtTopOfScrn, MaxLnAtTOS )

    if( ecx > LineAtTopOfScrn ) then

        mov( LineAtTopOfScrn, ecx );

    endif;
    mov( ecx, LineAtTopOfScrn );

    w.SetScrollRange( hwnd, w.SB_VERT, 0, MaxLnAtTOS, false );
    w.SetScrollPos( hwnd, w.SB_VERT, LineAtTopOfScrn, true );

    pop( ecx );
    pop( ebx );
    pop( eax );
```

```
end ComputeHeight;



// QuitApplication:
//
//  This procedure handles the w.WM_DESTROY message.
//  It tells the application to terminate.  This code sends
//  the appropriate message to the main program's message loop
//  that will cause the application to terminate.

procedure QuitApplication( hwnd: dword; wParam:dword; lParam:dword );
begin QuitApplication;

    // Free the font we created in the Create procedure:

    w.DeleteObject( FixedsysFontHandle );

    w.PostQuitMessage( 0 );

end QuitApplication;


// Create-
//
//  This procedure responds to the w.WM_CREATE message.
//  Windows sends this message once when it creates the
//  main window for the application.  We will use this
//  procedure to do any one-time initialization that
//  must take place in a message handler.

procedure Create( hwnd: dword; wParam:dword; lParam:dword );
readonly
    s           :string :=
                    "Debug Windows begin execution:" nl nl;

    FixedsysName:byte; @nostorage;
                 byte "Fixedsys", 0;

static
    hDC         :dword;
    hOldFont    :dword;
    Fixedsys    :w.LOGFONT;
    tm          :w.TEXTMETRIC;
    cds         :w.COPYDATASTRUCT;

begin Create;

    push( edi );


    // Zero out our line index, count, and the array of pointers:

    xor( eax, eax );
    mov( eax, StartLine );
    mov( eax, LineCnt );
    mov( MaxLines_c, ecx );
    mov( &Lines, edi );
    rep.stosd();

    // Zero out the FONT structure:

    mov( @size( w.LOGFONT ), ecx );
    mov( &Fixedsys, edi );
    rep.stosb();

    // Create the font using the system's "Fixedsys" font and select
    // it into the device context:

    str.zcpy( FixedsysName, Fixedsys.lfFaceName );
    w.CreateFontIndirect( Fixedsys );
    mov( eax, FixedsysFontHandle );     // Save, so we can free this later.
    GetDC( hwnd, hDC );

        // Select in the fixed system font:

        SelectObject( FixedsysFontHandle );
        mov( eax, hOldFont );

        // Determine the sizes we need for the fixed system font:

        GetTextMetrics( tm );


        mov( tm.tmHeight, eax );
        add( tm.tmExternalLeading, eax );
        inc( eax );
        mov( eax, CharHeight );

        mov( tm.tmAveCharWidth, eax );
        mov( eax, CharWidth );

        SelectObject( hOldFont );

    ReleaseDC;

    // Just for fun, let's send ourselves a message to print the
    // first line in the debug window:

    mov( DebugMsg_c, cds.dwData );
    mov( s, eax );
    mov( eax, cds.lpData );
    mov( (type str.strRec [ eax] ).length, eax );
    inc( eax ); // Count zero byte, too!
    mov( eax, cds.cbData );
    w.SendMessage( hwnd, w.WM_COPYDATA, hwnd, &cds );

    pop( edi );

end Create;


// RcvMsg-
```

```
//
//  Receives a message from some other process and prints
// the zstring passed as the data payload for that message.
// Note that lParam points at a w.COPYDATASTRUCT object. The
// dwData field of that structure must contain DebugMsg_c if
// we are to process this message.

procedure RcvMsg( hwnd: dword; wParam:dword; lParam:dword );
var
    tabPosn :uns32;
    line    :char[ 256];

    procedure addLine( lineToAdd:string in eax ); @nodisplay; @noframe;
    begin addLine;

        if( LineCnt >= MaxLines_c ) then

            mov( StartLine, ecx );
            strfree( Lines[ ecx*4 ] ); // Free the oldest line of text.
            inc( StartLine );
            if( StartLine >= MaxLines_c ) then

                mov( 0, StartLine );

            endif;

        else

            mov( LineCnt, ecx );
            inc( LineCnt );

        endif;
        mov( eax, Lines[ ecx*4 ] );

        // If we've got more than "DisplayLines" lines in the
        // output, bump "LineAtTopOfScrn" to scroll the window up:

        mov( LineCnt, ecx );
        if( ecx >= DisplayLines ) then

            inc( LineAtTopOfScrn );

        endif;
        ret();

    end addLine;

begin RcvMsg;

    push( esi );
    push( edi );

    // Process the incoming zero-terminated string.
    // Break it into separate lines (based on newline
    // sequences found in the string) and expand tabs.

    mov( lParam, esi );
```

```
if( (type w.COPYDATASTRUCT [esi]).dwData = DebugMsg_c ) then

    // Okay, we've got the w.COPYDATASTRUCT type with a valid debug
    // message. Extract the data and print it.

    mov( (type w.COPYDATASTRUCT [esi]).lpData, esi );
    mov( 0, tabPosn );
    lea( edi, line );
    while( (type char [esi]) <> #0 ) do

        mov( [esi], al );

        // Ignore carriage returns:

        if( al <> stdio.cr ) then

            if( al = stdio.tab ) then

                // Compute the number of spaces until the
                // next tab stop position:

                mov( tabPosn, eax );
                cdq();
                div( tabCols, edx:eax );
                neg( edx );
                add( tabCols, edx );

                // Emit spaces up to the next tab stop:

                repeat

                    mov( ' ', (type char [edi]) );
                    inc( edi );
                    inc( tabPosn );
                    if( tabPosn >= 255 ) then

                        dec( edi );

                    endif;
                    dec( edx );

                until( @z );

            elseif( al = stdio.lf ) then

                // We've just hit a new line character.
                // Emit the line up to this point:

                mov( #0, (type char [edi]) ); // Zero terminate.
                lea( edi, line );       // Resets edi for next loop iteration.
                str.a_cpyz( [edi] );    // Build HLA string from zstring.
                addLine( eax );         // Add to our list of lines.

                // Reset the column counter back to zero since
                // we're starting a new line:

                mov( 0, tabPosn );
            else

                // If it's not a special control character we process,
                // then add the character to the string:

                mov( al, [ edi] );
                inc( edi );
                inc( tabPosn );

                // Don't allow more than 255 characters:

                if( tabPosn >= 255 ) then

                    dec( edi );

                endif;

            endif;

        endif;

        // Move on to next character in source string:

        inc( esi );

    endwhile;

    // If edi is not pointing at the beginning of "line", then we've
    // got some characters in the "line" string that need to be added
    // to our list of lines.

    lea( esi, line );
    if( edi <> esi ) then

        mov( #0, (type char [ edi]) );   // Zero terminate the string.
        str.a_cpyz( [ esi] );            // Make a copy of this zstring.
        addLine( eax );

    endif;

    // Because adding and removing lines can affect the
    // maximum line length, recompute the width and horizontal
    // scroll bar stuff here:

    ComputeWidth( hwnd );

    // Ditto for the vertical scroll bars:

    ComputeHeight( hwnd );

    // Tell Windows to tell us to repaint the screen:

    w.InvalidateRect( hwnd, NULL, true );
    w.UpdateWindow( hwnd );

endif;
```

```
    pop( edi );
    pop( esi );

end RcvMsg;


// Paint:
//
//  This procedure handles the w.WM_PAINT message.
//  For this program, the Paint procedure draws all the
// lines from the scroll position to the end of the window.

procedure Paint( hwnd: dword; wParam:dword; lParam:dword );
var
    hdc         :dword;         // Handle to video display device context
    hOldFont    :dword;         // Saves old font handle.
    ps          :w.PAINTSTRUCT; // Used while painting text.

begin Paint;

    // Message handlers must preserve EBX, ESI, and EDI.
    // (They've also got to preserve EBP, but HLA's procedure
    // entry code already does that.)

    push( ebx );
    push( esi );
    push( edi );

    // When Windows requests that we draw the window,
    // fill in the string in the center of the screen.
    // Note that all GDI calls (e.g., w.DrawText) must
    // appear within a BeginPaint..EndPaint pair.

    BeginPaint( hwnd, ps, hdc );

        SelectObject( FixedsysFontHandle );
        mov( eax, hOldFont );


        // Figure out how many lines to display;
        // It's either DisplayLines (the number of lines we can
        // physically put in the Window) or LineCnt (total number
        // of lines in the Lines array), whichever is less:

        mov( DisplayLines, esi );
        if( esi > LineCnt ) then

            mov( LineCnt, esi );

        endif;

        // Display each of the lines in the Window:

        for( mov( 0, ebx ); ebx < esi; inc( ebx )) do

            // Get the string to display (address to edi):
```

```
            mov( LineAtTopOfScrn, eax );
            add( ebx, eax );
            if( eax < LineCnt ) then

                // If we've got more than a buffer full of lines,
                // base our output at "StartLine" rather than at
                // index zero:

                add( StartLine, eax );
                if( eax >= LineCnt ) then

                    sub( LineCnt, eax );

                endif;

                // Get the current line to output:

                mov( Lines[ eax*4 ], edi );

                // Compute the y-coordinate in the window:

                mov( ebx, eax );
                intmul( CharHeight, eax );

                // Compute the starting column position into the
                // line and the length of the line (to handle
                // horizontal scrolling):

                mov( (type str.strRec [ edi] ).length, ecx );
                add( ColAtLeftOfScrn, edi );
                sub( ColAtLeftOfScrn, ecx );

                // Output the line of text:

                TextOut
                (
                    CharWidth,
                    eax,
                    edi,
                    ecx
                );

            endif;

        endfor;
        SelectObject( hOldFont );

    EndPaint;

    pop( edi );
    pop( esi );
    pop( ebx );

end Paint;
```

```
// Size-
//
//  This procedure handles the w.WM_SIZE message
//
//  L.O. word of lParam contains the new X Size
//  H.O. word of lParam contains the new Y Size

procedure Size( hwnd: dword; wParam:dword; lParam:dword );
begin Size;

    // Convert new X size to 32 bits and save:

    movzx( (type word lParam), eax );
    mov( eax, ClientSizeX );

    // Convert new Y size to 32 bits and save:

    movzx( (type word lParam[ 2]), eax );
    mov( eax, ClientSizeY );

    // Compute the new screen height info here:

    ComputeHeight( hwnd );


    // Compute screen width and MaxWidth values, and set up the
    // horizontal scroll bars:

    ComputeWidth( hwnd );

    xor( eax, eax ); // return success.

end Size;



// HScroll-
//
//  Handles w.WM_HSCROLL messages.
//  On entry, L.O. word of wParam contains the scroll bar activity.

procedure HScroll( hwnd: dword; wParam:dword; lParam:dword );
begin HScroll;

    movzx( (type word wParam), eax );
    mov( ColAtLeftOfScrn, ecx );
    if( eax = w.SB_LINELEFT ) then

        // Scrolling left means decrement ColAtLeftOfScrn by one:

        dec( ecx );

    elseif( eax = w.SB_LINERIGHT ) then

        // Scrolling right means increment ColAtLeftOfScrn by one:
```

```
        inc( ecx );

    elseif( eax = w.SB_PAGELEFT ) then

        // Page Left means decrement ColAtLeftOfScrn by 25% of screen width:

        mov( ClientSizeX, eax );
        cdq();
        div( CharWidth, edx:eax );  // Computes screen width in chars.
        shr( 2, eax );              // 25% of screen width.
        adc( 0, eax );
        sub( eax, ecx );

    elseif( eax = w.SB_PAGERIGHT ) then

        // Page Right means increment ColAtLeftOfScrn by 25% of screen width:

        mov( ClientSizeX, eax );
        cdq();
        div( CharWidth, edx:eax );  // Computes screen width in chars.
        shr( 2, eax );              // 25% of screen width.
        adc( 0, eax );
        add( eax, ecx );


    elseif( eax = w.SB_THUMBTRACK ) then

        // H.O. word of wParam contains new scroll thumb position:

        movzx( (type word wParam[ 2] ), ecx );

