Chapter 5: The Event-Oriented Programming Paradigm

5.1: Who Dreamed Up This Nonsense?

A typical programmer begins their programming career in a beginning programming course or by learning programming on their own from some book that teaches them step-by-step how to write simple programs. In almost every case, the programming tutorial the student uses begins with a program not unlike the following:

```
#include <stdio.h>
int main( int argc, char **argv )
{
   printf( “Hello world\n” );
}
```

This program seems so quaint and simple, but keep in mind that there is considerable education and effort needed to get this trivial program running. A beginning programmer has to learn how to use the computer, an editor, various operating system commands, how to invoke the compiler (and possibly linker), and how to execute the resulting executable file. Though the (C language) program code may be quite trivial, the steps leading up to the point where the student can compile and run this simple program are not so trivial. It should come as no surprise then, that the programming projects that immediately follow the “hello world” program build upon the lessons before them.

One thing that quickly becomes taken for granted by a programming student’s “programmer’s view of the world” is that there is some concept of a *main program* that takes control of the CPU when the program first starts running and this main program drives the application. The main program calls the various functions and subroutines that make up the application and, most importantly, the main program (or functions/subroutines that the main program calls) makes requests for operating system services via calls to the OS, which return once the OS satisfies that service request. OS calls are always *synchronous*; that is, you call an OS function and when it completes, it returns control back to your program. In particular, the OS doesn’t simply call some function within your program without you explicitly expecting this to happen. In fact, in the normal programming paradigm the OS never calls a function in your program at all - it simply returns to your program *after you’ve called it*.

In the Windows operating system, the concept where the user’s application has control and calls the operating system when it needs some service done (like reading a value from the standard input device) is tossed out the window (pardon the pun). Instead, the OS takes control of the show. It is, effectively, the “main program” that tracks events throughout the system and calls functions within various applications as Windows accumulates events (like keypresses or mouse button presses) that it feels the application needs to service. This completely changes the way one writes a program from the application programmer’s perspective. Before, the application was in control and knew when things were happening in the application (mainly by virtue of where the program was executing at any given time). In the Windows programming paradigm, however, the OS can arbitrarily call any function in the application (well, not really, but you’ll see how this actually works in a little bit) without the application explicitly requesting that the OS call that function. This makes writing applications quite a bit more complex because any of a set of functions could be called at any one given time - the application has no way of predicting the order of invocation. Furthermore, convenient OS facilities like “read a line of text from the keyboard” or “read an integer value from the keyboard” simply don’t exist. Instead, the OS calls some function in your code every time the user presses a key on the keyboard. Your code has to save up each keystroke and decide when it has read a full line of text from the keyboard (or when it has read a complete integer value, at which time the application must convert the string to an integer value and pass it on to whatever section of the application needed the integer value). Perhaps even more frustrating is the fact that a Windows GUI application cannot even output data whenever it wants to. Instead, the application needs to save up any output it wishes to display and wait for Windows to send it an event saying “Okay, now update the display screen.” The days of slipping in a quick “printf” statement (or something comparable in whatever language you’re using) are long gone. Even worse, most programmers learned to write software in an environment where the program is doing only one thing at a time; for example, when reading an integer from the user, the program doesn’t have to worry about values magically appearing in other variables based on user input - no additional input may occur until the user inputs the current value. In a Windows GUI application, however, the user can actually enter a single digit for one numeric value, switch to a different text entry box and enter several digits for a different number, then switch back to the original input and continue entering data there. Not only does the program need to deal with the simultaneous entry of several different values, but it also has to handle partial inputs in a meaningful way (i.e., if the user ultimately enters the value 1234, the program has to be able to deal with the partial input values 1, 12, 123, as well as the file value, 1234). Since few programmers have had to deal with this type of activity when writing console (non-GUI) applications, this new programming paradigm requires some time before the programmer becomes comfortable with it.

This programming paradigm is known as the *event-oriented programming paradrgm*. It’s called *event-oriented* because the operating system detects events like keypresses, mouse activity, timer timeouts, and other system events, and then passes control to a program that is expecting one or more of these events to occur. Once the application processes the event, the application transfers control back to the operating system which waits for the next event to occur.

The event-oriented programming paradigm presents a perspective that is backwards from the way most programmers first learned to write software. Although this programming scheme takes a little bit of effort to become accustomed to, it’s not really that difficult to master. Although it may be frustrating at first, because it seems like you’re having to learn how to program all over again, fret not, before too long you’ll adjust to the “Windows way of doing things” and it will become second nature to you.

5.2: Message Passing

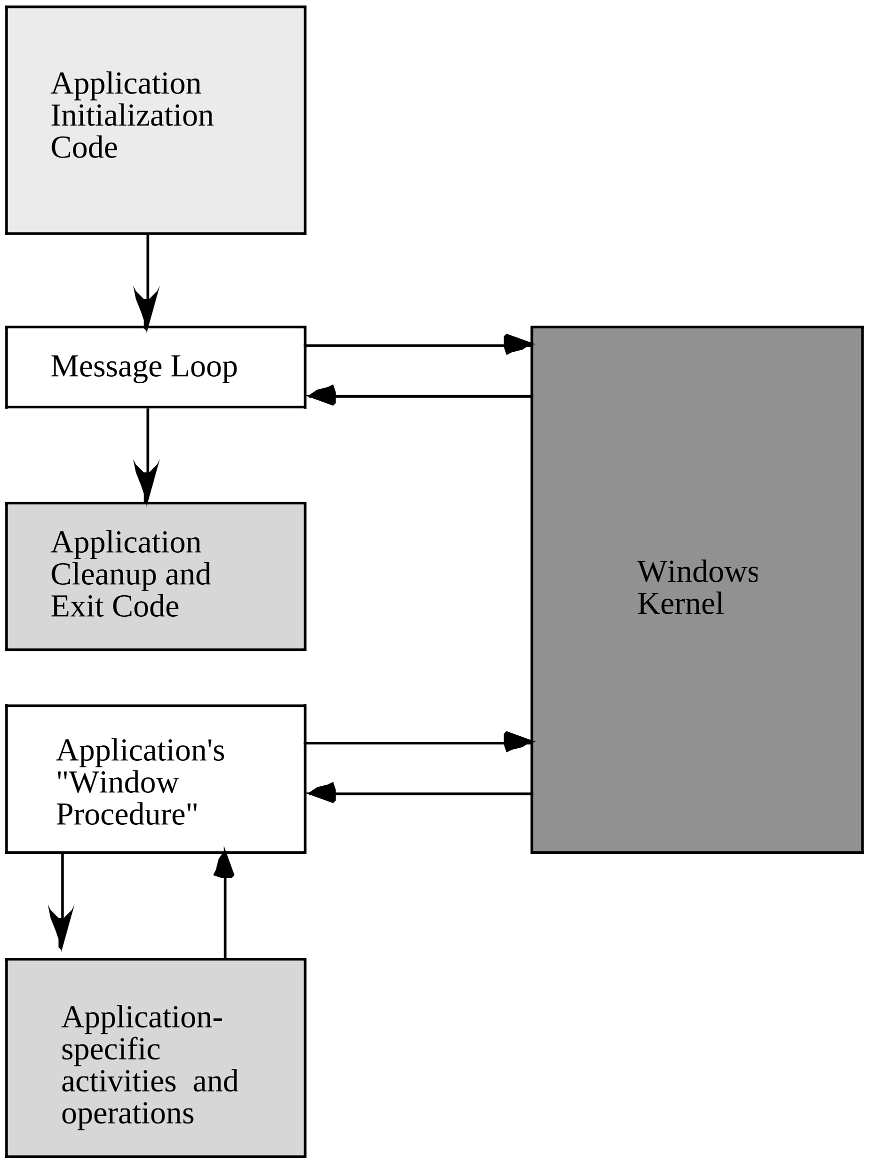
Windows isn’t actually capable of calling an arbitrary function within your application. Although Windows does provide a specialized mechanism for calling certain *callback* functions within your code, most of the time Windows communicates between itself and your application by send your application messages. *Message passing* is just a fancy term for a procedure call. A *message* is really nothing more than a parameter list. The major difference (from your application’s perspective) between a standard procedure call and a message being passed to your application is that the message often contains some value that tells the application what work it expects it to do. That is, rather than having Windows call any of several dozen different subroutines in your application, Windows simply calls a special procedure (known as the window procedure, or *wndproc*) and passes it the message (that is, a parameter list). Part of the message tells the window procedure what event has occurred that the window procedure must handle. The window procedure then transfers control (*dispatches*) to some code that handles that particular event.

