Chapter 1: Readme.txt

Note: This text is being made available in alpha form for proofreading and evaluation purposes only. This book contains known problems, use the information in this book at your own discretion. Please do not distribute this book. Send all corrections to rhyde@cs.ucr.edu.

1.1: Chapter Overview

This chapter provides the "ground rules" for this text. It describes the intended audience, the tools, and the intent of this text. This chapter also describes conventions used throughout the rest of these text.

1.2: Petzold, Yao, Boling, and Other Acknowledgements

Much of the material appearing in this book is patterned after Petzold's and Yao's "Programming Windows..." texts and Douglas Boling's "Programming Microsoft Windows CE" book. Of course, this is not a simple translation of those books to assembly language but this book does attempt to present many of the same topics appearing in these books. The reasons for doing this should be obvious - those texts have been around for quite some time and tens of thousands of C programmers have validated their approach. It would be foolish to ignore what what worked so well for C programmers when writing this book for assembly programmers.

As mentioned above, I have not cloned these existing books and swapped assembly instructions for the C statements appearing therein. Besides the obvious copyright violations and other ethical issues, assembly language is fundamentally different than C and Win32 programming in assembly language needs to be different to reflect this. Another issue is the legacy associated with those books. The original edition of Petzold's book appeared sometime around 1988. As such, the text carries a lot of historical baggage that is interesting to read, but is of little practical value to an assembly language programmer in the third millennium. Boling's book, while much better in this is regard, is targeted towards Windows CE, which has several limitations compared to the full Win32 API. I've cut out lots of the anecdotes that contribute little to your knowledge of Win32 assembly. For example, since Petzold's book was written in the very early days of Windows (1.0!), it contains a lot of user interface explanations and other GUI stuff that most people take for granted these days. I've dropped a lot of this material as well (after all, do you really need to be taught what a scroll bar is and what you would use it for?) If you're not comfortable with the Windows user interface, this document is not the place to start your journey into the world of Windows.

I should also note the contribution by Iczelion's Win32 Assembly Tutorials to this text. These nifty programs and documents got me started and taught me how to write "pure" win32 assembly language programs. I could not have started this text without the availability of these tutorials (and several others like them). Also, lots of thanks to Steve Hutchesson for doing the pioneering work on Win32 assembly language with his "MASM32" package, header files, and libraries. Also, lots of thanks to all of those who post answers to the Win32ASM Community messageboard. Your answers have provided a lot of help that made this book possible.

1.3: Ground Zero: The Programmer's Challenge

If you're coming at this with no prior Win32 programming experience, don't expect to be writing Windows applications in a short period of time. There is a tremendous learning curve to mastering the Win32 API (application programmer's interface) set. Petzold estimates this at six months for an established C programmer. I person-

ally don't have a feeling for this because (a) I had some prior experience with systems like Delphi and C++ Builder before attacking the raw Win32 APIs, and (b) I've spread my study of this subject out over several years, learning the material in a piecemeal fashion. Nevertheless, I can certainly vouch for the fact that learning the Win32 API is a lot of work and it is not something you're going to master in a few weekends of work.

Die-hard assembly programmers are going to find Windows a strange place to work. Under DOS, where assembly programmers have complete control of the machine, speed was the hallmark of the assembly language programmer. It was very easy to write programs that ran two to ten times faster than their HLL (high level language) counterparts. Under Windows, this task is much more difficult because a large percentage of the time is spent in Windows itself. Even if you reduce the execution time of your code to zero, Windows still consumes a dramatic portion of the CPU cycles. Hence it is very difficult to optimize whole programs; only those special applications that are compute intensive will see a big benefit in performance when writing in assembly rather than HLLs. For this reason, most programmers who use assembly language under Windows write the bulk of their application in a high level language like C, dropping into assembly for the time-critical functions. There are, however, other benefits to using assembly language besides speed, and the mixed-language approach tends to negate those benefits. So this book will concentrate on applications that are written entirely in assembly language (quite frankly, there really is no need for a book specifically on mixed language programming under Windows, since most of the real "Windows" stuff winds up getting done in the high level language in those environments).