    // else leave ColAtLeftOfScrn alone

    endif;

    // Make sure that the new ColAtLeftOfScrn value (in ecx) is within
    // a reasonable range (0..MaxColAtLeft):

    if( (type int32 ecx) < 0 ) then

        xor( ecx, ecx );

    elseif( ecx > MaxColAtLeft ) then

        mov( MaxColAtLeft, ecx );

    endif;
    mov( ColAtLeftOfScrn, eax );
    mov( ecx, ColAtLeftOfScrn );
    sub( ecx, eax );

    if( eax <> 0 ) then

        intmul( CharWidth, eax );
        w.ScrollWindow( hwnd, eax, 0, NULL, NULL );
        w.SetScrollPos( hwnd, w.SB_HORZ, ColAtLeftOfScrn, true );
        w.InvalidateRect( hwnd, NULL, true );
    endif;
    xor( eax, eax ); // return success

end HScroll;



// VScroll-
//
//  Handles the w.WM_VSCROLL messages from Windows.
//  The L.O. word of wParam contains the action/command to be taken.
//  The H.O. word of wParam contains a distance for the w.SB_THUMBTRACK
//  message.

procedure VScroll( hwnd: dword; wParam:dword; lParam:dword );
begin VScroll;

    movzx( (type word wParam), eax );
    mov( LineAtTopOfScrn, ecx );
    if( eax = w.SB_TOP ) then

        // Top of file means LATOS becomes zero:

        xor( ecx, ecx );

    elseif( eax = w.SB_BOTTOM ) then

        // Bottom of file means LATOS becomes MaxLnAtTOS:

        mov( MaxLnAtTOS, ecx );

    elseif( eax = w.SB_LINEUP ) then

        // LineUp - Decrement LATOS:

        dec( ecx );

    elseif( eax = w.SB_LINEDOWN ) then

        // LineDn - Increment LATOS:

        inc( ecx );

    elseif( eax = w.SB_PAGEUP ) then

        // PgUp - Subtract DisplayLines from LATOS:

        sub( DisplayLines, ecx );

    elseif( eax = w.SB_PAGEDOWN ) then

        // PgDn - Add DisplayLines to LATOS:

        add( DisplayLines, ecx );

    elseif( eax = w.SB_THUMBTRACK ) then
```

```
        // ThumbTrack - Set LATOS to L.O. word of wParam:

        movzx( (type word wParam[ 2]), ecx );

    // else - leave LATOS alone

    endif;


    // Make sure LATOS is in the range 0..MaxLnAtTOS-

    if( (type int32 ecx) < 0 ) then

        xor( ecx, ecx );

    elseif( ecx >= MaxLnAtTOS ) then

        mov( MaxLnAtTOS, ecx );

    endif;
    mov( LineAtTopOfScrn, eax );
    mov( ecx, LineAtTopOfScrn );
    sub( ecx, eax );

    if( eax <> 0 ) then

        intmul( CharHeight, eax );
        w.ScrollWindow( hwnd, 0, eax, NULL, NULL );
        w.SetScrollPos( hwnd, w.SB_VERT, LineAtTopOfScrn, true );
        w.InvalidateRect( hwnd, NULL, true );
        w.UpdateWindow( hwnd );

    endif;
    xor( eax, eax ); // return success.


end VScroll;



/**************************************************************************/
/*                    End of Application Specific Code                    */
/**************************************************************************/





// The window procedure.
// 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( hwnd:dword; uMsg:uns32; wParam:dword; lParam:dword );
    @stdcall;

begin WndProc;
```

```
    // uMsg contains the current message Windows is passing along to
    // us.  Scan through the "Dispatch" table searching for a handler
    // for this message.  If we find one, then call the associated
    // handler procedure.  If we don't have a specific handler for this
    // message, then call the default window procedure handler function.

    mov( uMsg, eax );
    mov( &Dispatch, edx );
    forever

        mov( (type MsgProcPtr_t [edx]).MessageHndlr, ecx );
        if( ecx = 0 ) then

            // If an unhandled message comes 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.

            w.DefWindowProc( hwnd, uMsg, wParam, lParam );
            exit WndProc;


        elseif( eax = (type MsgProcPtr_t [edx]).MessageValue ) then

            // If the current message matches one of the values
            // in the message dispatch table, then call the
            // appropriate routine.  Note that the routine address
            // is still in ECX from the test above.

            push( hwnd );   // (type tMsgProc ecx)(hwnd, wParam, lParam)
            push( wParam ); //  This calls the associated routine after
            push( lParam ); //  pushing the necessary parameters.
            call( ecx );

            sub( eax, eax ); // Return value for function is zero.
            break;

        endif;
        add( @size( MsgProcPtr_t ), edx );

    endfor;

end WndProc;



// Here's the main program for the application.

begin DebugWindow;

    // Get this process' handle:

    w.GetModuleHandle( NULL );
    mov( eax, hInstance );
    // Set up the window class (wc) object:

    mov( @size( w.WNDCLASSEX ), wc.cbSize );
    mov( w.CS_HREDRAW | w.CS_VREDRAW, wc.style );
    mov( &WndProc, wc.lpfnWndProc );
    mov( NULL, wc.cbClsExtra );
    mov( NULL, wc.cbWndExtra );
    mov( w.COLOR_WINDOW+1, wc.hbrBackground );
    mov( NULL, wc.lpszMenuName );
    mov( ClassName, wc.lpszClassName );
    mov( hInstance, wc.hInstance );

    // Get the icons and cursor for this application:

    w.LoadIcon( NULL, val w.IDI_APPLICATION );
    mov( eax, wc.hIcon );
    mov( eax, wc.hIconSm );

    w.LoadCursor( NULL, val w.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.

    w.RegisterClassEx( wc );

    w.CreateWindowEx
    (
        NULL,
        ClassName,
        AppCaption,
        w.WS_OVERLAPPEDWINDOW | w.WS_VSCROLL |  w.WS_HSCROLL,
        w.CW_USEDEFAULT,
        w.CW_USEDEFAULT,
        w.CW_USEDEFAULT,
        w.CW_USEDEFAULT,
        NULL,
        NULL,
        hInstance,
        NULL
    );
    mov( eax, hwnd );

    w.ShowWindow( hwnd, w.SW_SHOWNORMAL );
    w.UpdateWindow( hwnd );

    // Here's the event loop that processes messages
    // sent to our window.  On return from GetMessage,
    // break if EAX contains false and then quit the
    // program.

    forever

        w.GetMessage( msg, NULL, 0, 0 );
        breakif( !eax );
        w.TranslateMessage( msg );
        w.DispatchMessage( msg );

    endfor;

    // The message handling inside Windows has stored
    // the program's return code in the wParam field
    // of the message.  Extract this and return it
    // as the program's return code.

    mov( msg.wParam, eax );
    w.ExitProcess( eax );

end DebugWindow;
```

6.6.4: Using *DebugWindow*

By itself, the *DebugWindow* application isn t very interesting. When you run it, it displays an initial debug message and then waits for another message to arrive. In order to make practical use of this application, other applications must send messages to *DebugWindow* so it can display their debug output. In this section, we ll explore the additions you ll need to make to an application in order to send output to *DebugWindow*.

Of course, the obvious way to send output to the *DebugWindow* process is to explicitly build up `w.COPY-DATASTRUCT` objects and make the `w.SendMessage` call yourself. There are two problems with this approach - first, it s a lot of work. Of course, it s easy enough to write a short procedure you can pass a string that will build the data structure and call `w.SendMessage`, so this problem is easily corrected. Another problem, not quite as easy to fix, is that HLA users (particularly those coming from a console application background) expect to be able to use a statement like dbg.put(hwnd, ...);[5] to print all kinds of data, not just string data. So this is what we want to shoot for, something like the `stdout.put` or `str.put` macros that will allow a *DebugWindow* user to easily print all sorts of data.

Another issue with the debug output statements that we place in our applications is the effect they have on the application s size and performance. Although debug statements are quite useful during the debugging and testing phases of program development, we don t want these statements taking up space or slowing down the program when we produce a production version of the software. That is, we want to easily enable all these statements during debugging, but we want to be able to immediately remove them all when compiling a production version of the software. Fortunately, HLA s compile-time language provides all the facilities we need to achieve this.

To control the emission of debug code in our applications, we ll use a tried and true mechanism: *conditional assembly*. HLA s `#if` statement allows use to easily activate or eliminate sequences of statements during a compile by defining certain symbols or setting certain constants true or false. For example, consider the following code sequence:

```
#if( @defined( debug ))

   dbg.put( hwnd, "Reached Point A" nl ); // hwnd is this process' window handle

#endif
```

5. The `hwnd` parameter is the application s window handle. `dbg.put` will need this value to pass on to the *DebugWindow* process via the `wParam` parameter of the `w.SendMessage` API function.

This sequence of statements checks to see if the symbol debug is defined prior to the `#if` statement. If so, then HLA will compile the `dbg.put` statement; if not, then HLA will treat everything between the `#if` and `#endif` as a comment (and will ignore it). You may control the definition of the `debug` symbol a couple of different ways. One way is to explicitly place a statement like the following near the beginning of your program:

?debug := true;

(Note that the type and value of this symbol are irrelevant; the `@defined` compile-time function simply checks the symbol to see if it is previously defined; the convention for such symbols is to declare them as boolean constants and initialize them with true.)

Another way to define a symbol is from the HLA command line. Consider the following Windows cmd.exe command:

hla -Ddebug AppUsesDbgWin.hla

This command line will predefine the symbol `debug` as a boolean constant that is set to the value true. Note that the generic makefiles we ve created for HLA applications include a debug option that defines this symbol when you specify debug from the command line.

The only problem with using the `@defined` compile-time function to control conditional assembly is that it s an all or nothing proposition. A symbol is either defined or it is not defined in HLA. You cannot arbitrarily undefine a symbol. This means that you cannot turn the debugging code on or off on a line-by-line basis in your files. A better solution is to use a boolean compile-time variable and set it to true or false. This allows you to activate or deactivate the debug code at will throughout your source file, e.g.,

```
?debug := true;
      .
      .
      .
   << During this section of the file, debugging is active >>
      .
      .
      .
?debug := false;
      .
      .
      .
   << During this section of the file, debugging is deactivated >>
      .
      .
      .
?debug := true;
      .
      .
      .
   << Here, the debugging code is once again active >>
      .
      .
      .
```

The only catch to this approach is that you must define `debug` before the first conditional assembly statement that tests the value of debug, or HLA will report an error. An easy way to ensure that `debug` is defined is to include code like this in the header file that contains the other definitions for the calls that output data to the debug window:

```
#if( !@defined( debug ))

   // If the symbol is not defined, default to a value of false-

   ?debug := false;

#endif
```

The combination of the definition of a symbol like debug and the use of conditional assembly (also called *conditional compilation*) in the program provides exactly what we need - the ability to quickly activate or deactivate the debugging code by defining (or not defining) a single symbol. There is, however, a minor problem with this approach - all those #if and #endif statements tend to clutter up the program and make it harder to read. Fortunately, there is an easy solution to this problem - make the dbg.put invocation a macro rather than a procedure call[6]. The dbg.put macro would then look something like the following:

```
namespace dbg; // This is where the "dbg." comes from...