> If you’ve ever written a 16-bit DOS application in assembly language, you’ve done some message passing. The INT 21h instruction that “calls” DOS is equivalent to calling DOS’ “window procedure”. The values you pass in the 80x86 registers correspond to the message and, in particular, the value in the AH register selects the particular DOS function you wish to invoke. Although the perspective is different (Windows is calling you instead of you calling Windows), the basic idea behind message passing is exactly the same.

The messages that Windows explicitly sends to your applications are actually quite small: just a 16-byte payload. Four of those bytes contain the message identification (that is, an integer value that tells your window procedure what operation to perform), four bytes form a *window handle* that various functions you call will need, and two double-word data parameters provide message-specific data. Since Windows defines the type of messages it sends to your window procedure, this small message payload (that is, the data) was chosen because it's sufficient for the vast majority of messages and it's efficient to pass between Windows and your application (Windows memory address space is actually in a different address space on the 80x86 from your application and copying large amounts of data between address spaces can be expensive). For those few calls where Windows needs to pass more than eight bytes of data to your application, Windows will allocate a block of memory within your process' address space and pass a pointer to that block of data in one of the two four-byte double words that comprise the message's payload.

Figure 5-1 provides a block diagram of a typical Windows application. This diagram shows how the main program and the window procedure are disconnected. That is, the main program doesn't call the actual application code; instead, Windows handles that activity.

Figure 5-1: General Organization of a Windows Program


Application Initialization Code
Message Loop
Application Cleanup and Exit Code
Application's "Window Procedure"
Application-specific activities and operations
Windows Kernel

For those who are comfortable with client-server architectures, another way to view a Windows application is as a server for Windows' messages. That is, Windows is a client that needs to have certain work done. The application is a server that is capable of performing this work. The server (that is, the application) waits for messages to arrive from Windows telling it what services to perform (e.g., what system events the application should handle). When such a message comes along, the application handles it and then waits for the next message (service request) from the client (Windows). Of course, calling the Window/application relationship a client/server relationship is stretching the point somewhat, because from other perspectives the application is a client that often requests Windows' services. Nonetheless, from the perspective of an application's main program, the client/ server relationship is a useful model.

One question you might have is "how does Windows know the address of your window procedure?" The short answer is "you tell it." The discussion in this chapter has given the impression that Windows applications don't have a main program. Strictly speaking, this is not true. Windows applications do have a main program that the operating system calls when you first run the application. In theory, this main program could execute just like the old-fashioned programs, taking control of the CPU and making (certain) calls to the OS and doing all its processing the old fashioned way. The only catch is that such an application wouldn't behave like a standard Windows GUI application. This is how you write console applications under Windows, but presumably you're reading this book to learn how to write Windows GUI apps rather than Windows console apps. So we'll not consider this possibility any farther.

The real purpose of a Windows main program is to initialize the application, *register the window procedure with the Windows OS*, and then execute an event loop that receives messages from Windows, checks to see if Windows has asked the application to terminate, and then calls a Windows OS function that dispatches the message to whomever it belongs (which is usually your window procedure). Take special note of the phrase "register the window procedure...". This is where you pass Windows the address of your window procedure so it knows how to pass messages to that code. As it turns out, the operation of the main program in a typical Windows application is so standardized that most of the time you will simply "cut and paste" the main program from your previous Windows application into your new application. Rarely will you need to change more than one or two lines in this main program.

When writing a Windows GUI application in HLA, you place the code for the main program of your application between the `begin` and `end` associated with the `program` in HLA (i.e., in the main program section of the HLA program). This may seem completely obvious to an HLA programmer, but to someone who has C/Windows programming experience, this is actually unusual. The main program for a Windows application is usually called *winmain*, at least, if you're writing the application in C/C++. However, the name "winmain" is actually a C/C++ programming convention; the operating system does not require this name at all. To avoid confusion, we'll continue to place our main program where HLA expects the main program when writing GUI applications.

5.3: Handles

Before discussing actual Windows code, the first thing we must discuss is a very important Windows data structure that you'll use everywhere: the *handle*. The Windows operating system uses handles to keep track of objects internal to Windows that are not present in the application's address space. Since there is often need to refer to such internal objects, Windows provides values known as handles to make such reference possible. A handle is simply a small integer value (held in a 32-bit `dword` variable) whose value has meaning only to Windows. Undoubtedly, the handle's value is an index into some internal Windows table that contains the data (or the address of the data) to which the handle actually refers. Windows returns handle values via API function calls, your application must save these values and use them whenever referring to the object that Windows allocated or create via the call.

The Windows C/C++ header files include all kinds of different names for handle object types. This book will simply declare all handles as `dword` variables rather than trying to differentiate them by type. The truth is, you don't do any operations on handles (other than to pass their values to Win32 API functions), so there is little need to go to the extreme of creating dozens (if not hundreds) of different types that are all just isomorphisms of the `dword` type. This book will adopt the Windows/Hungarian notation of prepending an “h” to handle object names (e.g., `hWnd` could be a window handle).

5.4: The Main Program

The main program of a GUI application changes very little from application to application. Indeed, most of the time you’ll simply cut and paste the main program from your previous application and then edit one or two lines when creating a new Windows application. One problem with the main program in a Windows application is that it quickly becomes “out of sight, out of mind” and the knowledge of what is going on inside the main program quickly becomes forgotten. Therefore, it’s worthwhile to spend some time carefully describing the typical Windows main program so you’ll have a good idea of what you can (or should) change with each application that you write.

As noted earlier, one of the most important things the main program of a GUI application does is register a window procedure with the operating system. Actually, registering the window procedure is part of a larger operation: *registering a window class* with Windows. A window class is simply a data structure that maintains important information about a window associated with an application. One of the main tasks of the main program is to initialize this data structure and then call Windows to register the window class.

Although Microsoft uses the term “class” to describe this data structure, don’t let this term confuse you. It really has little to do with C++ or HLA class types and objects. This is really just a fancy name for an instance (that is a variable) of a Windows’ `w.WNDCLASSEX` struct/record. Keep in mind, Windows was originally designed before the days of C++ and before object-oriented programming in C++ became popular. So terms like “class” in Windows don’t necessarily correspond to what we think of as a class today.

5.4.1: Filling in the w.WNDCLASSEX Structure and Registering the Window

Here’s what the definition of `w.WNDCLASSEX` looks like in the HLA *windows.hhf* header file:

```
type
         WNDCLASSEX: record
            cbSize          : dword;
            style           : dword;
            lpfnWndProc     : WNDPROC;
            cbClsExtra      : dword;
            cbWndExtra      : dword;
            hInstance       : dword;
            hIcon           : dword;
            hCursor         : dword;
            hbrBackground   : dword;
            lpszMenuName    : string;
            lpszClassName   : string;
            hIconSm         : dword;
            align(4);
        endrecord;
```

Since the application’s main program must fill in each of these fields, it’s a good idea to take a little space to describe the purpose of each of the fields. The following paragraphs describe these fields.

`cbSize` is the size of the structure. The main program must initialize this with the size of a `w.WNDCLASSEX` structure. Windows uses the value of this field as a “sanity check” on the `w.WNDCLASSEX` structure (i.e., are you really passing a reasonable structure to the function that registers a window class?). Assuming you have a variable `wc` (window class) of type `w.WNDCLASSEX`, you can initialize the `cbSize` field using a single statement like the following:

```
            mov( @size( w.WNDCLASSEX ), wc.cbSize );
```

The `style` field specifies the window's style and how Windows will display the window. This field is a collection of bits specifying several boolean values that control the window's appearance. You may combine these styles using the HLA constant expression bitwise OR operator ("|"). The following paragraphs describe the predefined bit values that are legal for this field:

`w.CS_BYTEALIGNCLIENT`	This style tells Windows to align the window's client area (the part of the screen where the application can draw) on an even byte boundary in order to speed up redraw operations. Note that the use of this option affects where Windows can place the open window on the screen (i.e., dragging the window around may require the window to jump in discrete steps depending on the bit depth of the window). Note that individual pixels on modern video display cards tend to consume multiple bytes, so this option may not affect anything on a modern PC.
`w.CS_BYTEALIGNWINDOW`	This style tells Windows to align the whole window on an even byte boundary in order to speed up redraw operations. Note that the use of this option affects where Windows can place the open window on the screen (i.e., dragging the window around may require the window to jump in discrete steps depending on the bit depth of the window). Note that individual pixels on modern video display cards tend to consume multiple bytes, so this option may not affect anything on a modern PC.
`w.CS_CLASSDC`	Allocates a single device context (which this book will discuss in a later chapter) to be used by all windows in a class. Generally useful in multithreaded applications where multiple threads are writing to the same window.
`w.CS_DBLCLKS`	Tells Windows to send double-click messages to the window procedure when the user double-clicks the mouse within a window belonging to the class. Generally, this option is specified for controls that respond to double-clicks (which are, themselves, windows); you wouldn't normally specify this option for the main window class of an application.
`w.GLOBALCLASS`	This option is mainly for use by DLLs. We won't consider this option here.
`w.CS_HREDRAW`	This class style tells Windows to force a redraw of the window if a movement or size adjustment occurs in the horizontal direction. Most window classes you create for your main window will specify this style option.
`w.CS_NOCLOSE`	This style option disables the close command for this window on the system menu.
`w.CS_OWNDC`	Allocates a unique device context for each window in the class. This option is the converse of `w.CS_CLASSDC` and you wouldn't normally specify both options.
`w.CS_PARENTDC`	Specifies that child windows inherit their parent window's device context. More efficient in certain situations.
`w.CS_SAVEBITS`	Tells Windows to save any portion of a window that is temporarily obscured by another window as a bitmap in Windows' system memory. This can speed up certain redraws, and the window procedure for that window won't have to process as many redraw operations, but it may take longer to display the window in the first place and it does consume extra memory to hold the bitmap. This option is gener-

ally useful for small windows and dialog boxes that don't appear on the screen for long periods of time but may be obscured for brief periods.