Another reason for writing applications in assembly is the short size of the resulting applications. Some might argue that such efficiency is wasted on today's machines with tens (and hundreds) of gigabytes of disk storage and hundreds of megabytes of RAM. However, it's still impressive to see a 700 byte "Hello World" program blow up to a 100 KByte executable when you implement it in a HLL. Nevertheless, there is little need for the world's smallest "Hello World" program, and real Windows applications tend to be large because they contain lots of data, not because they contain lots of code. Once you get to the point that you're writing real applications, assembly language programs aren't tremendously smaller (i.e., an order of magnitude) than their HLL counterparts. Perhaps half the size, but not amazingly smaller.

There are two real reasons I can immediately think of for writing Win32 programs in assembly language: you don't know one of these other HLLs (nor do you want to learn them), or you want to have full control over the code your program executes. This text does not attempt to evangelize the use of assembly language programming under Windows. If you think people are nuts who attempt this, stop reading right now and go do something more useful. If you have some reason for wanting to write complete applications in assembly language, then this book is for you and I'm not going to question your reasons.

If you are a die-hard assembly programmer that tries to squeeze every last cycle out of your code and you're offended by inefficiencies that exist in commercial code, let me warn you right away that there will be many things that you won't like about this book. I'm a firm believer in writing quality, correct, and easy to maintain assembly code. If that means I lose a few cycles here and there, so be it. I've even been known, now and then, to stick extra code into my programs to verify their proper operation (e.g., asserts). I'm also a big fan of good formatting, decent variable names, and good programming style. If "structured programming" and long variable names drive you nuts, you should put this book down and go back to DOS. Hopefully, you'll find my programming style a refreshing change from a lot of the Win32/assembly stuff that's out there.

Writing good, user-friendly, applications under Windows is a lot of work; a lot more work that is required to write a typical DOS (or UNIX, or other console-based) application. The combination of assembly language (more work for the programmer) and Windows (more work for the programmer) spells a lot more work for the programmer. However, if you take the time to do it properly, assembly language programming under Windows definitely has its rewards.

1.4: The Tools

Although a certain amount of masochism is necessary to want to write assembly language programs under Windows, there's a big difference between those who just want to write assembly code under Windows and those who insist on doing it the most painful way possible. Some people believe that the only "true" way to write assembly code under Windows is to manually push all the parameters themselves and call their Win32 API routines directly. Even MASM doesn't require this level of pain. To reduce your pain considerably, this book uses the High Level Assembler (HLA). HLA is public domain (i.e., free) and readily available (if you're reading this document, you probably have access to HLA). HLA reduces the effort needed to write Windows assembly programs without your giving up the control that assembly provides. If you are truly sick and really want to manually push all the Win32 API parameters on the stack prior to a call, HLA will let you do this. However, this book is not going to teach you how to do it that way because it's a ton of work with virtually no benefit. If you're a masochist and you want to write the "purest" assembly possible, this book is not for you. For the rest of us though, let the journey begin...

This text uses the following tool set:

- The HLA high level assembler
- The HLA Standard Library
- The RadASM Integrated Development Environment (IDE) for HLA
- The OllyDbg Debugger
- An assembler capable of processing HLA output to produce PE/COFF files (e.g., MASM or FASM)
- An appropriate linker that can process the assembler's output (e.g., MS Link)
- A "resource compiler" such as rc.exe from Microsoft
- Some sort of make utility for building applications (e.g., Microsoft's nmake or Borland's make)
- Win32 library modules (either those supplied with the Microsoft SDK or other freely available library modules)

Although the RadASM package provides a complete programmer's text editing system, you may wish to use a text editor with which you're already comfortable. That's fine; most programmer's text editors will work great with HLA and, in fact, certain text editors include an integrated development environment that can support HLA. For example, the UeMake IDE add-in for Ultra-Edit32 supports HLA and is a viable alternative to RadASM. For more details on setting up your text editor as a simple IDE for HLA, please consult your editor's documentation.