   #macro put( hwnd, operand );
      #if( debug )
         <<Code that passes the string data to DebugWindow>>
      #endif
   #endmacro

end dbg;
```

With a macro definition like this, you can write code like the following and the system will automatically make the code associated with dbg.put go away if you ve not set the debug symbol to true:

dbg.put(hwnd, Some Message nl);

Of course, we d like dbg.put to be able to handle multiple operands and automatically convert non-string data types to their string format for output (just like stdout.put and str.put). We could copy the code for the stdout.put or str.put macros into the dbg namespace with the appropriate modification to construct a string to send to the *DebugWindow* application. However, this is a lot of code to copy and modify (these macros are several hundred lines long each). It turns out, however, that all we really need to do is allocate storage for a string, pass the dbg.put parameters to str.put (that will convert the data to string format), and then pass the resulting string on to the *DebugWindow* application. That is, we could get by with the following two statements:

```
   str.put( someStr, <<dbg.put parameters>> );
   sendStrToDebugWindow( hwnd, someStr );
```

The only major complication here is the need to allocate storage for a string (someStr) before executing these two statements. A minor issue is the fact that you cannot easily pass all the parameters from one variable-length parameter macro to another. Fortunately, with the HLA compile-time language and an understanding of the HLA string format, it s easy enough to write a macro that automates these two statements for us. Consider the following code:

6. It turns out, we have to do this anyway, so this is a trivial modification to make to the program.

```
namespace dbg;

    #macro put( _hwnd_dbgput_, _dbgput_parms_[] ):
            _dbgput_parmlist_,
            _cur_dbgput_parm_;

        #if( debug & @elements( _dbgput_parms_ ) <> 0 )

            sub( 4096, esp );       // Make room for string on stack.
            push( edi );            // We need a register, EDI is as good as any.
            lea( edi, [esp+12]);    // Turn EDI into a string pointer.

            // Initialize the maxlength field of our string (at [edi-8]) with 4084
            // (this string has 12 bytes of overhead).

            mov( 4084, (type str.strRec [edi]).MaxStrLen );

            // Initialize the current length to zero (empty string ):

            mov( 0, (type str.strRec [edi]).length );

            // Zero terminate the string (needed even for empty strings):

            mov( #0, (type char [edi]) );

            // Okay, invoke the str.put macro to process all the dbg.put
            // parameters passed to us. Note that we have to feed our parameters
            // to str.put one at a time:

            ?_dbgput_parmlist_ := "";
            #for( _cur_dbgput_parm_ in _dbgput_parms_ )

                ?_dbgput_parmlist_ :=
                    _dbgput_parmlist_ + "," + _cur_dbgput_parm_;

            #endfor

            str.put
            (
                (type string edi)   // Address of our destination string
                @text( _dbgput_parmlist_ )
            );  // End of str.put macro invocation
            dbg.sendStrToDebugWindow( _hwnd_dbgput_, (type string edi) );

            // Clean up the stack now that we're done:

            pop( edi );
            add( 4096, esp );

        #endif

    #endmacro

   #if( @defined( debug ))

      procedure sendStrToDebugWindow( hwnd:dword; s:string );
```

```
        begin sendStrToDebugWindow;

           << Code that sends s to DebugWindow >>

        end sendStrToDebugWindow;

    #endif

end dbg;
```

This code creates an HLA string on the stack (maximum of 4084 characters, which is probably more than enough for just about any debug message you can imagine[7]) and initializes it appropriately. Then it invokes the `str.put` macro to convert `dbg.put's` parameters to string form. Note how this code uses a compile-time `#for` loop to process all the `dbg.put` parameters (variable parameters supplied to a macro are translated to an array of strings by HLA; the `#for` loop processes each element of this array). Also note the use of the `@text` function to convert each of these strings into text for use by `str.put` (and also note the sneaky placement of the comma inside the `#for` loop so that we get exactly enough commas, in all the right places, when the `#for` loop executes during compile time). Effectively, the `#for` loop is simple injecting all of the `dbg.put` parameters into the `str.put` macro invocation after the first parameter (the destination string). This implementation of `dbg.put` requires a whole lot less code than implement it directly (i.e., by writing the same code that the `str.put` and `stdout.put` macros require).

The only thing left to implement is the `sendStrToDebugWindow` procedure. Here s the code for that procedure:

```
     #if( @global:debug )

         procedure sendStrToDebugWindow( hwnd:dword; s:string );
             @nodisplay;
             @noalignstack;
         var
             cds :@global:w.COPYDATASTRUCT;

         static
             GoodHandle :boolean := @global:true;

         begin sendStrToDebugWindow;

             push( eax );
             push( ebx );
             push( ecx );
             push( edx );

             mov( s, ebx );
             mov( (type @global:str.strRec [ ebx] ).length, eax );
             inc( eax ); // Account for zero terminating byte
             mov( ebx, cds.lpData );
             mov( eax, cds.cbData );
             mov( @global:DebugMsg_c, cds.dwData );
             @global:w.FindWindow( "DebugWindowClass", NULL );

             // Logic to bring up a dialog box complaining that
```

7. If you attempt to print more than 4084 characters with a single dbg.put macro, HLA will raise a string overflow exception.

```
        // DebugWindow is not current running if we fail to
        // find the window.  Note that we don't want to display
        // this dialog box on each execution of a dbg.put call,
        // only on the first instance where we get a w.FindWindow
        // failure.

        if( GoodHandle ) then

            if( eax = NULL ) then

                // Whenever GoodHandle goes from true to false,
                // print the following error message in a dialog box:

                @global:w.MessageBox
                (
                    0,
                    "Debug Windows is not active!",
                    "dbg.put Error",
                    @global:w.MB_OK
                );
                mov( @global:false, GoodHandle ); // Only report this once.
                xor( eax, eax );    // Set back to NULL

            endif;

        else

            // If the handle becomes good after it was bad,
            // reset the GoodHandle variable.  That way if
            // DebugWindow dies sometime later, we can once again
            // bring up the dialog box.

            if( eax <> NULL ) then

                mov( @global:true, GoodHandle );

            endif;

        endif;
        lea( ebx, cds );
        @global:w.SendMessage( eax, @global:w.WM_COPYDATA, hwnd, ebx );

        pop( edx );
        pop( ecx );
        pop( ebx );
        pop( eax );

    end sendStrToDebugWindow;

  #endif
```

The `@global:` prefixes before several of the identifiers exist because these symbols are not local to the `dbg` namespace and HLA requires the `@global:` prefix when referencing symbols that are not local to the name space (this was not necessary in the macro `put` because HLA doesn t process the macro body until you actually expand the `put` macro, and usually this is outside the namespace). You ll also notice a quick check to see if `w.FindWindow` returns a valid handle. If this is the first call to `sendStrToDebugWindow` or the previous call to `w.FindWindow` returned a valid handle, and the current call returns an invalid handle, then this procedure brings up a dialog box to warn the user that the *DebugWindow* application is not active. To display this dialog box, `sendStrToDebugWindow` calls the `w.MessageBox` API function:

```
static
    MessageBox: procedure
    (
        hWnd            :dword;
        lpText          :string;
        lpCaption       :string;
        uType           :dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__MessageBoxA@16" );
```

The `hWnd` parameter is the window handle for the calling procedure, the `lpText` parameter is a string that Windows displays within the dialog box, the `lpCaption` parameter is a string that Windows displays on the title bar of the dialog box, and the last parameter specifies the type of buttons to display in the dialog box (e.g., the `w.MB_OK` constant tells `w.MessageBox` to display an OK button). Note that this code doesn t display the message box on each failure by `w.FindWindow`. If the *DebugWindow* application is not running, it would be annoying to have this dialog box appear every time the application tried to send a debug message to *DebugWindow*. If the user starts up the *DebugWindow* application after the dialog box appears, then `sendStrToDebugWindow` repairs itself and begins sending the messages again.

The astute reader (and expert assembly language programmer) might notice a whole bunch of inefficiencies present in this code. For example, this code calls `w.FindWindow` for every debug message it processes. It wouldn t be too difficult to allocate a static variable inside `sendStrToDebugWindow` and initialize that variable on the very first call, saving the overhead of the call to `w.FindWindow` on future calls. Other inefficiencies include the fact that the `put` macro generates a ton of code to construct and destroy the string on each invocation of the `dbg.put` macro. Even within the *DebugWindow* application itself there are some situations where performance could be improved (e.g., this code could be smarter about how often it recomputes the maximum width and height values). On the other hand, keep in mind that the purpose of this code is to support debugging, not support a production version of an application. The minor inefficiencies present in this code are of little concern because such inefficiencies will not appear in the final code anyway.

We re not done with the *DebugWindow* application by any means. There are many, many, improvements we can make to this code. We ll revisit this application in later chapters when we cover features such as menus. At that time, we ll enhance the operation of this application to make it even more powerful and flexible.

6.7: The Windows Console API

A chapter on text manipulation in a Windows programming book isn t really complete without a brief look at the Win32 console API. Windows provides several features that let you manipulate text on the display from a console application (i.e., traditional text-based application), providing considerable control over the text display. Although the thrust of this book is to teach you how to write event-driven GUI applications under Windows, it s worth pointing out the console features that Windows provides because many applications run just fine in a text environment.

Note that, by default, HLA produces console applications when you link in and use the HLA Standard Library (e.g., when calling functions like HLA s `stdout.put` routine). In fact, HLA even provides a CONSOLE library module that provides a thin layer above the Win32 console API. The HLA Standard Library console module, however, doesn t provide anywhere near the total capabilities that are possible with the Win32 API (HLA s console module purposefully limits its capabilities to produce a modicum of compatibility with Linux console code). We ll take a look at many of the Win32 Console API capabilities in this section.

6.7.1: Windows' Dirty Secret - GUI Apps Can Do Console I/O!

A well-kept Microsoft secret, well, at least a seldom-used feature in Windows is that every application, including GUI applications, can have a console window associated with them. Conversely, every application can open up a graphics window. In fact, the only difference between a console app and a GUI app under Windows is that console applications automatically allocate and use a console window when they begin execution, whereas GUI applications do not automatically create a console window. However, there is nothing stopping you from allocating your own console window in a GUI app. This section will demonstrate how to do that.

Creating a console window for use by a GUI app is very simple - just call the `w.AllocConsole` function. This function has the following prototype:

```
static
    AllocConsole: procedure;
        @stdcall;
        @returns( "eax" ); // Zero if failure
        @external( "__imp__AllocConsole@0" );
```

You may call `w.AllocConsole` from just about anywhere in your program that you want (generally, calling it as the first statement of your main program is a good idea). However, you may only call this function once (unless you call the corresponding `w.FreeConsole` function to deallocate the console window prior to calling `w.AllocConsole` a second time; see the Windows documentation for more details about `w.FreeConsole`).

Once you call `w.AllocConsole`, Windows attaches the standard input, standard output, and standard error devices to that console window. This means that you can use any code that writes to the standard output (or standard error) or reads from the standard input to do I/O in the new console window you ve created. Because HLA s `stdout.XXXX` and `stdin.XXXX` procedures read and write the standard input and standard output.

In the previous section, this chapter discussed the creation of the *DebugWindow* application that lets one application write debug information to a console-like text window. The *DebugWindow* application is really nice because if your main application crashes, the *DebugWindow* application is still functioning and you can scroll through the debug output your application produced to try and determine the root cause of your problem. However, if all you want to do is display some text while your program is operating normally, using an application like *DebugWindow* to capture the output of your application is overkill. A better solution is to allocate and open the console window associated with the application and then use HLA s Standard Library `stdout.XXXX` (and `stdin.XXXX`) functions to manipulate the console window. The following *ConsoleHello* program is very similar to the *DBGHello* program from the previous section, except it writes its output to the console window it opens up rather than sending this output to the *DebugWindow* application.