`w.CS_VREDRAW`	This option tells Windows to redraw the window if a vertical movement or resize operation occurs. This is another option you'll usually specify for the main application's window that a GUI app creates.

Typically, an application will set the `w.CS_HREDRAW` and `w.CS_VREDRAW` style options. A few applications with special requirements might include one or two of the other styles as well. The following is a typical statement that you'll find in a GUI application that sets these two style options for the application's main window (again, assuming that `wc` is a variable of type `w.WNDCLASSEX`):

```
mov( w.CS_HREDRAW | w.CS_VREDRAW, wc.style );
```

The `lpfnWndProc` field of `w.WNDCLASSEX` holds the address of the window procedure for the application's main window. Initializing this field is how you tell Windows where it can find the window procedure that is going to process all the messages that Windows passes to your application. The window procedure must have the following generic prototype:

```
type
        WNDPROC:
            procedure
            (
                var lpPrevWndFunc   :var;
                    hWnd            :dword;
                    Msg             :dword;
                    _wParam         :dword;
                    _lParam         :dword
            );
                @stdcall;
                @returns( "eax" );
```

If you've got an HLA procedure named `WndProc`, you can initialize the `wc.lpfnWndProc` field using the following code:

```
mov( &WndProc, wc.lpfnWndProc );
```

(note that the type declaration above is in the `w` namespace, so there isn't a name conflict between the `w.WNDPROC` type and the local `WndProc` procedure in your program).

The `cbClsExtra` field specifies the number of bytes of storage to allocate immediately after the window class structure in memory. This provides room for application-specific information associated with the window class. Note that if you have more than one instance of this window class (that is, if you create multiple windows from this same class), they will all share this same storage. Windows will initialize this extra storage with zero bytes. Most applications don't need any extra storage associated with their main window class, so this parameter is usually zero. However, you must still explicitly initialize it with zero if you don't need the extra storage:

```
mov( 0, wc.cbClsExtra );
```

The `cbWndExtra` field specifies the number of bytes of extra storage Windows will allocate for each instance of the window that you create. As you see before too long, it's quite possible to create multiple instances of a single window class; this is unusual for the main window of an application, but it's very common for other "windows" in the system like pushbuttons, text edit boxes, and other controls. This extra storage could hold the data associated with that particular control (e.g., possibly the text associated with a text manipulation control). Windows will allocate this storage in memory immediately following the window instance and initializes the bytes to zeros. Few main application windows need this extra storage, so most Windows' main programs will initialize this field to zero in the window class object for the main window, e.g.,

```
            mov( 0, wc.cbWndExtra );
```

The `hInstance` field is a handle that identifies the window instance for this application. Your program will have to get this value from Windows by making the `w.GetModuleHandle` API call (which returns the `hInstance` handle value in the EAX register). You can initialize the hInstance field using the following code:

```
   w.GetModuleHandle( NULL ); // NULL tells Windows to return this process' handle.
   mov( eax, wc.hInstance );  // Save handle away in wc structure so Windows knows
                              // which process owns this window.
```

The `hIcon` field is a handle to a Windows icon resource. The icon associated with this handle is what Windows will draw whenever you minimize the application on the screen. Windows also uses this code for other purposes throughout the system (e.g., showing a minimized icon on the task bar and in the upper left hand corner of the Window). Later, this book will discuss how to create your own custom icons. For the time being, however, we can simply request that Windows use a "stock icon" as the application's icon by calling the *w.LoadIcon* API function and passing a special value as the icon parameter:

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

The second parameter to `w.LoadIcon` is usually a string containing the name of the icon resource to use. However, Windows also accepts certain small integer values (values that string pointers are never equal to) to specify certain "canned" or "stock" icons. Normally, you cannot pass such a constant where HLA is expecting a reference parameter, however, by prefixing the parameter with the HLA `val` keyword, you can tell HLA to pass the value of the constant as the address for the reference parameter. The value of `w.IDI_APPLICATION` is a Windows predefined constant that tells Microsoft Windows to use the stock application icon for this application. Note that if you pass NULL as the value of the second parameter (e.g., rather than w.IDI_APPLICATION), Windows will tell the application to draw the icon whenever the user minimizes the application. You could use this feature, for example, if you want a dynamic icon that changed according to certain data the application maintains.

The `hCursor` field of the `w.WNDCLASSEX` record holds a handle to a cursor resource that Windows will draw whenever the user moves the cursor over the top of the window. Like the `hIcon` field discussed previously, this handle must be a valid handle that Windows has given you. And just like the initialization of the `hIcon` field, we're going to call a Windows API function to get a stock cursor we can use for our application. Specifically, we're going to ask Windows to give us the handle of an arrow cursor that will draw an arrow cursor whenever the user moves the cursor over our window. Here's the code to do that:

```
   w.LoadCursor( NULL, val w.IDC_ARROW );
   mov( eax, wc.hCursor );
```

The `w.IDC_ARROW` constant is a special Windows-defined value that we supply instead of a pointer to a cursor name to tell Windows to use the standard arrow cursor. Like the `w.LoadIcon` function, if you pass NULL (e.g., rather than w.IDC_ARROW) as the second parameter to `w.LoadCursor`, Windows will expect the application to draw the cursor whenever the mouse moves over the application's window.

The `hbrBackground` field specifies the "brush" that Windows will use to paint the background of a window. A Windows' *brush* is simply a color and pattern to draw. Generally, you'll specify one of the following color constants as this handle value (though you could create a custom brush and use that; this book will discuss the creation of brushes later on):

- w.COLOR_ACTIVEBORDER
- w.COLOR_ACTIVECAPTION
- w.COLOR_APPWORKSPACE
- w.COLOR_BACKGROUND
- w.COLOR_BTNFACE
- w.COLOR_BTNSHADOW
- w.COLOR_BTNTEXT
- w.COLOR_CAPTIONTEXT
- w.COLOR_GRAYTEXT
- w.COLOR_HIGHLIGHT
- w.COLOR_HIGHLIGHTTEXT
- w.COLOR_INACTIVEBORDER
- w.COLOR_INACTIVECAPTION
- w.COLOR_MENU
- w.COLOR_MENUTEXT
- w.COLOR_SCROLLBAR
- w.COLOR_WINDOW
- w.COLOR_WINDOWFRAME
- w.COLOR_WINDOWTEXT

Actually, the value you must supply for the `hbrBackground` value is one of the above constants *plus one*. This is just a Windows idiosyncrasy you'll have to keep in mind. `w.COLOR_WINDOW` (a solid white background) is the typical window color you'll probably use. The following code demonstrates this assignment:

```
            mov( w.COLOR_WINDOW+1, wc.hbrBackground );
```

The `lpszMenuName` field contains the address of a string specifying the *resource name* of the class' main menu, as the name appears in a *resource file*. This book will discuss menus and resource files a little later. In the meantime, if your window class doesn't have a main menu associated with it (or you want to assign the menu later), simply set this field to NULL:

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

The `lpszClassName` field is a string that specifies the class name for this window. This is an important name that you'll use in a couple of other places. Generally, you'll specify the application's name as this string, e.g.,

```
readonly
   myAppClassName :string := "MyAppName";
      .
      .
      .
   mov( myAppClassName, eax );
   mov( eax, wc.lpszClassName );
```

The `hIconSm` is a handle to a small icon associated with the window class. This handle was used in Windows 95, but was ignored by Win NT (and later versions of Windows). The Windows documentation claims that you should initialize this field to NULL in NT and later OSes (and that Windows will set this field to NULL upon return). Most applications, however, seem to initialize this field with the same value they shove into the hIcon field; probably not a bad idea, even if Windows does set this field to NULL later.