```
// ConsoleHello.hla:
//
// Displays "Hello World" in a window.

program ConsoleHello;
#include( "stdlib.hhf" )     // HLA Standard Library.
#include( "w.hhf" )          // Standard windows stuff.
#include( "wpa.hhf" )        // "Windows Programming in Assembly" specific stuff.
```

```
?@nodisplay := true;
?@nostackalign := true;

static
    hInstance:  dword;          // "Instance Handle" supplied by Windows.

    wc:     w.WNDCLASSEX;       // Our "window class" data.
    msg:    w.MSG;              // Windows messages go here.
    hwnd:   dword;              // Handle to our window.


readonly

    ClassName:  string := "TextAttrWinClass";       // Window Class Name
    AppCaption: string := "Text Attributes";        // Caption for Window

// The following data type and DATA declaration
// defines the message handlers for this program.

type
    MsgProc_t:  procedure( hwnd:dword; wParam:dword; lParam:dword );

    MsgProcPtr_t:
        record

            MessageValue:   dword;
            MessageHndlr:   MsgProc_t;

        endrecord;



// The dispatch table:
//
//  This table is where you add new messages and message handlers
//  to the program.  Each entry in the table must be a tMsgProcPtr
//  record containing two entries: the message value (a constant,
//  typically one of the wm.***** constants found in windows.hhf)
//  and a pointer to a "tMsgProc" procedure that will handle the
//  message.

readonly

    Dispatch:   MsgProcPtr_t; @nostorage;

        MsgProcPtr_t
            MsgProcPtr_t:[  w.WM_DESTROY, &QuitApplication   ],
            MsgProcPtr_t:[  w.WM_PAINT,   &Paint             ],
            MsgProcPtr_t:[  w.WM_CREATE,  &Create            ],
            MsgProcPtr_t:[  w.WM_SIZE,    &Size              ],

            // Insert new message handler records here.

            MsgProcPtr_t:[  0, NULL ];   // This marks the end of the list.
```

```
/**************************************************************************/
/*          A P P L I C A T I O N   S P E C I F I C   C O D E             */
/**************************************************************************/

// QuitApplication:
//
//  This procedure handles the "wm.Destroy" message.
//  It tells the application to terminate.  This code sends
//  the appropriate message to the main program's message loop
//  that will cause the application to terminate.


procedure QuitApplication( hwnd: dword; wParam:dword; lParam:dword );
begin QuitApplication;

    w.PostQuitMessage( 0 );

end QuitApplication;


// Create-
//
//  Just send a message to our console window when this message comes along:

procedure Create( hwnd: dword; wParam:dword; lParam:dword );
begin Create;

    // When the create message comes along, send a message to
    // the console window:

    stdout.put( "Create was called" nl );

end Create;



// Size-
//
//  Display the window's new size whenever this message comes along.

procedure Size( hwnd: dword; wParam:dword; lParam:dword );
begin Size;

    // When the size message comes along, send a message to
    // the console window:

    stdout.put
    (
        "Size was called, new X-size is ",
        (type uns16 lParam),
        " new Y-size is ",
        (type uns16 lParam[ 2] ),
        nl
    );

end Size;
```

```
// Paint:
//
//  This procedure handles the "wm.Paint" message.
//  For this simple "Hello World" application, this
//  procedure simply displays "Hello World" centered in the
//  application's window.

procedure Paint( hwnd: dword; wParam:dword; lParam:dword ); @nodisplay;
var
    hdc:    dword;              // Handle to video display device context
    ps:     w.PAINTSTRUCT;      // Used while painting text.
    rect:   w.RECT;             // Used to invalidate client rectangle.

static
    PaintCnt    :uns32 := 0;

begin Paint;

    // Display a count to the console window:

    inc( PaintCnt );
    stdout.put( "Paint was called (cnt:", PaintCnt, ")" nl );

    // When Windows requests that we draw the window,
    // fill in the string in the center of the screen.
    // Note that all GDI calls (e.g., w.DrawText) must
    // appear within a BeginPaint..EndPaint pair.

    w.BeginPaint( hwnd, ps );
    mov( eax, hdc );

    w.GetClientRect( hwnd, rect );
    w.DrawText
    (
        hdc,
        "Hello World!",
        -1,
        rect,
        w.DT_SINGLELINE | w.DT_CENTER | w.DT_VCENTER
    );

    w.EndPaint( hwnd, ps );

end Paint;

/**************************************************************************/
/*                    End of Application Specific Code                    */
/**************************************************************************/




// 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( hwnd:dword; uMsg:uns32; wParam:dword; lParam:dword  );
    @stdcall;
    @nodisplay;
    @noalignstack;

begin WndProc;

    // uMsg contains the current message Windows is passing along to
    // us.  Scan through the "Dispatch" table searching for a handler
    // for this message.  If we find one, then call the associated
    // handler procedure.  If we don't have a specific handler for this
    // message, then call the default window procedure handler function.

    mov( uMsg, eax );
    mov( &Dispatch, edx );
    forever

        mov( (type MsgProcPtr_t [edx]).MessageHndlr, ecx );
        if( ecx = 0 ) then

            // If an unhandled message comes 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.

            w.DefWindowProc( hwnd, uMsg, wParam, lParam );
            exit WndProc;


        elseif( eax = (type MsgProcPtr_t [edx]).MessageValue ) then

            // If the current message matches one of the values
            // in the message dispatch table, then call the
            // appropriate routine.  Note that the routine address
            // is still in ECX from the test above.

            push( hwnd );   // (type tMsgProc ecx)(hwnd, wParam, lParam)
            push( wParam ); //  This calls the associated routine after
            push( lParam ); //  pushing the necessary parameters.
            call( ecx );

            sub( eax, eax ); // Return value for function is zero.
            break;

        endif;
        add( @size( MsgProcPtr_t ), edx );

    endfor;

end WndProc;
```

```
// Here's the main program for the application.

begin ConsoleHello;

    // Create a console window for this application:

    w.AllocConsole();

    // Set up the window class (wc) object:

    mov( @size( w.WNDCLASSEX ), wc.cbSize );
    mov( w.CS_HREDRAW | w.CS_VREDRAW, wc.style );
    mov( &WndProc, wc.lpfnWndProc );
    mov( NULL, wc.cbClsExtra );
    mov( NULL, wc.cbWndExtra );
    mov( w.COLOR_WINDOW+1, wc.hbrBackground );
    mov( NULL, wc.lpszMenuName );
    mov( ClassName, wc.lpszClassName );

    // Get this process' handle:

    w.GetModuleHandle( NULL );
    mov( eax, hInstance );
    mov( eax, wc.hInstance );

    // Get the icons and cursor for this application:

    w.LoadIcon( NULL, val w.IDI_APPLICATION );
    mov( eax, wc.hIcon );
    mov( eax, wc.hIconSm );

    w.LoadCursor( NULL, val w.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.

    w.RegisterClassEx( wc );

    w.CreateWindowEx
    (
        NULL,
        ClassName,
        AppCaption,
        w.WS_OVERLAPPEDWINDOW,
        w.CW_USEDEFAULT,
        w.CW_USEDEFAULT,
        w.CW_USEDEFAULT,
        w.CW_USEDEFAULT,
        NULL,
        NULL,
        hInstance,
        NULL
    );
    mov( eax, hwnd );

    w.ShowWindow( hwnd, w.SW_SHOWNORMAL );
    w.UpdateWindow( hwnd );

    // Demonstrate printing to the console window:

    stdout.put( "Hello World to the console" nl );

    // Here's the event loop that processes messages
    // sent to our window.  On return from GetMessage,
    // break if EAX contains false and then quit the
    // program.

    forever

        w.GetMessage( msg, NULL, 0, 0 );
        breakif( !eax );
        w.TranslateMessage( msg );
        w.DispatchMessage( msg );

    endfor;

    // Read a newline from the console window in order to pause
    // before quitting this program:

    stdout.put( "Hit 'enter' to quit this program:" );
    stdin.readLn ();

    // The message handling inside Windows has stored
    // the program's return code in the wParam field
    // of the message.  Extract this and return it
    // as the program's return code.

    mov( msg.wParam, eax );
    w.ExitProcess( eax );

end ConsoleHello;
```

6.7.2: Win32 Console Functions

Whether you open your own console window within a GUI application or write a standard console app, Windows provides a large set of console related functions that let you manipulate the console window. Although there are far too many of these functions to describe in detail here, this section will attempt to describe the more useful functions available to console window programmers. For full details on the console window functions, check out the Microsoft document (e.g., on MSDN).

6.7.2.1: w.AllocConsole

```
static
    AllocConsole: procedure;
        @stdcall;
        @returns( "eax" ); // Zero if failure
        @external( "__imp__AllocConsole@0" );
```

The `w.AllocConsole` function will allocate a console for the current application. This function returns zero if the call fails, it returns a non-zero result if the function succeeds. An application may only have one allocated console associated with it. See the description of the `w.FreeConsole` function to learn how to deallocate a console window (so you can allocate another one with a second call to `w.AllocConsole`).

6.7.2.2: w.CreateConsoleScreenBuffer

```
static
    CreateConsoleScreenBuffer: procedure
    (
            dwDesiredAccess:        dword;
            dwShareMode:            dword;
        var lpSecurityAttributes:   SECURITY_ATTRIBUTES;
            dwFlags:                dword;
            lpScreenBufferData:     dword  // Should be NULL.
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__CreateConsoleScreenBuffer@20" );
```

A console window can have multiple screen buffers associated with it, though only one screen buffer can be active at a time (see the description of `w.SetConsoleActiveScreenBuffer` for details on switching console buffers). By default, a console window as a single console buffer associated with it when you first create a console window (via `w.AllocConsole`). However, you may create additional console buffers by calling the `w.CreateConsoleScreenBuffer` API function.

The `dwDesiredAccess` parameter specifies the access rights the current process will have to the buffer. This should be the constant `w.GENERIC_READ | w.GENERIC_WRITE` for full access to the buffer (at the very least, this parameter should have `w.GENERIC_WRITE`, but read access is useful, too).

In general, the `dwShareMode` parameter should be zero. See the Microsoft documentation for more details about file sharing. File sharing isn t a particularly useful feature in console windows, so you ll normally set this parameter to zero.

The `lpSecurityAttribute` parameter should be set to NULL. See the Microsoft documentation if you want to implement secure access to a console window (not very useful, quite frankly).

The `dwFlags` parameter should be set to the value `w.CONSOLE_TEXTMODE_BUFFER` (this is currently the only legal value you can pass as this parameter s value).

The `lpScreenBufferData` parameter must be passed as NULL. Undoubtedly, Microsoft originally intended to use this parameter for something else, and then changed their mind about it s purpose.

If this function fails, it returns zero in the EAX register. If it succeeds, it returns a handle to the console buffer that Windows creates. You will need to save this handle so you can use it to switch to this console buffer using the `w.SetConsoleActiveScreenBuffer` API function.

6.7.2.3: w.FillConsoleOutputAttribute