Once you fill in all the fields of the w.WNDCLASSEX structure (i.e., wc), you register the window class with Windows by calling the w.RegisterClassEx API function, passing the window class object (wc) as the single parameter, e.g.,

```
            w.RegisterClassEx( wc );
```

The following is all the code appearing throughout this section collected into a contiguous fragment so you can see the complete initialization of the wc variable and the registration of the window class:

```
readonly
   myAppClassName :string := “MyAppName”;
      .
      .
      .
   mov( @size( w.WNDCLASSEX ), wc.cbSize );
   mov( w.CS_HREDRAW | w.CS_VREDRAW, wc.style );
   mov( &WndProc, wc.lpfnWndProc );
   mov( 0, wc.cbClsExtra );
   mov( 0, wc.cbWndExtra );

   w.GetModuleHandle( NULL );
   mov( eax, hInstance );      // Save in a global variable for future use
   mov( eax, wc.hInstance );

   mov( w.COLOR_WINDOW+1, wc.hbrBackground );
   mov( NULL, wc.lpszMenuName );
   mov( myAppClassName, eax );
   mov( eax, wc.lpszClassName );

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

   w.LoadCursor( NULL, w.IDC_ARROW );
   mov( eax, wc.hCursor );

   w.RegisterClass( wc );
```

5.4.2: “What is a ‘Window Class’ Anyway?”

This chapter has made considerable use of the Windows’ term *window class* with only a cryptic discussion of the fact that window classes are not the same thing as C++ or HLA classes. This section will explain the difference between classes in traditional object-oriented programming languages, window classes in Windows, and instances of window classes.

In a language like HLA, a class is a data type. As a general rule, there is no run-time memory associated with a class definition[1]. It’s only when you allocate storage for an *instance* of that class, that is create an *object vari-*

1. One could argue that virtual method table and static class data is associated with the class, not an individual instance of a class, but unless you have at least one instance (object) of a class, there is no need for the static data or virtual method table in memory.

able, that there is storage associated with that class. A class, therefore, is a *layout* of how an object actually uses the memory allocated to it; that is, like a record or structure definition, a class simply defines how the program should treat blocks of memory cells at some offset from the object's base address.

Window classes, on the other hand, do have memory allocated for them. The `wc` variable of the previous section is a good example of a window class that has storage associated with it (indeed, the main purpose of that section was to describe how to initialize the memory storage associated with that window class). So from the very start, we see the major difference between classes in an object-oriented language and windows class: storage. Lest you wonder what Microsoft's engineers were thinking when they created this terminology, just keep in mind that Windows was designed long before object-oriented programming became popular (i.e., before the advent of C++, HLA, and many other popular OOP languages) and terminology like *objects* versus *classes* was not as well-known as it is today.

So, then, exactly what is a window class? Well, a window class is a template[2] that describes a common structure in memory that programs will often duplicate when creating multiple copies of a window. The beautiful thing about a window class is that it lets you initialize the window class record just once and then make multiple copies of that window without having to initialize the data structure associated with each instance of that window class. Now, perhaps, it's a bit difficult to understand why you would want multiple copies of a window or why this is even important based upon the one example we've had in this book to this point. After all, how many times does an application need more than one copy of the application's window (and in the few cases where they do, who really cares about the extra work needed to initialize the window class record, since this is done so infrequently?). Well, if the application's main window were the only window an application would use, there would be little need for window classes. However, a typical Windows GUI application will use dozens, if not hundreds of different windows. This is because Microsoft Windows supports a hierarchical window structure with smaller (*child*) windows appearing within larger (*parent*) windows. Most user interface components (buttons, text edit boxes, lists, etc.) are examples of windows in and of themselves. Each of these windows has its own window class. Although an application may have but a single main window, that application may have many, many, different buttons. Each button appearing on the screen is a window in and of itself, having a window procedure and all the other information associated with a window class. However, all the buttons (at least, of the same type) within a given application share the same windows class. Therefore, to create a new button all you have to do is create a new window based on the button window class. There is no need to initialize a new window class structure for each button if that button shares the attributes common to other buttons the application uses.

Another nice thing about window classes is that Microsoft pre-initializes several common window classes (e.g., the common user interface objects like buttons, text edit boxes, and lists) so you don't even have to initialize the window class for such objects. If you want a new button in your application, you simply create a new window specifying the "button" window class. Since Windows has already registered the button's window class, you don't have to do this. Therein lies the whole purpose of the `w.RegisterWindow` API call: it tells Microsoft Windows about this new window class. Once you register a window class with Microsoft Windows, your application can create instances of that window via the `w.CreateWindowEx` API call (which the next section describes). Although your application will typically create only a single instance of the main application's window, it is quite likely you'll create other window classes that represent *custom controls* that appear within your application. Then your application can create multiple instances of those custom controls by simply calling the `w.CreateWindowEx` API for each instance of the control.

2. The use of the term template, in this context, is generic. This has nothing to do with C++ templates.

5.4.3: Creating and Displaying a Window

Registering a window with the `w.RegisterWindowEx` API call does not actually create a window your application can use, nor does it display the window on your video screen. All this API does is create a template for the window and let Microsoft Windows know about the template so future calls can create instances of that window. The API function that actually creates the window is called (obviously enough) `w.CreateWindowEx`.

The `w.CreateWindowEx` prototype (appearing in the *user32.hhf* header file) is the following:

```
CreateWindowEx: procedure
(
        dwExStyle       :dword;
        lpClassName     :string;
        lpWindowName    :string;
        dwStyle         :dword;
        x               :dword;
        y               :dword;
        nWidth          :dword;
        nHeight         :dword;
        hWndParent      :dword;
        hMenu           :dword;
        hInstance       :dword;
    var lpParam         :var
);
@stdcall;
@returns( "eax" );
@external( "__imp__CreateWindowExA@48" );
```

The `dwExStyle` parameter specifies an extended style value for this window (the extended style parameter is what differentiates the `w.CreateWindowEx` function from the older `w.CreateWindow` API call). This parameter is a bitmap containing up to 32 different style settings that are enabled or disabled by setting the appropriate bit. The *windows.hhf* header file defines a set of constants with names of the form `w.WS_EX_*` that correspond to the possible extended styles. There are a few too many of these, and most of them are a bit too complex, to present at this time. Please see the user32 reference manual (appearing on the CD-ROM accompanying this book) for more details on these extended style values). For the time being, you can initialize this field with zero or, if you prefer you can use the constant `w.WS_EX_APPWINDOW` which tells Windows to put an icon on the taskbar for a top-level instance of this window.

The `lpClassName` field specifies the name of the window class on which you're basing the window you're creating. Generally, this is the string you've supplied as the class name in the call to `w.RegisterWindow`. For certain pre-defined window classes that Windows defines, you can also supply an *atom* value here. An atom is a small 16-bit integer value that uniquely specifies an existing window class (e.g., like the cursor and icon values we saw in the last section). Windows differentiates atoms from strings by looking at the H.O. word of the `lpClassName` parameter value. If this H.O. word contains zero, then Windows assumes that it's an atom value, if the H.O. word is non-zero, then Windows assumes that this parameter contains the address of some string object (note that pointer values in Windows always have a H.O. word that is non-zero).

To pass an atom value rather than a string object as this first parameter, you should use the HLA `val` keyword as a prefix on the atom value, e.g.,

```
   w.CreateWindowEx
   (
            0,
      val   SomeAtomValue,         // Atom values need the “VAL” keyword prefix.
            “WindowName”,
            w.WS_OVERLAPPEDWINDOW, // We’ll explain the following momentarily...
            w.CW_USEDEFAULT,
            w.CW_USEDEFAULT,
            w.CW_USEDEFAULT,
            w.CW_USEDEFAULT,
            NULL,
            NULL,
            hInstance,
            NULL
   );
```

Technically, the `lpClassName` parameter points at a zero-terminated string. However, since HLA string objects are upwards compatible with zero-terminated strings, the `w.CreateWindowEx` prototype specifies an HLA string variable as this parameter. This turns out to be most convenient because most calls to `w.CreateWindowEx` will specify a literal string constant or an HLA string variable here. However, if you’ve got a zero-terminated string that you’d like to use, you don’t need to first convert it to an HLA string, you can use code like the following to directly pass the address of that zero-terminated string to `w.CreateWindowEx`:

```
   lea( eax, SomeZeroTerminatedString );
   w.CreateWindowEx
   (
         0,
         (type string eax),        // Passes pointer to zstring found in EAX.
         “WindowName”,
         w.WS_OVERLAPPEDWINDOW,    // We’ll explain the following momentarily...
         w.CW_USEDEFAULT,
         w.CW_USEDEFAULT,
         w.CW_USEDEFAULT,
         w.CW_USEDEFAULT,
         NULL,
         NULL,
         hInstance,
         NULL
   );
```

Here are some constant values that Windows predefines that you may pass as atom values in place of a string for the `lpClassName` parameter (an in-depth explanation of these class types will appear later in this book):

`w.BUTTON`	This is a small rectangular window that corresponds to a push button the user can click to turn it on or off.
`w.COMBOBOX`	Specifies a control that consists of a list box and a text edit control combined into a single control. This control allows the user to select some text from a list or type the text from the keyboard.
`w.EDIT`	This specifies an edit box which is a rectangular window into which the user may type some text.
`w.LISTBOX`	This atom specifies a list of character strings. The user may select one of these strings by clicking on it.
`w.MDICLIENT`	Designates an MDI (multiple document interface) client window. This tells Windows to send MDI messages to the window procedure associated with this window.