```
static
    FillConsoleOutputAttribute: procedure
    (
            hConsoleOutput :dword; // Handle to screen buffer
            wAttribute     :dword; // Color attribute (in L.O. 16 bits)
            nLength        :dword; // Number of character positions to fill
            dwWriteCoord   :w.COORD; // Coordinates of first position to fill
        var lpNumberOfAttrsWritten :dword //Returns # of positions written here
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__FillConsoleOutputAttribute@20" );

const
    FOREGROUND_BLUE := $1;
    FOREGROUND_GREEN := $2;
    FOREGROUND_RED := $4;
    FOREGROUND_INTENSITY := $8;
    BACKGROUND_BLUE := $10;
    BACKGROUND_GREEN := $20;
    BACKGROUND_RED := $40;
    BACKGROUND_INTENSITY := $80;

type
    COORD:
        record
            x: word;
            y: word;
        endrecord;
```

A *character attribute* in a console window is the combination of foreground and background colors for that character cell. The `w.FillConsoleOutputAttribute` function lets you set the foreground and background colors for a sequential set of characters on the screen without otherwise affecting the characters on the screen.

The `hConsoleOutput` parameter is the handle of a screen buffer (e.g., a handle returned by `w.CreateConsoleScreenBuffer` or by a call to `w.GetStdHandle(w.STD_OUTPUT_HANDLE)`).

The `wAttribute` value is a bitmap containing some combination of the foreground and background colors defined in the *windows.hhf* header file (included by *w.hhf*). The base foreground colors are `w.FOREGROUND_BLUE`, `w.FOREGROUND_GREEN`, and `w.FOREGROUND_RED`. OR ing these constants together forms additional colors, i.e.,

¥ `w.FOREGROUND_GREEN | w.FOREGROUND_BLUE` produces cyan.

¥ `w.FOREGROUND_BLUE | w.FOREGROUND_RED` produces magenta.

¥ `w.FOREGROUND_GREEN | w.FOREGROUND_RED` produces yellow.

¥ `w.FOREGROUND_GREEN | w.FOREGROUND_BLUE | w.FOREGROUND_REG` produces white.

Merging in w.FOREGROUND_INTENSITY with any of the individual colors or combination of colors produces a lighter (brighter) version of that color. Because you may combine four bits in any combination, it is possible to specify up to 16 different foreground colors with these values.

You set the background colors exactly the same way, except (of course) you use the `w.BACKGROUND_`*xxx* constants rather than the `w.FOREGROUND_`*xxx* constants. Note that you merge both the foreground and background colors into the same `wAttribute` parameter value. Also note that this bitmap consumes only eight of the 32 bits in the `wAttribute` parameter - the other bits must be zero.

The nLength parameter specifies the number of character positions to fill with the specified attribute. If this length would specify a character position beyond the end of the current line, then Windows will wrap around to the beginning of the next line and continue filling the attributes there. If the nLength parameter, along with the dwWriteCoord parameter, specifies a position beyond the end of the current screen buffer, Windows stops emitting attribute values at the end of the current screen buffer.

The dwWriteCoord parameter specifies the starting coordinate in the screen buffer to begin the attribute fill operation. Note that this is a record object consisting of a 16-bit X-coordinate and 16-bit Y-coordinate value pair. Generally, most people simply pass this parameter as a 32-bit value with the X-coordinate value appearing in the L.O. 32 bits of the double word and the Y-coordinate appearing in the upper 32-bits of the double word. You can use HLAs type coercion operation to pass a 32-bit register as this parameter value, e.g.,

```
mov( $1_0002, edx );  // x=2, y=1
w.FillConsoleOutputAttribute
(
   hOutputWindow,
   w.BACKGROUND_GREEN | w.BACKKGROUND_BLUE | w.BACKGROUND_INTENSITY,
   (type w.COORD edx),
   charsWritten
);
```

The last parameter, lpNumberOfAttrsWritten, is a pointer to an integer variable that receives the total number of attribute positions modified in the screen buffer by this function. The w.FillConsoleOutputAttr function stores a value different from nLength into the variable referenced by this address if nLength specifies output positions beyond the end of the screen buffer (forcing w.FillConsoleOutputAttr to write fewer than nLength attribute values to the screen buffer).

You would normally use this function to change the attributes (colors) of existing text on the display; you would not normally use this function to write new data to the screen (the w.SetConsoleTextAttribute function lets you set the default character attributes to use when writing characters to the console). You could use this function, for example, to highlight a line of existing text on the display (perhaps a menu selection) without otherwise affecting the text data.

The w.FillConsoleOutputAttribute function writes only a single foreground/background color combination to the console. If you want to write a sequence of different attribute values to the display, you should consider using the w.WriteConsoleOutputAttribute function, instead.

6.7.2.4: w.FillConsoleOutputCharacter

```
static
    FillConsoleOutputCharacter: procedure
    (
        hConsoleOutput:             dword;
        cCharacter:                 char;
        nLength:                    dword;
        dwWriteCoord:               COORD;
        var lpNumberOfAttrsWritten: dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__FillConsoleOutputCharacterA@20" );
```

The `w.FillConsoleOutputCharacter` function fills a sequence of character cells on the console with a single character value. This function, for example, is useful for clearing a portion of the screen by writing a sequence of space characters (or some other background fill character). This function does not affect the attribute values at each character cell location; therefore, the characters that this function writes to the console will retain the foreground/background colors of the previous characters in those same character cell positions.

This function writes `nLength` copies of the character `cCharacter` to the console screen buffer specified by `hConsoleOutput` starting at the (x,y) coordinate position specified by `dwWriteCoord`. If `nLength` is sufficiently large that it will write characters beyond the end of a line, `w.FillConsoleOutputCharacter` will wrap around to the beginning of the next line. If `w.FillConsoleOutputCharacter` would write characters beyond the end of the screen buffer, then this function ignores the write request for those characters beyond the buffer s end.

6.7.2.5: w.FlushConsoleInputBuffer

```
static
    FlushConsoleInputBuffer: procedure
    (
        hConsoleInput:  dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__FlushConsoleInputBuffer@4" );
```

A call to this function flushes all the input events from the input buffer. Input events include key presses, mouse events, and window resize events. Most console programs ignore mouse and resize events; such programs would call this function to flush any keystrokes from the type ahead buffer.

The `hConsoleInput` handle value is the handle to the console s input buffer. You may obtain the current console input buffer s handle using the API call `w.GetStdHandle(w.STD_INPUT_HANDLE);`

6.7.2.6: w.FreeConsole

```
static
    FreeConsole: procedure;
        @stdcall;
        @returns( "eax" );
        @external( "__imp__FreeConsole@0" );
```

This API call frees (and detaches) the console window associated with the current process (this also closes that console window). Because you can only have one console window associated with an application, you must call this function if you want to create a new window via a `w.AllocConsole` call if the application already has a console window associated with it. Because Windows automatically frees the console window when an application terminates and you can only have one console window active at a time, few programs actually call this function.

6.7.2.7: w.GetConsoleCursorInfo

```
static
    GetConsoleCursorInfo: procedure
    (
            hConsoleOutput:         dword;
```

```
        var lpConsoleCursorInfo:    CONSOLE_CURSOR_INFO
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__GetConsoleCursorInfo@8" );

type
    CONSOLE_CURSOR_INFO:
        record
            dwSize: dword;
            bVisible: dword;
        endrecord;
```

This function returns the size and visibility of the console cursor in a `w.CONSOLE_CURSOR_INFO` data structure. The `dwSize` field of the record is a value in the range 1..100 and represents the percentage of the character cell that the cursor fills. This ranges from an underline at the bottom of the character cell to filling up the entire character cell (at 100%). The `hConsoleOutput` parameter must be the handle of the console s output buffer.

6.7.2.8: w.GetConsoleScreenBufferInfo

```
static
    GetConsoleScreenBufferInfo: procedure
    (
            handle: dword;
        var csbi:   CONSOLE_SCREEN_BUFFER_INFO
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__GetConsoleScreenBufferInfo@8" );

const
    FOREGROUND_BLUE := $1;
    FOREGROUND_GREEN := $2;
    FOREGROUND_RED := $4;
    FOREGROUND_INTENSITY := $8;
    BACKGROUND_BLUE := $10;
    BACKGROUND_GREEN := $20;
    BACKGROUND_RED := $40;
    BACKGROUND_INTENSITY := $80;

type
    CONSOLE_SCREEN_BUFFER_INFO:
        record
            dwSize: COORD;
            dwCursorPosition: COORD;
            wAttributes: word;
            srWindow: SMALL_RECT;
            dwMaximumWindowSize: COORD;
        endrecord;

    SMALL_RECT:
        record
            Left: word;
            Top: word;
            Right: word;
            Bottom: word;
        endrecord;

    COORD:
        record
            x: word;
            y: word;
        endrecord;
```

w.GetConsoleScreenBufferInfo retrieves information about the size of the screen buffer, the position of the cursor in the window, and the scrollable region of the console window. The handle parameter is the handle of the console screen buffer, the csbi parameter is where this function returns its data.

The dwSize field in the w.CONSOLE_SCREEN_BUFFER data structure specifies the size of the console screen buffer. Remember, these values specify the size of the buffer, not the physical size of the console window. Usually, the buffer is larger than the physical window. The dwSize.x field specifies the width of the screen buffer, the dwSize.y field specifies the height of the screen buffer. These values are in character cell coordinates.

The dwCursorPosition field specifies the (x,y) coordinate of the cursor within the screen buffer. Remember, this is the position of the cursor within the screen buffer, not in the console window.

The wAttribute field specifies the default foreground and background colors that Windows will use when you write a character to the display. This field may contain any combination of the w.FOREGROUND_*xxx* and w.BACKGROUND_*xxx* character constants. See the discussion of output attributes in the section on w.FillConsoleOutputAttribute for more details.

The srWindow field is a pair of coordinates that specify the position of the upper left hand corner and the lower right hand corner of the display window within the screen buffer. Note that you can scroll the screen buffer through the display window or even resize the display window by changing the srWindow coordinate values and calling w.SetConsoleScreenBufferInfo.

The dwMaximumWindowSize field specifies the maximum size of the console display window based on the current font in use, the physical screen size, and the screen buffer size.

6.7.2.9: w.GetConsoleTitle

```
static
    GetConsoleTitle: procedure
    (
        var lpConsoleTitle: var;
            nSize:          dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__GetConsoleTitleA@8" );
```

The w.GetConsoleTitle function returns a string containing the name found in the console window s title bar. The lpConsoletitle parameter must be the address of a buffer that will receive a zero-terminated string holding the window s title. The nSize parameter specifies the maximum number of characters (including the zero byte) that w.GetConsoleTitle will transfer. This function returns the actual length of the string (not counting the zero-terminating byte) in the EAX register.

There are two things to note about the lpConsoleTitle parameter. First of all, this is not the address of an HLA string, but the address of a buffer that will receive a zero-terminated string. Second, this is an untyped reference parameter, so HLA will automatically compute the address of whatever object you pass as this parameter.

So if you pass a pointer variable (e.g., a string object), HLA will compute the address of the pointer, not the address of the buffer that the pointer references. You may override this behavior by using the VAL keyword to tell HLA to use the value of the pointer rather than its address. If you want this function to fill in an HLA string variable, you could use code like the following:

```
mov( stringToUse, edx );
w.GetConsoleTitle( [edx], (type str.strRec [edx]).MaxStrLen ) );
mov( stringToUse, edx ); // Windows' API calls don't preserve EDX!
mov( eax, (type str.strRec [edx]).length );
```

This code loads EDX with the address of the first byte of the character data buffer in the string referenced by `stringToUse`. This code passes the address of this character buffer in EDX to the function (brackets are necessary around EDX to tell HLA that this is a memory address rather than just a register value). This function passes the string s maximum length field as the buffer size to the `w.GetConsoleTitle` function.

On return, this function stores the actual string length (returned in EAX) into the HLA string s string length field. The end result is that stringToUse now contains valid HLA string data (the console window s title string).

6.7.2.10: w.GetConsoleWindow

```
static
    GetConsoleWindow: procedure;
        @stdcall;
        @returns( "eax" );
        @external( "__imp__GetConsoleWindow@0" );
```

The `w.GetConsoleWindow` function returns the window handle associated with the application s console window. Note that there is a big difference between a console screen buffer handle and a console window handle. The window handle is the same type of handle that GUI apps use when painting to the screen. Screen buffer handles are the handles you pass to console function to manipulate data in the screen buffer.

6.7.2.11: w.GetNumberOfConsoleInputEvents

```
static
    GetNumberOfConsoleInputEvents: procedure
    (
            hConsoleInput:      dword;
        var lpcNumberOfEvents:  dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__GetNumberOfConsoleInputEvents@8" );
```

The `w.GetNumberOfConsoleInputEvents` returns the total number of pending keystroke, mouse, or resize input events sent to the console window. This function stores the count at the address specified by the `lpcNumberOfEvents` parameter (note that this function returns success/failure in EAX, it does not return the count in EAX). There is, unfortunately, no way to directly check to see if there are any keystrokes sitting in the input queue. However, if w.GetNumberOfConsoleInputEvents tells you that there are pending input events, you can use w.PeekConsoleInput to determine if the input event at the front of the queue is keystroke waiting to be read. If not, you can call w.ReadConsolInput to remove that first input event from the queue. You can repeat the get-number/peek/read sequence until you empty the input buffer or you peek at a keyboard event. By this processes, you can determine if there is at least one keyboard input event in the queue.

6.7.2.12: w.GetStdHandle

```
static
    GetStdHandle: procedure
    (
        nStdHandle:dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__GetStdHandle@4" );
const
    STD_INPUT_HANDLE := -10;
    STD_OUTPUT_HANDLE := -11;
    STD_ERROR_HANDLE := -12;
```

This function returns the current handle for the standard input, standard output, or standard error devices. The single parameter (`nStdHandle`) is one of the values `w.STD_INPUT_HANDLE`, `w.STD_OUTPUT_HANDLE`, or `w.STD_ERROR_HANDLE`, depending on which of these functions you want to retrieve. By default (meaning you ve not redirected the standard input, standard output, or standard error devices), the standard output and standard error handles will be the same as the output screen buffer handle. The standard input, by default, will be the handle of the console s input buffer.

6.7.2.13: w.PeekConsoleInput

```
static
    PeekConsoleInput: procedure
    (
            hConsoleInput:          dword;
        var lpBuffer:               INPUT_RECORD;
            nLength:                dword;
        var lpNumberOfEventsRead:   dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__PeekConsoleInputA@16" );

const
    KEY_EVENT := $1;
    mouse_eventC := $2;
    WINDOW_BUFFER_SIZE_EVENT := $4;
    MENU_EVENT := $8;
    FOCUS_EVENT := $10;

type
    INPUT_RECORD:
        record
            EventType:word;
            Event:
                union
                    KeyEvent                :KEY_EVENT_RECORD;
                    MouseEvent              :MOUSE_EVENT_RECORD;
                    WindowBufferSizeEvent   :WINDOW_BUFFER_SIZE_RECORD;
                    FocusEvent              :FOCUS_EVENT_RECORD;
                endunion;
        endrecord;

    KEY_EVENT_RECORD:
        record
            bKeyDown: dword;
            wRepeatCount: word;
            wVirtualKeyCode: word;
            wVirtualScanCode: word;
            _uChar: CHARTYPE;
            dwControlKeyState: dword;
        endrecord;

    MOUSE_EVENT_RECORD:
        record
            dwMousePosition: COORD;
            dwButtonState: dword;
            dwControlKeyState: dword;
            dwEventFlags: dword;
        endrecord;

     WINDOW_BUFFER_SIZE_RECORD:
        record
            dwSize: COORD;
       endrecord;

    FOCUS_EVENT_RECORD:
        record
            bSetFocus: dword;
        endrecord;
```

The w.PeekConsoleInput function reads one or more input records (if available) from the console s input queue without removing those records from the input queue. You may use the w.GetNumberOfConsoleInputEvents to determine how many pending events there are, allocate an array of w.INPUT_RECORD objects with the number of elements returned by w.GetNumberOfConsoleInputEvents, and then call w.PeekConsoleInput to read all of those events into the array you ve allocated. Then you can scan through the array at your leisure to determine what type of input records are in the buffer.

The hConsoleInput parameter is the handle of the console input buffer you want to peek at. The lpBuffer parameter is a pointer to one or more w.INPUT_RECORD objects that will hold the returned records; note that you may specify a pointer, a scalar variable, or an array variable here, HLA will do the right thing with your parameter. The nLength parameter specifies the number of array elements you re passing via the lpBuffer parameter. If you re only passing a scalar w.INPUT_BUFFER object as the lpBuffer parameter, you should specify an nLength value of one. Note that the nLength parameter specifies the maximum number of input records that w.PeekConsoleInput will read; if there are fewer records than this number specifies, w.PeekConsoleInput will only read those records that are available. The lpNumberOfEventsRead parameter is a pointer to a 32-bit integer variable where w.PeekConsoleInput will store the actual number of events read from the input queue. This value will always be less than or equal to the value you pass in nLength. Note that it is perfectly possible (and reasonable) for w.PeekConsoleInput to read zero input records (if there are no pending input records) and return zero in the variable referenced by lpNumberOfEventsRead.

The lpBuffer parameter to w.PeekConsoleInput must contain the address of a w.INPUT_EVENT record. The first field of this record is the EventType field that specifies the type of the event that the record holds. This field will contain one of the following values: w.KEY_EVENT, w.mouse_eventC, w.WINDOW_BUFFER_SIZE_EVENT, w.MENU_EVENT, or w.FOCUS_EVENT. Yes, w.mouse_eventC is spelled funny and doesn t match the Window s documentation. This is one of those rare cases where Windows reused an identifier by simply changing the case of the identifier, thus creating a case conflict in HLA. The solution was to rename Windows s constant MOUSE_EVENT as w.mouse_eventC. The value of EventType determines whether the Event union s data is a w.KEY_EVENT_RECORD, w.MOUSE_EVENT_RECORD, w.WINDOW_BUFFER_SIZE_RECORD, or a w.FOCUS_EVENT_RECORD type.

If Event contains the value w.KEY_EVENT then the KeyEvent field of the w.INPUT_RECORD union contains data representing a keyboard event. The fields of the w.KEY_EVENT_RECORD data structure have the following meanings:

¥ bKeyDown contains true (non-zero value) if the key was just pressed, false (0) if it was just released.

¥ wRepeatCount may contain a repeat count for the key saying that the key auto-repeated this many times. You must treat this as n keypresses of the current key code.

¥ wVirtualKeyCode specifies a Windows virtual keycode for this particular keypress. See the appendices for a list of the Windows virtual keycodes.

¥ wVirtualScanCode specifies an OEM/Hardware dependent scan code for this particular keystroke. Most applications should ignore this value.

¥ uChar is the translated ASCII/ANSI character code for this keystroke. Keystrokes that do not have ANSI character code equivalents generally return zero here and you have to use the wVirtualKeyCode field to determine which key was pressed (or released).

¥ dwControlKeyState reports the state of the modifier keys when this key event occurred. This field is a bit map with a set bit indicating that a particular modifier key (e.g., control or shift) was held down. Here are the values associated with the various modifier keys on a PC s keyboard:

```
w.RIGHT_ALT_PRESSED    $1
w.LEFT_ALT_PRESSED     $2
w.RIGHT_CTRL_PRESSED   $4
w.LEFT_CTRL_PRESSED    $8
w.SHIFT_PRESSED        $10
w.NUMLOCK_ON           $20
w.SCROLLLOCK_ON        $40
w.CAPSLOCK_ON          $80
w.ENHANCED_KEY         $100
```

The w.ENHANCED_KEY modifier specifies that the user has pressed an enhanced key on a keyboard. Enhanced keys are the INS, DEL, HOME, END, PAGEUP, PAGE DN, and arrow keys that are separate from the numeric keypad. The slash (/) and ENTER keys on the numeric keypad are also enhanced keys.

If Event contains the value w.MOUSE_EVENT then the KeyEvent field of the w.INPUT_RECORD union contains data representing a mouse event. The fields of the w.MOUSE_EVENT_RECORD data structure have the following meanings:

¥ dwMousePosition specifies the location of the mouse cursor in screen buffer character cell coordinates. Note that the Y-coordinate appears in the H.O. word and the X-coordinate appears in the L.O. word of the w.COORD data type.

¥ dwButtonState is a bitmap specifying the state of the buttons on the mouse. The w.hhf header files defines the following constants specifying button positions with in the bitmap:

```
w.FROM_LEFT_1ST_BUTTON_PRESSED  $1
w.RIGHTMOST_BUTTON_PRESSED      $2
w.FROM_LEFT_2ND_BUTTON_PRESSED  $4
w.FROM_LEFT_3RD_BUTTON_PRESSED  $8
w.FROM_LEFT_4TH_BUTTON_PRESSED  $10
```

If a mouse supports more than five buttons, then the mouse driver will, undoubtedly, set additional H.O. bits in this bitmap.

¥ `dwControlKeyState` is a bitmap specifying the current state of the modifier keys on the keyboard when this mouse event occurred. See the description of `dwControlKeyState` in the description of `w.KEY_EVENT_RECORD` for more details.

¥ dwEventFlags specifies the type of mouse event that has occurred. If this field contains zero, it means that a mouse button has been pressed or released. If this field is not zero, it contains one of the following values:

```
w.MOUSE_MOVED    $1
w.DOUBLE_CLICK   $2
```

Console applications should ignore `w.WINDOW_BUFFER_SIZE_RECORD` and `w.FOCUS_EVENT` as these are intended for internal use by Windows.

6.7.2.14: w.ReadConsole

```
static
    ReadConsole: procedure
    (
            hConsoleInput:          dword;
        var lpBuffer:               var;
            nNumberOfCharsToRead:   dword;
        var lpNumberOfCharsRead:    dword;
        var lpReserved:             var
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__ReadConsoleA@20" );
```

The `w.ReadConsole` function reads keyboard events from the console input buffer. The `hConsoleInput` parameter specifies the console input buffer to read from (this should be the standard input handle). The `lpBuffer` parameter is the address of a character array (buffer) that will receive the characters read from the console input buffer. The `nNumberOfCharsToRead` parameter specifies the number of keystrokes that will be read from the input buffer. By default, `w.ReadConsole` will return when the user types this many characters at the keyboard or when the user presses the enter key (prior to typing this many characters). Please see the Windows documentation for details on the `w.GetConsoleMode` and `w.SetConsoleMode` functions that let you specify Windows behavior with respect to console input. the `lpNumberOfCharactersRead` is a pointer to an integer variable; Windows will store the actual number of characters read from the keyboard into this integer variable. The `lpReserved` field is reserved and you must pass NULL in this parameter.

If the input buffer contains records other than keyboard events, `w.ReadConsole` removes those events from the buffer and ignores them. If you want to be able to read mouse events, use `w.ReadConsoleInput` instead of `w.ReadConsole`. Also note that `w.ReadConsole` is virtually the same as the `w.ReadFile` function. The pure Windows version of `w.ReadConsole` allows you to read Unicode characters, but the HLA prototype for `w.Read-`

`Console` only reads ANSI characters (there is nothing stopping you from creating your own prototype to call the actual Windows `ReadConsole` API function, though).

6.7.2.15: w.ReadConsoleInput