`w.RichEdit`	Specifies a Rich Edit 1.0 control. This provides a rectangular window that supports text entry and formatting and may include embedded COM objects.
`w.RICHEDIT_CLASS`	Specifies a Rich Edit 2.0 control
`w.SCROLL_BAR`	Specifies a rectangular window used to hold a scroll bar control with direction arrows at both ends of the scroll bar.
`w.STATIC`	Specifies a text field, box, or rectangle used to label, box, or separate other controls.

For the main application's window, you would not normally specify one of these atoms as a window class. Instead, you'd supply a string specifying a name for the application's window class. We'll return to the discussion of controls in a later chapter in this book.

The third `w.CreateWindowEx` parameter, `lpWindowName`, is a string that holds the window's name. This is caption that is associated with the window's title bar. Some applications will also identify an instance of a window on the screen by using this string. Typically, if you have multiple instances of a window class appearing on the screen at the same time, you will give each instance a unique window name so you can easily differentiate them. Generally, the class name and the window name are similar, but not exactly the same. A class name typically looks like a program identifier (i.e., no embedded spaces and the characters in the name would be those that are legal in a program source file). The window name, on the other hand, is usually formatted for human consumption.

The fourth parameter, `dwStyle`, specifies a set of window styles for the window. Like the `dwExStyle` parameter, this object is a bitmap containing a set of boolean values that specify the presence or absence of some window attribute. The following is a partial list of values you may logically OR together for form the `dwStyle` value. We'll explain the terminology and specifics later in this book. These are thrown out here just for completeness. We'll actually only use a single window style for our application's main window.

`w.WS_BORDER`	Specifies a thin border around the window.
`w.WS_CAPTION`	Creates a window that has a title bar (also sets the `w.WS_BORDER` attribute).
`w.WS_CHILD`	Creates a child window. Mutually exclusive to the `w.WS_POPUP` attribute.
`w.WS_CHILDWINDOW`	Same as `w.WS_CHILD` attribute.
`w.WS_CLIPCHILDREN`	Excludes the area occupied by child windows when drawing occurs within the parent window. Use this style when creating a parent window.
`w.WS_CLIPSIBLINGS`	Clips child windows relative to one another. Specify this when creating a child window when you have several child windows that could overlap one another.
`w.WS_DISABLED`	Creates a window that is initially disabled.
`w.WS_DLGFRAME`	Creates a window with a border designed for a dialog box.
`w.WS_GROUP`	Specifies the first control of a group of controls (remember, controls are windows). The next control that has the w.WS_GROUP style ends the current group and begins the next group.
`w.WS_HSCROLL`	Creates a window with a horizontal scroll bar.
`w.WS_ICONIC`	Creates a window that is initially minimized.
`w.WS_MAXIMIZE`	Creates a window that is initially maximized.
`w.WS_MAXIMIZEBOX`	Creates a window that has a maximize button.
`w.WS_MINIMIZE`	Creates a window that is initially minimized (same as w.WS_ICONIC).
`w.WS_MINIMIZEBOX`	Creates a window that has a minimize button.

`w.WS_OVERLAPPED`	Creates an overlapped window.
`w.WS_OVERLAPPEDWINDOW`	This is a combination of several styles include `w.WS_OVERLAPPED`, `w.WS_CAPTION`, `w.WS_SYSMENU`, `w.WS_SIZEBOX`, `w.WS_MINIMIZEBOX`, and `w.WS_MAXIMIZEBOX`. This is the typical style an application's window will use.
`w.WS_POPUP`	Creates a popup window. Mutually exclusive to the `w.WS_CHILD` window style.
`w.WS_POPUPWINDOW`	Creates a pop-up window with the following styles: `w.WS_BORDER`, `w.WS_POPUP`, `w.WS_SYSMENU`. The `w.WS_POPUPWINDOW` and `w.WS_CAPTION` styles must both be active to make the system menu visible.
`w.WS_SIZEBOX`	Creates a window that has a sizing border. This style is the same as the `w.WS_THICKFRAME` style.
`w.WS_SYSMENU`	Creates a window that has a system menu box in its title bar. You must also specify the `w.WS_CAPTION` style when specifying this attribute.
`w.WS_THICKFRAME`	Same as `w.WS_SIZEBOX` style.
`w.WS_TILED`	Save as the `w.WS_OVERLAPPED` style.
`w.WS_TILEDWINDOW`	Same as the `w.WS_OVERLAPPEDWINDOW` style.
`w.WS_VISIBLE`	Creates a window that is initially visible.
`w.WS_VSCROLL`	Creates a window that has a vertical scroll bar.

These styles are appropriate for generic windows. Certain window classes have their own specific set of window styles. In particular, the button window class, the combobox window class, the text edit window class, the list box window class, the scroll bar window class, the static window class, and the dialog window class have their own set of window style values you can supply for this parameter. We'll cover this specific window styles when we discuss those controls later in this book.

For generic windows, the `w.WS_OVERLAPPEDWINDOW` style is a good style to use. Depending on your needs, you may want to merge in the `w.WS_HSCROLL` and `w.WS_VSCROLL` styles as well. You can also specify the `w.WS_VISIBLE` style if you like, but we'll be making a call to make the window visible soon after calling `w.CreateWindowEx`, so merging in this style isn't necessary.

The next four parameters, `x, y, nWidth` and `nHeight` specify the position and size of the window on the display. If your window must be a certain size and it must appear at a certain location on the screen, then you may fill in this parameter with appropriate screen coordinate values. Another good use of these parameters is to automatically restore the application window's position and size from their values the last time the user ran the application (presumably, you've saved the values in a file or in the system registry before quitting if your application is going to do this). Most applications (particularly, those that allow the user to resize the window) don't really care about the initial size and position of the main application window. After all, if the user doesn't like what comes up, the user can move or resize the window to their liking. In such situations, a user can supply the generic constant `w.CW_USEDEFAULT` that tells Windows to place the window at an appropriate point on the screen. Windows will typically center such windows and have them consume approximate half the screen's size.

If you decide to supply explicit coordinates and dimensions for the application's window, be cognizant of the fact that Windows runs on a wide variety of machines with window sizes ranging from 640x480 (and, technically even smaller) to very large. When choosing a screen position and size for your window, be sure to consider the fact that someone may be running your application on a machine with a smaller screen than the one on your machine. This is why using `w.CW_USEDEFAULT`, if possible, is a good idea. Windows can automatically adjust the window dimensions as appropriate for the machine on which the application is running.

The `hWndParent` parameter supplies the handle of a parent window whenever you're creating a child window. Buttons, text edit boxes, and other controls are good examples of child windows. An application's main window, however, isn't a child window. So you'll normally supply NULL for this parameter when creating the main window for an application.

The `hMenu` parameter provides the handle for a menu to be used with a window or a child window identifier for the child window style. We'll come back to the discussion of menus in a later chapter. For now, you can place a NULL in this field to tell windows that your application's window doesn't have a menu.

The `hInstance` parameter is where you pass the module (application) handle. You obtain this value via the `w.GetModuleHandle` API call. Note that the window class variable (`wc` in the previous section) also requires this handle, when the application's main program initialized the class variable it also saved the application's handle into a global variable hInstance for use by `w.CreateWindowEx` API calls. Because future calls will need this value as well, having it available in a global variable is a good idea (of course, it's also present in the `wc.hInstance` field, but it's still convenient to keep it in a global variable).

The last `w.CreateWindowEx` parameter is used to specify the address of a `w.CREATESTRUCT` object for MDI windows. If you're not creating an MDI window (and most applications don't), you can specify NULL for this field.

The `w.CreateWindowEx` API function returns a handle to the window it creates in the EAX register. You will use this handle whenever referencing the window. Therefore, it's a good idea to save away this variable into a global variable immediately upon return from `w.CreateWindowEx` (you'll want to use a global variable because lots of different procedures and functions through out the application will need to reference this variable's value).