```
static
    ReadConsoleInput: procedure
    (
            hConsoleInput:          dword;
        var lpBuffer:               INPUT_RECORD;
            nLength:                dword;
        var lpNumberOfEventsRead:   dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__ReadConsoleInputA@16" );
```

This function reads console input records from the console input buffer. This reads both keyboard and mouse events and stores them into a `w.INPUT_RECORD` array whose address you pass to this function.

The `hConsoleInput` parameter is the console input handle (e.g., the standard input handle). The `lpBuffer` parameter is the address of a `w.INPUT_RECORD` structure or an array of these structures (see the description of this data structure in the section on `w.PeekConsoleInput`). The `nLength` parameter specifies the number of input records to read from the console input queue. As with `w.ReadConsole`, Windows may actually read fewer than this number of input records if the console mode is set to return when the user presses the Enter key. In any case, `w.ReadConsoleInput` will store the actual number of input records at the address specified by the `lpNumberO-fEventsRead` parameter. See the description of `w.ReadConsole` for more details. Also see the discussion of `w.PeekConsoleInput` for a discussion of the `w.INPUT_RECORD` type.

6.7.2.16: w.ReadConsoleOutput

```
static
    ReadConsoleOutput: procedure
    (
            hConsoleOutput: dword;
        var lpBuffer:       CHAR_INFO;
            dwBufferSize:   COORD;
            dwBufferCoord:  COORD;
        var lpReadRegion:   SMALL_RECT
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__ReadConsoleOutputA@20" );

type
    COORD:
        record
            x: word;
            y: word;
        endrecord;

    SMALL_RECT:
        record
```

```
            Left    :word;
            Top     :word;
            Right   :word;
            Bottom  :word;
         endrecord;

    CHARTYPE:
        union
            UnicodeChar :word;  // Note: HLA's prototype only returns ASCII characters.
            AsciiChar   :byte;
        endunion;

    CHAR_INFO:
        record
            CharVal    :CHARTYPE;
            Attributes :word;
        endrecord;
```

The `w.ReadConsoleOutput` function reads a rectangular region off the screen and stores the character data into a two-dimensional array of characters. The `hConsoleOutput` parameter is the handle of the console s output screen buffer (i.e., the standard output handle). The `lpBuffer` parameter is a pointer to a two-dimensional array of characters that will receive the character data read from the screen buffer. The `dwBufferSize` parameter specifies the size of the buffer that will receive the characters; the L.O. word (`x`) specifies the number of columns in the array, the H.O. word (`y`) specifies the number of rows in the array. The `dwBufferCoord` parameter specifies the (x,y) coordinates (in character cell positions) of the upper-leftmost character in the array specified by `lpBuffer` where `w.ReadConsoleOutput` will begin storing the character data. On entry, the values pointed at by the `lpReadRegion` parameter specify the upper left hand corner and the lower right hand corner of a rectangular region in the screen buffer to copy into the buffer specified by `lpBuffer`. On exit, the `w.ReadConsoleOutput` routine stores the actual coordinates used by the copy operation. The input and output values will be different if the input values exceeded the boundaries of the screen buffer (in which case, `w.ReadConsoleOutput` truncates the coordinates so that they remain within the screen buffer).

6.7.2.17: w.ReadConsoleOutputAttribute

```
static
    ReadConsoleOutputAttribute: procedure
    (
            hConsoleOutput:         dword;
        var lpAttribute:            word;
            nLength:                dword;
            dwReadCoord:            COORD;
        var lpNumberOfAttrsRead:    dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__ReadConsoleOutputAttribute@20" );

type
    COORD:
        record
            x: word;
            y: word;
        endrecord;
```

This function is very similar to `w.ReadConsoleOutput` except it copies attribute values rather than characters to the `lpAttribute` array. The `lpAttribute` parameter should hold the address of a word array than contains at least `nLength` elements. The `dwReadCoord` specifies the coordinate in the screen console buffer where `w.ReadConsoleOutputtAttribute` begins reading attributes (the H.O. word holds the y-coordinate, the L.O. word holds the x-coordinate). The `lpNumberOfAttrsRead` parameter holds the address of an integer where `w.ReadConsoleOutputAttribute` will store the number of attribute values it reads from the screen buffer. The value this function returns will be less than `nLength` if the function attempts to read data beyond the end of the screen buffer.

Unlike `w.ReadConsoleOutput`, `w.ReadConsoleOutputAttribute` does not read a rectangular block of data into a two-dimensional array. Instead, this function simply reads a linear block of attributes from the screen buffer starting at the specified coordinate. If this function reaches the end of the current line and `nLength` specifies additional attributes to read, this function returns the attributes beginning at the start of the next line. It stops, however, at the end of the screen buffer.

6.7.2.18: w.ReadConsoleOutputCharacter

```
static
    ReadConsoleOutputCharacter: procedure
    (
            hConsoleOutput:         dword;
        var lpCharacter:            char;
            nLength:                dword;
            dwReadCoord:            COORD;
        var lpNumberOfCharsRead:    dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__ReadConsoleOutputCharacterA@20" );

type
    COORD:
        record
            x: word;
            y: word;
        endrecord;
```

The `w.ReadConsoleOutputCharacter` function is like `w.ReadConsoleOutput` except it reads a linear sequence of characters from the screen buffer (like `w.ReadConsoleOutputAttribute` reads attributes) from the buffer rather than reading a rectangular block of characters. The `hConsoleOutput` parameter specifies the handle for the screen output buffer (i.e., the standard output handle). The `lpCharacter` parameter holds the address of the buffer to receive the characters; it should have room for at least `nLength` characters (the third parameter, that specifies the number of characters to read). The `dwReadCoord` parameter specifies the character cell coordinate in the screen buffer where this function begins reading characters. This function returns the number of characters actually read in the integer whose address you pass in the `lpNumberOfCharsRead` parameter; this number will match `nLength` unless you attempt to read beyond the end of the screen buffer.

6.7.2.19: w.ScrollConsoleScreenBuffer

```
static
```

```
    ScrollConsoleScreenBuffer: procedure
    (
            hConsoleOutput:         dword;
        var lpScrollRectangle:      SMALL_RECT;
        var lpClipRectangle:        SMALL_RECT;
            dwDestinationOrigin:    COORD;
        var lpFill:                 CHAR_INFO
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__ScrollConsoleScreenBufferA@20" );

type
    COORD:
        record
            x: word;
            y: word;
        endrecord;

    CHARTYPE:
        union
            UnicodeChar :word;  // Note: HLA's prototype only returns ASCII characters.
            AsciiChar   :byte;
        endunion;

    CHAR_INFO:
        record
            CharVal    :CHARTYPE;
            Attributes :word;
        endrecord;

    SMALL_RECT:
        record
            Left    :word;
            Top     :word;
            Right   :word;
            Bottom  :word;
         endrecord;
```

The `w.ScrollConsoleScreenBuffer` function scrolls a rectangular region of a screen buffer. This function achieves this by moving a source rectangular region of the screen buffer to some other point in the screen buffer. The `lpScrollRectangle` parameter specifies the source rectangle (via the coordinates of the upper left hand and lower right hand corners of the rectangle). The `dwDestinationOrigin` parameter specifies the upper left hand corner of the target location where this function will move the source rectangle. Any locations in the source rectangle that do not overlap with the source rectangle wind up getting filled with the character and attribute specified by the `lpFill` parameter. The `lpClipRectangle` parameter specifies a clipping rectangle. Actual scrolling only takes place within this clipping rectangle. For example, if you wanted to scroll the screen up one line, you would set the `lpScrollRectangle` parameter to encompass the whole screen (say, 80x25), set the destination origin to (0,-1), and then set the `lpClipRectangle` to the same values as the `lpScrollRectangle` value. You may also pass NULL as the value for `lpClipRectangle`, in which case `w.ScrollConsoleScreenBuffer` assumes that the clip region is equal to the source region (`lpScrollRectangle`).

6.7.2.20: w.SetConsoleActiveScreenBuffer

```
static
    SetConsoleActiveScreenBuffer: procedure
    (
        hConsoleOutput: dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__SetConsoleActiveScreenBuffer@4" );
```

This function displays the screen buffer whose handle you pass as the parameter in the console window. Although an application may have only one console window, it may have multiple screen buffers that it can display in that window. When the program first begins execution, the handle associated with the default console screen buffer is the standard output handle. You may create new screen buffers (and obtain their handles) via the `w.CreateConsoleScreenBuffer` call and then you can switch to these screen buffers using the `w.SetConsoleActiveScreenBuffer` API invocation.

6.7.2.21: w.SetConsoleCursorInfo

```
static
    SetConsoleCursorInfo: procedure
    (
            hConsoleOutput:         dword;
        var lpConsoleCursorInfo:    CONSOLE_CURSOR_INFO
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__SetConsoleCursorInfo@8" );

type
    CONSOLE_CURSOR_INFO:
        record
            dwSize    :dword;
            bVisible  :dword;
        endrecord;
```

The `w.SetConsoleCursorInfo` function lets you specify the size of the cursor in the console window and whether the cursor is visible or invisible. The `hConsoleOutput` parameter is the screen buffer handle of the screen buffer to which you wish the operation to apply. The `lpConsoleCursorInfo` parameter is a pointer to a `w.CONSOLE_CURSOR_INFO` data structure whose two fields (`dwSize` and `bVisible`) specify the size of the cursor and whether or not it is visible. The dwSize field must contain a value between 1 and 100 and specifies the percentage of a character cell that the cursor will fill. The cursor fills the cell from the bottom of the cell to the top, so a 1 would produce a cursor that is just an underline, 100 would produce a cursor that completely fills the character cell, and 50 would produce a cursor that fills the bottom half of the character cell. The `bVisible` field specifies whether the cursor will be visible in the console window. If this field contains true, then the cursor will be visible, if it contains false then Windows will not display the cursor.

6.7.2.22: w.SetConsoleCursorPosition

```
static
```

```
    SetConsoleCursorPosition: procedure
    (
        hConsoleOutput:     dword;
        dwCursorPosition:   COORD
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__SetConsoleCursorPosition@8" );

type
    COORD:
        record
            x: word;
            y: word;
        endrecord;
```

This function positions the cursor at the coordinate specified by the `dwCursorPosition` parameter. The `hConsoleOutput` parameter specifies the handle of the console screen buffer to which the cursor movement operation applies. If this function moves the cursor to a point in the screen buffer that is not displayed in the console window, then Windows will adjust the window to make the character cell specified by `dwCursorPosition` visible in the window.

6.7.2.23: w.SetConsoleScreenBufferSize

```
static
    SetConsoleScreenBufferSize: procedure
    (
        hConsoleOutput: dword;
        dwSize:         COORD
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__SetConsoleScreenBufferSize@8" );

type
    COORD:
        record
            x: word;
            y: word;
        endrecord;
```

The `w.SetConsoleScreenBufferSize` sets the size of the screen buffer. The `hConsoleOutput` parameter is the handle of the screen buffer whose size you want to change, the `dwSize` parameter specifies the new size of the buffer (the `dwSize.x` field specifies the new width, the `dwSize.y` field specifies the new height). This function fails if the screen buffer size (in either dimension) is smaller than the size of the console window or less than the minimum specified by the system (see the `w.GetSystemMetrics` call to find the minimally permissible size).

6.7.2.24: w.SetConsoleTextAttribute