The w.CreateWindowEx API function creates an actual instance of some window class and initializes it appropriately. It does not, however, actually put the window on the screen. That takes another couple of calls and some extra work. To tell windows to show your window (i.e., make it visible), you use the `w.ShowWindow` API call thusly[3]:

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

The first parameter to this function is the window handle that `w.CreatWindowEx` returns. The second parameter specifies how Windows should display the window, the `w.SW_SHOWNORMAL` is the appropriate value to use when displaying the window for the first time.

Despite its name, `w.ShowWindow` doesn't actually make your window visible on the display. It simply sets the "show state" for this particular window. Although Windows will draw the frame of your window for you, it is your responsibility to actually fill in the "client area" of the window. That is done by having Windows send your application a message telling it to paint itself. Although you currently have control of the CPU, one thing you cannot arbitrarily do is draw to the screen without Windows telling you to do so (this is especially important because your window isn't even on the screen at this point). In order to draw your window, you've got to tell Windows to send your window procedure a message and then your window procedure can do the job of actually filling in the screen information. You can do this with a `w.UpdateWindow` call as follows:

```
w.UpdateWindow( hwnd );
```

Again, remember, `w.UpdateWindow` does not actually draw the window. It simply tells Windows to send your application a message that will cause it to draw the window (inside the window procedure). The actual drawing does not take place in your application's main program.

Once you've told Windows to update your window so it can be drawn for the first time, all that's left for your main program to do is to process Windows' messages. The next section describes that activity. At this point, you've created your window and told Windows to display it. Once you begin processing Windows' messages,

3. Despite its name, you actually use the `w.ShowWindow` API function to show or hide a window. See the API documentation for more details.

you'll actually display the window (since one of the first messages that will come along is the message telling your application to draw its window).

5.4.4: The Message Processing Loop

After you initialize, register, and create your application's main window and tell Windows to display the window, the last major piece of work your application's main program must do is begin processing Windows messages. The message processing loop is actually a small piece of code, so short that we'll just reproduce the whole thing in one chunk:

```
forever

   w.GetMessage( msg, NULL, 0, 0 );  // Get a message from Windows
   breakif( EAX = 0 );               // When GetMessage returns zero, time to quit
   w.TranslateMessage( msg );        // Converts keyboard codes to ASCII
   w.DispatchMessage( msg );         // Calls the appropriate window procedure

endfor;

mov( msg.wParam, eax );              // Get this program's exit code
w.ExitProcess( eax );                // Quit the application
```

This code repeatedly calls `w.GetMessage` until `w.GetMessage` returns false (zero) in the EAX register. This is a signal from Windows that the user has decided to terminate our amazing program. If `w.GetMessage` returns true, then the message loop calls `w.TranslateMessage` (which mainly processes keystrokes) and then it calls `w.DispatchMessage` (which passes the messages on to the window procedure, if appropriate).

The `w.GetMessage` function transfers control from your program to Windows so Windows can process keystrokes, mouse movements, and other events. When such an event occurs (and is directed at your program), Windows returns from `w.GetMessage` after having filled in the `msg` variable with the appropriate message information. The filter parameters should contain zero (so `w.GetMessage` will return all messages from the queue). The second parameter normally contains NULL which means that the program will process all messages sent to any window in the program. If you put a window handle here, then `w.GetMessage` will only return those messages directed at the specified window.

On return, the `msg` parameter contains the message information returned by Windows. Normally, you can ignore the contents of this message variable, all you really need to do is pass the message on to the `w.TranslateMessage` and `w.DispatchMessage` functions. However, just in case you're interested, here's the definition of the `w.MSG` type in the *windows.hhf* header file:

```
type
    MSG: record
        hwnd    : dword;
        message : dword;
        wParam  : dword;
        lParam  : dword;
        time    : dword;
        pt      : POINT;
    endrecord;
```

The `w.TranslateMessage` API function takes messages containing keyboard virtual scan codes and computes the ASCII/ANSI code associated with that keystroke. By placing this function call in the main message passing loop, Microsoft effectively provides a “hook” allowing you to replace this translation operation with a function of your own choosing. The `w.TranslateMessage` takes scan codes of the form *shift down*, *shift up*, *‘A’ key down*, *‘A’ key up*, *control key down*, and *control key up* and decides whether a virtual key code like the code for the ‘A’ key should be converted to the character ‘a’, ‘A’, control-A, Alt-A, etc. Normally, you’ll want this default translation to take place, so you’ll leave in the call to `w.TranslateMessage`. However, by breaking out the call in this fashion, Windows allows you to replace `w.TranslateMessage` entirely, or inject some code to handle a specific keystroke sequence that you want to handle specially within your application.

The `w.DispatchMessage` API function takes the translated message and calls the appropriate window procedures, passing along the (translated) message. Upon return from `w.DispatchMessage`, every application window that has reason to deal with that message will have done so.

At first blush, it might seem weird that Microsoft would even make you write the message processing loop as part of your main program. After all, the loop simply makes three calls to Win32 API functions; surely the OS could bury this code inside the operating system and spare the application’s main program the (admittedly small) expense of dealing with this operation. However, the main reason for requiring this code in the application program is explicitly to provide the application with the ability to hook into the message processing loop both before and after the call to `w.DispatchMessage`.

5.4.5: The Complete Main Program

Here’s the source code for a complete Windows’ main program, collected into one spot:

```
program main;
#include( “wpa.hhf” )        // Abridged version of windows.hhf/w.hhf

storage
   hInstance  :dword;        // Application’s module handle
   hwnd       :dword;        // Main application window handle
   msg        :w.MSG;        // Message data passed in from Windows
   wc         :w.WNDCLASSEX; // Windows class for main app window

readonly
   myAppClassName :string := “MyAppName”;

   << Other declarations and procedures would go here... >>

begin main;

   mov( @size( w.WNDCLASSEX ), wc.cbSize );
   mov( w.CS_HREDRAW | w.CS_VREDRAW, wc.style );
   mov( &WndProc, wc.lpfnWndProc );
   mov( 0, wc.cbClsExtra );
   mov( 0, wc.cbWndExtra );

   w.GetModuleHandle( NULL );
   mov( eax, hInstance );      // Save in a global variable for future use
   mov( eax, wc.hInstance );

   mov( w.COLOR_WINDOW+1, wc.hbrBackground );
   mov( NULL, wc.lpszMenuName );
   mov( myAppClassName, eax );
   mov( eax, wc.lpszClassName );
```

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

    w.LoadCursor( NULL, w.IDC_ARROW );
    mov( eax, wc.hCursor );

    w.RegisterClass( wc );

    w.CreateWindowEx
    (
        0,                      // No specific extended styles
        myAppClassName,         // This application’s class name.
        “My First App”,         // Window caption
        w.WS_OVERLAPPEDWINDOW, // Draw a normal app window.
        w.CW_USEDEFAULT,        // Let Windows choose the initial
        w.CW_USEDEFAULT,        //  size and position for this window.
        w.CW_USEDEFAULT,
        w.CW_USEDEFAULT,
        NULL,                   // This is the parent window.
        NULL,                   // This window has no default menu.
        hInstance,              // Application’s handle.
        NULL                    // We’re not a child window.
    );

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

    forever

       w.GetMessage( msg, NULL, 0, 0 );  // Get a message from Windows
       breakif( EAX = 0 );               // When GetMessage returns zero, time to quit
       w.TranslateMessage( msg );        // Converts keyboard codes to ASCII
       w.DispatchMessage( msg );         // Calls the appropriate window procedure

    endfor;

    mov( msg.wParam, eax );              // Get this program’s exit code
    w.ExitProcess( eax );                // Quit the application

end main;
```

5.5: The Window Procedure

Since the application’s main program doesn’t call any other functions within the application, someone reading the source code to a Windows application for the first time may very well wonder how the rest of the code in the application executes. As this chapter notes in several places, Windows automatically calls the window procedure whose address appears in the `lpfnWndProc` field of the window class variable when it needs to send the application a message. Part of the message package that Windows passes to the window procedure is a value that specifies the message type. The window procedure interprets this value to determine what activity to perform in response to the message. The window procedure (or subroutines called by the window procedure) is where all the activity takes place in a typical windows application.

The prototype for a window procedure takes the following form:

```
procedure WndProc( hwnd: dword; uMsg:dword; wParam:dword; lParam:dword );
   @stdcall;
   @nodisplay;
   @nostackalign;
```

The traditional name for this procedure is `WndProc` and that's the name you'll see most programs use. However, you may use any name you like here. All that Windows cares about is that you initialize the `lpfnWndProc` field of the window class variable with the address of this procedure prior to registering the window. So if you named this procedure `MyWindowProcedure` it would work fine as long as you initialized the window class variable (say, `wc`) with its address as follows:

```
         mov( &MyWindowProcedure, wc.lpfnWndProc );
```

The `hwnd` parameter is a handle to the window at which this message is explicitly directed. All of the windows instantiated from the same window class share the same window procedure. This allows a single window procedure to process messages for several different windows. Of course, typically there is only a single instance of the main application's window class (that is, the main application's window) so your main window procedure typically handles messages for only one window. However, if you create multiple instances of some window class (e.g., you're creating a component like a button), you can explicitly test to see if the message is directed at a specific instance of that window class by comparing the `hwnd` parameter against the handle value that `w.CreateWindowEx` returns. In this chapter, we'll assume that there is only one instance of the main application's window, so we'll just ignore the `hwnd` parameter.

The `uMsg` parameter is an unsigned integer value that specifies the type of the message Windows is sending the window procedure. There are, literally, hundreds of different messages that Windows can send an application. You can find their values in the windows header files by searching for the constant definitions that begin with "WM_" (the WM, obviously, stands for "Windows Message"). There are far too many to present the entire list here, but the following constant declarations provide examples of some common Windows messages that could be sent to your application's window procedure:

```
const
    WM_CREATE := $1;
    WM_DESTROY := $2;
    WM_MOVE := $3;
    WM_SIZE := $5;
    WM_ACTIVATE := $6;

    WM_PAINT := $0F;
    WM_CLOSE := $10;

    WM_CUT := $300;
    WM_COPY := $301;
    WM_PASTE := $302;
    WM_CLEAR := $303;
    WM_UNDO := $304;
```

The important thing to notice is that commonly used message values aren't necessarily contiguous (indeed, they can be widely spaced apart) and there are a lot of them. This pretty much precludes using a `switch/case` statement (or an assembly equivalent - a jump table) because the corresponding jump table would be huge. Since few window procedures process more than a few dozen messages, many application's window procedures just use a `if..else if` chain to compare `uMsg` against the set of messages the window procedure handles; therefore, a typical window procedure often looks somewhat like the following:

```
   procedure WndProc( hwnd: dword; uMsg:dword; wParam:dword; lParam:dword );
      @stdcall;
      @nodisplay;
      @nostackalign;

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 );
    if( eax = w.WM_DESTROY ) then

        w.PostQuitMessage( 0 );   // Do this to quit the application

    elseif( eax = w.WM_PAINT ) then

        << At this point, do whatever needs to be done to draw the window >>

    else

        // 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 );

    endif;

end WndProc;
```

There are two problems with this approach. The major problem with this approach is that you wind up processing all your application’s messages in a single procedure. Although the body of each `if` statement could, in theory, call a separate function to handle that specific message, in practice what really happens is the program winds up putting the code for a lot of the messages directly into the window procedure. This makes the window procedure really long and more difficult to read and maintain. A better solution would be to call a separate procedure for each message type.

The second problem with this organization for the window procedure is that it is effectively doing a linear search using the `uMsg` value as the search key. If the window procedure processes a lot of messages, this linear search can have a small impact on the performance of the application. However, since most window procedures don’t process more than a couple dozen messages and the code to handle each of these messages is usually complex (often involving several Win32 API calls, which are slow), the concern about using a linear search is not too great. However, if you are processing many, many, different types of messages, you may want to consider using a binary search or hash table search to speed things up a bit. We’ll not worry about the problem of using a linear search in this book; however, the cost of getting to the window procedure and the cost associated with processing the message is usually so great that it swamps any savings you obtain by using a better search algorithm. However, those looking to speed up their applications in certain circumstances may want to consider a better search algorithm and see if it produces better results. Of course, another alternative is to go ahead and use a jump table (large though it might be) which can transfer control to an appropriate handler in a fixed amount of time.

There are a couple of solutions to the first problem (organizing the code so that it is easier to read and maintain). The most obvious solution, as noted earlier, is to call a procedure within each `if..then` body. A possibly better solution, however, is to use a table of message values and procedure addresses and search through the table until the code matches a message value; then the window procedure can call the corresponding procedure for that message. This scheme has a couple of big advantages over the `if..then..elseif` chain. First of all, it allows you to write a generic window procedure that doesn't change as you change the set of messages it has to handle. Second, adding new messages to the system is very easy. Here's the data structures we'll use to implement this:

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

    MsgProcPtr_t:
        record

            MessageValue:   dword;
            MessageHndlr:   MsgProc_t;

        endrecord;
```

`MsgProc_t` is the generic prototype for the message handler procedures we're going to write. The parameters to this function almost mirror the parameters Windows passes to the window procedure; the `uMsg` parameter is missing because, presumably, each different message value invokes a different procedure so the procedure should trivially know the message value. `MsgProcPtr_t` is a record containing two entries: a message number (`MessageValue`) and a pointer to the message handler procedure (`MessageHndlr`) to call if the current message number matches the first field of this record. The window procedure will loop through an array of these records comparing the message number passed in by Windows (in `uMsg`) against the `MessageValue` field. If a match is made, then the window procedure calls the function specified by the `MessageHndlr` field. Here's what a typical table (named `Dispatch`) of these values looks like in HLA:

```
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.
```

Each entry in the table consists of a record constant (e.g., `MsgProcPtr_t:[w.WM_DESTROY,&QuitApplication])` containing a message number constant and the address of the procedure to call when the current message number matches that constant. The end of the list contains zeros (NULL) in both entries (e.g., `MsgProcPtr_t:[0,NULL]`).

To handle a new message in this system, all you have to do is write the message handling procedure and stick a new entry into the table. No changes are necessary in the window procedure. This makes maintenance of the window procedure very easy. The window procedure itself is fairly straight-forward, here's an example of a window procedure that processes the entries in the `Dispatch` table:

```
// 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:dword; wParam:dword; lParam:dword );
      @stdcall;
      @nodisplay;
      @nostackalign;

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 = NULL ) 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 );
            break;


        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. This code manually
            // pushes the parameters and calls the handler procedure (note
            // that the message handler procedure uses the HLA/Pascal calling
            // sequence, so we must push the actual parameters in the same
            // order as the formal parameters were declared).

            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 );  // Move on to next table entry.

    endfor;

end WndProc;
```

This code uses EDX to step through the table of `MsgProcPtr_t` records. This procedure begins by initializing EDX to point at the first element of the `Dispatch` array. This code also loads the `uMsg` parameter into EAX where the procedure can easily compare it against the `MessageValue` field pointed at by EDX. A zero routine address marks the end of the `Dispatch` list, so this code first moves the value of that field into ECX and checks for zero. When the code reaches the end of the `Dispatch` list without finding a matching message number, it calls the Windows API `w.DefWindowProc` function that handles default message handling (that is, it handles any messages that the window procedure doesn't explicitly handle).

If the window procedure dispatch loop matches the value in EAX with one of the `Dispatch` table's message values, then this code calls the associated procedure. Since the address is already in ECX (from the comparison against NULL for the end of the list), this code manually pushes the parameters for the message handling procedure onto the stack (in the order of their declaration, since the message handling functions using the HLA/Pascal calling convention) and then calls the handler procedure via the address in ECX.

This routine chose EAX, ECX, and EDX because the Intel ABI (and Windows) allows you to trash these registers within a procedure call. The Intel ABI also specifies that functions should return 32-bit results in the EAX register, which is another reason for using EAX - it's going to get trashed by the return result anyway. Note that the message handler procedures must also follow these rules. That is, they are free to disturb the values in EAX, ECX, and EDX, but they must preserve any other registers that they modify. Also note that upon entry into the message handling procedures, EAX contains the message number. So if having this value is important to you (for example, if you use the same message handler procedure for two separate messages), then just reference the value in EAX.

Once we have the `Dispatch` table and the `WndProc` procedure, all that's left to do is write the individual message handling procedures and we'll have a complete Windows application. The question that remains is: "What applications shall we write?" Well, historically, most programming books (including almost every Windows programming book) has started off with the venerable "Hello World" program. So it makes perfect sense to continue that fine tradition here.

5.6: Hello World

To create a complete Windows GUI application based on the code we've written thus far, we've only got to add two procedures: `QuitApplication` and `Paint`. A minimal Windows GUI application (like *HelloWorld*) will have to handle at least two messages: `w.WM_DESTROY` (which tells the application to destroy the window created by the main program and terminate execution) and `w.WM_PAINT` (which tells the application to draw its main window).

The `QuitApplication` is a fairly standard procedure; almost every Windows GUI app you write with HLA will use the same code. Here is a sample implementation:

```
// 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;
```

The `w.PostQuitMessage` API function does just what its name implies - it sends ("posts") a message to the main message loop that tells the message loop to terminate the program. On the next iteration of the message loop in the main program, the `w.GetMessage` function will return zero in EAX which tells the application to terminate (look back at the main program example for details). The parameter you pass to `w.PostQuitMessage` winds up in the `msg.wParam` object in the main program, this is the program's return code. By convention, main programs return a zero when they successfully terminate. If you wanted to return an error code, you'd pass that error code as the parameter to `w.PostQuitMessage`.