```
static
    SetConsoleTextAttribute: procedure
    (
        hConsoleOutput: dword;
        wAttributes:    word
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__SetConsoleTextAttribute@8" );

const
    FOREGROUND_BLUE := $1;
    FOREGROUND_GREEN := $2;
    FOREGROUND_RED := $4;
    FOREGROUND_INTENSITY := $8;
    BACKGROUND_BLUE := $10;
    BACKGROUND_GREEN := $20;
    BACKGROUND_RED := $40;
    BACKGROUND_INTENSITY := $80;
```

The `w.SetConsoleTextAttribute` function lets you specify the default text attribute that Windows will use when writing text to the console window (e.g., via the HLA Standard Library stdout.put call). The hConsoleOutput parameter is the handle of the screen buffer whose default attribute you wish to set; the wAttributes parameter is the attribute value. This is a bitmap of color values created by combining the `w.FOREGROUND_`*`xxx`* and `w.BACKGROUND_`*`xxx`* values, e.g.,

w.SetConsoleTextAttribute(hndl, w.FOREGROUND_RED | w.BACKGROUND_BLUE);

6.7.2.25: w.SetConsoleTitle

```
static
    SetConsoleTitle: procedure
    (
        lpConsoleTitle: string
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__SetConsoleTitleA@4" );
```

The `w.SetConsoleTitle` function copies the string you pass as a parameter (`lpConsoleTitle`) to the title bar of the console window.

6.7.2.26: w.SetConsoleWindowInfo

```
static
    SetConsoleWindowInfo: procedure
    (
            hConsoleOutput:     dword;
            bAbsolute:          dword;
        var lpConsoleWindow:    SMALL_RECT
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__SetConsoleWindowInfo@12" );

type
    SMALL_RECT:
        record
            Left    :word;
            Top     :word;
            Right   :word;
            Bottom  :word;
         endrecord;
```

The `w.SetConsoleWindowInfo` function sets the size and position of the console s window. The `hConsoleOutput` parameter specifies the handle of a screen buffer (to which this operation applies). The `bAbsolute` parameter is a boolean value (true/1 or false/0) that specifies whether the coordinates you re supplying are relative to the current coordinates of the window (that is, relative to the current upper left hand corner of the window within the screen buffer) or absolute (specifies the coordinates of the upper left hand corner of the window within the screen buffer). If `bAbsolute` is true, then the coordinates are absolute and specify a character cell position within the screen buffer for the upper-left hand corner of the window. Thee lpConsoleWindow parameter is the address of a `w.SMALL_RECT` structure that specifies the new origin and size of the console display window. This function fails if the new window size would exceed the boundaries of the screen buffer.

You may use this function to scroll through vertically the screen buffer by adjusting the `Top` and `Bottom` fields of the `w.SMALL_RECT` record by the same amount. Similarly, you can use this function to scroll horizontally through the screen buffer by adjusting the `Left` and `Right` fields by the same amount.

6.7.2.27: w.SetStdHandle

```
static
    SetStdHandle: procedure
    (
        nStdHandle: dword;
        hHandle:    dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__SetStdHandle@8" );

const
    STD_INPUT_HANDLE := 10;
    STD_OUTPUT_HANDLE := 11;
    STD_ERROR_HANDLE := 12;
```

This function redirects the standard input, standard output, or standard error devices to a handle you specify. The first parameter is one of the constant values `w.STD_INPUT_HANDLE`, `w.STD_OUTPUT_HANDLE`, or `w.STD_ERROR_HANDLE` that specifies which of the standard handles you want to redirect. The `hHandle` parameter is a file handle that specifies where you want the output redirected. For example, you may redirect the standard output to a specific console screen buffer by specifying that screen output buffer s handle.

6.7.2.28: w.WriteConsole

```
static
    WriteConsole: procedure
    (
            hConsoleOutput:         dword;
        var lpBuffer:               var;
            nNumberOfCharsToWrite:  dword;
        var lpNumberOfCharsWritten: dword;
        var lpReserved:             var
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__WriteConsoleA@20" );
```

The `w.WriteConsole` function writes a string to the console screen buffer specified by the `hConsoleOutput` handle. The `lpBuffer` parameter specifies the address of the character data to write to the console buffer. The `nNumberOfCharsToWrite` parameter specifies the character count. This function returns the actual number of characters written in the integer variable whose address you pass in the `lpNumberOfCharsWritten` parameter. The `lpReserved` parameter is reserved and you must pass NULL for its value.

Note that Windows actually provides two versions of this function - one for ASCII characters and one for Unicode characters. The HLA prototype specifies the ASCII version. If you need to call the Unicode version, you will need to create your own prototype for that function. Because the HLA version calls the ASCII version, this function is virtually identical to a w.WriteFile call (e.g., stdout.put).

6.7.2.29: w.WriteConsoleInput

```
static
    WriteConsoleInput: procedure
    (
            hConsoleInput:              dword;
        var lpBuffer:                   INPUT_RECORD;
            nLength:                    dword;
        var lpNumberOfEventsWritten:    dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__WriteConsoleInputA@16" );
```

The `w.WriteConsoleInput` function writes one or more `w.INPUT_RECORD` structures to the console s input buffer. This function places any input records written at the end of the buffer, behind any pending input events. You may use this function to simulate keyboard or mouse input.

The `hConsoleInput` parameter specifies the handle of the console input buffer. You could, for example, supply the handle of the standard input device here. The lpBuffer parameter is a pointer to an array of one or more `w.INPUT_RECORD` objects (please see the discussion of input records in the section on the `w.ReadConsoleInput` function). The `nLength` parameter is the number of input records in the array that `lpBuffer` points at that you want this function to process. This function stores the number of input records processed in the integer variable whose address you pass in the `lpNumberOfEventsWritten` record.

6.7.2.30: w.WriteConsoleOutput

```
static
    WriteConsoleOutput: procedure
    (
            hConsoleOutput: dword;
        var lpBuffer:       CHAR_INFO;
            dwBufferSize:   COORD;
            dwBufferCoord:  COORD;
        VAR lpWriteRegion:  SMALL_RECT
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__WriteConsoleOutputA@20" );

type
    COORD:
        record
            x: word;
            y: word;
        endrecord;

    SMALL_RECT:
        record
            Left    :word;
            Top     :word;
            Right   :word;
            Bottom  :word;
         endrecord;

    CHARTYPE:
        union
            UnicodeChar :word;  // Note: HLA's prototype only returns ASCII characters.
            AsciiChar   :byte;
        endunion;

    CHAR_INFO:
        record
            CharVal    :CHARTYPE;
            Attributes :word;
        endrecord;
```

The `w.WriteConsoleOutput` function writes a rectangular block of character and attribute data to a screen buffer. The `hConsoleOutput` parameter is the handle of the screen buffer where this function is to write the data. The `lpBuffer` parameter is the address of an array of character/attribute values that this function is to write to the screen. Note that each character/attribute element in `lpBuffer` consumes 32-bits (16 bits for the attribute and 16 bits for the character). The `dwBufferSize` parameter specifies the of the character/attribute data at which `lpBuffer` points; the `dwBufferSize.x` field specifies the number of columns, the `dwBufferSize.y` field specifies the number of rows of character/attribute data in the buffer area. Note that this parameter specifies the size of the source buffer, not the size of the data to write; the actual data written may be a subset of this two-dimensional array. The `dwBufferCoord` parameter specifies the coordinate of the upper-left hand corner of the rectangular area to write within the source buffer. The `lpWriteRegion` parameter is the address of a `w.SMALL_RECT` data structure; on input, these record specifies the region of the screen buffer where Windows is to write the data from the source buffer. When Windows returns, it writes the size of the actual rectangle written into this record structure (the actual rectangle may be smaller if the source rectangle s size violates the boundaries of the screen buffer).

6.7.2.31: w.WriteConsoleAttribute

```
static
    WriteConsoleOutputAttribute: procedure
    (
            hConsoleOutput:         dword;
        var lpAttribute:            word;
            nLength:                dword;
            dwWriteCoord:           COORD;
        var lpNumberOfAttrsWritten: dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__WriteConsoleOutputAttribute@20" );

type
    COORD:
        record
            x: word;
            y: word;
        endrecord;
```

The `w.WriteConsoleAttribute` function writes a linear sequence of attribute values to the console without affecting the character data at the cell positions where it writes those attributes. The `hConsoleOutput` parameter specifies the handle of a console screen buffer. The `lpAttribute` parameter is the address of an array of one or more 16-bit attribute values. The `nLength` parameter specifies the number of attribute values to write to the screen. The `dwWriteCoord` parameter specifies the (`x,y`) coordinate of the screen buffer where this function is to begin writing the attribute values (the `dwWriteCoord.x` field appears in the L.O. word and the `dwWriteCoord.y` field appears in the H.O. word of this double word parameter value). The `lpNumberOfAttrsWritten` parameter is the address of an integer variable that will receive the number of attributes actually written to the screen buffer. This number will be less than `nLength` if the function attempts to write data beyond the end of the screen buffer.

6.7.2.32: w.WriteConsoleOutputCharacter

```
static
    WriteConsoleOutputCharacter: procedure
    (
            hConsoleOutput:         dword;
            lpCharacter:            string;
            nLength:                dword;
            dwWriteCoord:           COORD;
        var lpNumberOfCharsWritten: dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__WriteConsoleOutputCharacterA@20" );

type
    COORD:
        record
            x: word;
            y: word;
        endrecord;
```

The `w.WriteConsoleOutputCharacter` function writes a linear sequence of attribute values to the console without affecting the character data at the cell positions where it writes those attributes. The `hConsoleOutput` parameter specifies the handle of a console screen buffer. The `lpCharacter` parameter is the address of an array of one or more character values. The `nLength` parameter specifies the number of character values to write to the screen. The `dwWriteCoord` parameter specifies the (`x,y`) coordinate of the screen buffer where this function is to begin writing the character values (the `dwWriteCoord.x` field appears in the L.O. word and the `dwWriteCoord.y` field appears in the H.O. word of this double word parameter value). The `lpNumberOfCharsWritten` parameter is the address of an integer variable that will receive the number of attributes actually written to the screen buffer. This number will be less than `nLength` if the function attempts to write data beyond the end of the screen buffer.

Note that Windows actually provides two versions of this function, one for Unicode and one for ASCII character data. The HLA Standard Library function prototypes the ASCII version. If you need the Unicode version, you can easily prototype that function yourself.

6.7.2.33: Plus More!

There are many, many, additional console related functions beyond the more common ones presented here. If you re interested in high-performance console application programming under Windows, you ll want to read the API documentation appearing on the accompanying CD-ROM or read up on console I/O on Microsoft s MSDN system.

6.7.3: HLA Standard Library Console Support

The HLA Standard Library provides a small console library module that simulates an ANSI terminal. Although this set of library functions provides but a small set of available Win32 console features, it is slightly easier to use for the more common operations (like cursor positioning, clearing portions of the screen, and things like that). Another advantage to the HLA Standard Library Console module is that it is portable - console applications that use this console module will compile and run, unchanged, under Linux. However, because this is a book on Windows programming, we won t bother exploring this portability issue any farther.

There is also a set of obsolete console library functions available with in the HLA Examples code (available on the accompanying CD-ROM or via download at http://webster.cs.ucr.edu). This set of library routines provided a much stronger set of console capabilities than the current HLA Standard Library (at lot of functionality was removed in order to achieve portability to Linux). These older functions are mostly wrappers around Win32 console API calls that make them more convenient to use. They re also great examples of calls to the Win32 API functions. The major drawback to this source code is that it is quite old - hailing from the days when HLA was missing several important features that make interfacing with Windows a lot easier; so the source code makes the calls to the Win32 API functions in some peculiar ways.

6.8: This Isn’t the Final Word!

There is quite a bit more to text processing under Windows than a even a long chapter like this one can present. However, most people who are reading this text probably want to learn how to write GUI applications involving graphics, so it s time to set the discussion of text down for a while and start looking at how to do graphics in a Windows environment.