One embellishment you could make to the `QuitApplication` procedure is to add any application-specific code needed to clean up the execution state before the program terminates. This could include flushing and closing files, releasing system resources, freeing memory, etc. Another possibility is that you could open up a dialog box and ask the user if they really want to quit the program.

The other procedure you'll need to supply to have a complete, functional, *HelloWorld* program is the `Paint` procedure. The `Paint` procedure in our Win32 application is responsible for drawing window data on the screen. Explaining exactly what goes into the `Paint` procedure is actually the subject of much of the rest of this book and it would be foolish to try and explain everything that `Paint` must do in the few words available in this section. So rather than try and anticipate questions with a lot of premature explanation, here's the `Paint` procedure without too much ado:

```
// 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.

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.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;
```

The `w.BeginPaint` and `w.EndPaint` procedure calls must bracket all the drawing that takes place in the `Paint` procedure. These procedures set up a device context (`hdc`) that Windows uses to determine where the output should wind up (typically, the video display, but it could wind up somewhere else like on a printer). We'll have a lot more to say about these functions in the very next chapter, for now just realize that they're a requirement in order to draw on the window.

The `w.GetClientRect` API function simply returns the x- and y-coordinates of the outline of the client area of the window. The client area of a window is that portion of the window where the application can draw (the client area, for example, does not include the scroll bars, title bar, and border). This function returns the outline of the client area in a `w.RECT` object (the `rect` parameter, in this case). The `Paint` function retrieves this information so it can print a string centered within the client area.

The `w.DrawText` function is what does the real work as far as the nature of this program is concerned: this is the call that actually displays "Hello World!" within the window. The `w.DrawText` function uses the following prototype:

```
    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 handle to the device context where `w.DrawText` is to put the text. In the call to `w.DrawText` appearing earlier, the `Paint` procedure passes in the `hdc` value returned by `w.BeginPaint`. The `lpString` parameter is a pointer to a zero-terminated sequence of ASCII characters (e.g., an HLA string object). The `nCount` parameter specifies the number of characters to print from the string; if you pass -1 (as this call does) then `w.DrawText` will display all the characters up to the zero-terminating byte. The `lpRect` parameter specifies a pair of (X,Y) coordinates that form a rectangle in the client area; `w.DrawText` will draw the text within this rectangular area based on the value of the `uFormat` parameter. The `w.DT_SINGLELINE, w.DT_CENTER`, and `w.VCENTER` parameters tell `w.DrawText` to place a single line of text in the window, centered vertically and horizontally within the rectangle supplied as the `lpRect` parameter.

After the call to `w.DrawText`, the `Paint` procedure calls the `w.EndPaint` API function. This completes the drawing sequence and it is at this point that Windows actually renders the text on the display device. Note that all drawing must take place between the `w.BeginPaint` and `w.EndPaint` calls. Additional calls to functions like `w.DrawText` are not legal once you call `w.EndPaint`. There are many additional functions you can use to draw information in the client area of the window; we'll start taking a look at some of these functions in the next chapter.

Here's the complete *HelloWorld* application:

```
// HelloWorld.hla:
//
// The Windows "Hello World" Program.

program HelloWorld;
#include( "wpa.hhf" )       // Standard windows stuff.


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 := "HWWinClass";             // Window Class Name
    AppCaption: string := "Hello World Program";    // 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.
//  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.

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.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 HelloWorld;


    // 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 HelloWorld;
```

5.7: Compiling and Running *HelloWorld* From the Command Line

The *hla.exe* command-line program automatically runs several different programs during the compilation of an HLA source file. It runs the HLA compiler, proper (*hlaparse.exe*), it runs the *ml.exe* (MASM, the Microsoft

Macro Assembler) program to assemble the *.asm* file that HLA produces[4], it optionally runs the *rc.exe* (resource compiler) program if you specify any *.rc* files on the HLA command line, and it runs the *link.exe* program to link all the object files together to produce an executable. The *hla.exe* program is so flexible, it is all you will need to use for small projects[5]. However, there is one issue that you must consider when compiling GUI Windows applications with HLA: by default, HLA generates console applications, not Windows applications. Since we're compiling actual Windows applications, we need to tell HLA about this.

Telling HLA to compile Windows applications rather than console applications is very easy. All you've got to do is include the "-w" command line option as follows[6]:

```
            hla -w helloWorld.hla
```

This command line option passes some information to the *link.exe* program so that it generates appropriate object code for a Windows app versus a console app. That's all there is to it! However, don't forget to include this option or your application may misbehave.

To run the *helloWorld.exe* application, you can either type "helloWorld" at the command line prompt or you can double-click on the *helloWorld.exe* application's icon. This should bring up a window in the middle of your display screen heralding the phrase "Hello World!" You can quit the program by clicking on the application's close box in the upper right hand corner of the window.

Although it is a relatively trivial matter to compile the "Hello World" program directly from the command line, this book will always provide a makefile that you can use to completely compile any full program example. That way, you can always use the same command to compile trivial as well as complex Windows applications. The accompanying CD-ROM contains all the source code for each major project appearing in this book; with each project appearing in its own subdirectory and each subdirectory containing a makefile that will build the executable for that project. Following the plan from Chapters one and three, the makefile for the "Hello World" application provide several options that interface with RadASM (see the next section) and provide the ability to do several different types of compiles from the command line. Here's a makefile for the "Hello World" application:

```
build: HelloWorld.exe

buildall: clean HelloWorld.exe

compilerc:
        echo No Resource Files to Process!

syntax:
        hla -s HelloWorld.hla

run: HelloWorld.exe
        HelloWorld

clean:
        delete /F /Q tmp
```

4. HLA can produce code for other assemblers like TASM, FASM, and Gas. In this book, however, we'll assume the use of MASM.
5. For larger projects, you will probably want to consider using a "make" program like Microsoft's NMAKE.EXE in order to speed up the development process and ease maintenance of your code. This text will generally avoid the use of makefiles so that there is one less thing you have to be concerned about.
6. This book assumes that you've properly installed HLA and you've been able to compile small console-mode applications like a text-based "Hello World" program. See the HLA documentation for more details on setting up HLA if you haven't done this already.

```
        delete *.exe
        delete *.obj
        delete *.link
        delete *.inc
        delete *.asm
        delete *.map


HelloWorld.obj: HelloWorld.hla wpa.hhf
        hla -p:tmp -w -c HelloWorld

HelloWorld.exe: HelloWorld.hla wpa.hhf
        hla -p:tmp -w HelloWorld
```

By default (that is, if you just type “make” at the command line) this *makefile* will build the executable for the *HelloWorld.exe* program, if it is currently out of date. You may also specify command line options like “buildall” or “clean” to do other operations. See Chapters one and three for more details on these options.

Whenever you consider the text-based version of the HLA “Hello World” program, this GUI version seems somewhat ridiculous. After all, the text-based version only requires the following HLA code:

```
program helloWorldText;
#include( “stdlib.hhf” )
begin helloWorldText;

   stdout.put( “Hello World!” nl );

end helloWorldText;
```

So why must the GUI version be so much larger? Well, for starters, the GUI version does a whole lot more than the text version. The text version prints “Hello World!” and that’s about it. The GUI version, on the other hand, opens up a window that you can move around on the screen, resize, open up a system menu, minimize, maximize, and close. Today, people have been using Windows and Macintosh applications for so long that they take the effort needed to write such “trivial” code for granted. Rest assured, doing what this simple GUI “Hello World” application does would be a tremendous amount of work when running under an operating system like Microsoft’s old DOS system where all the graphics manipulation was totally up to the application programmer. What the GUI “Hello World” application accomplishes in fewer than 300 lines of code would take thousands of lines of code under an OS like DOS.

5.8: Compiling and Running *HelloWorld* from RadASM

The HelloWorld directory on the accompanying CD-ROM contains the RadASM “.rap” (RadAsm Project) file and the makefile that RadASM can use to build this file. Just load *HelloWorld.rap* into RadASM and select “Build” or “Run” from the “Make” menu.

5.9: Goodbye World!

Well, we’ve just about beat the *HelloWorld* program into the ground. But that’s good. Because you’ll discover in the very next chapter than most Windows programs we write will not be written from scratch. Instead, we’ll take some other program (usually *HelloWorld*) and tweak it according to our needs. So if you just skimmed through this material and said “uh-huh” and “oh-yeah” but you didn’t really follow everything here, go back and read it again (and again, and again, and...). This chapter is truly the basis of everything that follows.