A complete set of tools that will let you develop all the code in this book needs to be gathered from several sources. The accompanying CD-ROM contains almost everything you'll need. A few tools you may want to use do not appear on the CD-ROM, for example, you may want to use MASM and the MASM32 resource compiler available by downloading Steve Hutchesson's excellent MASM32 package from his web site at http://www.movsd.com. Between the MASM32 download and the software appearing on the CD-ROM accompanying this book, you'll have all the software you need to develop Win32 assembly language applications.

This book assumes that you have access to a make utility (e.g., Borland's make.exe or Microsoft's nmake.exe). If you do not have a reasonable version of make that runs under Windows, you can obtain one as part of the Borland C++ command-line toolset which is available for free from Borland's website (well, not exactly free, you do have to register with Borland before you can download it, but the cost is free). Go to Borland's web site (www.borland.com) and click on downloads. Then select C++ and download the C++Builder compiler (Borland C++ 5.5). This download includes several useful tools including make.exe, touch.exe, tdump.exe, coff2omf.exe, the Borland C++ compiler, and other software. See the Borland site for more details. The only tool

from this package that this book will assume you're using is the make.exe program (or somebody else's version of make, e.g., Microsoft's nmake.exe).

1.5: Why HLA?

Long-time assembly programmers will probably want to know why this book uses the high level assembler (HLA) rather than MASM32 or one of the other assemblers that run under Windows (e.g., Gas, NASM, FASM, SpAsm, TASM, and so on). There are a couple of reasons for this; this section will explain the reasoning.

From the start, it was clear that the primary choice for the assembler would be either Steve Hutchesson's MASM32 package or HLA. Both of these assemblers are "high level" assemblers that dramatically ease the development of assembly language software under Windows (TASM, by the way, is also a high level assembler). MASM32 and HLA both provide the necessary Windows include and library modules (support that is missing in a coherent form in many other assemblers). HLA and MASM are both documented extremely well (I know this, having written much of the HLA documentation myself). So the choice of one of these two assemblers was fairly clear.

So why choose HLA over MASM32? Perhaps the most obvious reason is that I personally developed the HLA system and I've got my own horn to toot here. I certainly wouldn't be so bold as to claim that's not a part of the reason I've selected HLA for the examples appearing in this book. However, that is really a minor reason for selecting HLA. I designed and wrote HLA as a teaching tool; it is more suitable for teaching assembly language programming (at both beginning and advanced levels) than is MASM. Also, HLA source code is quite a bit more readable than MASM (assuming, of course, that the programmer takes care in creating the program in the first place). HLA also comes with the HLA Standard Library that makes assembly programming, in general, much easier. Another important consideration to many people: HLA is free. Totally free. As in public domain. The full source code to the HLA assembler, standard library, and utilities is available and anyone can do anything they want with it. Although you can download MASM free from various sites on the internet, the legalities of doing so are somewhat questionable. By using HLA in this book, I was able to supply almost all of the software you need on the accompanying CD-ROM without having to obtain licenses and (possibly) charge extra for that code. Last, but certainly not least, HLA supports an option to compile HLA code into MASM assembly code. So those who want to see these examples in "pure" MASM code can run the HLA source code through the HLA compiler to produce a MASM assembly source file. Unfortunately, there is no option that will translate MASM code into HLA source code. Therefore, supplying the examples appearing in this book in HLA source form will appeal to the widest audience.

There are another couple of reasons for supporting HLA that is political, rather than technical, in nature. At one time, MASM was *unquestionably* the standard for 80x86 assembly language programming. Unfortunately, towards the end of the 1990s Microsoft stopped selling MASM as a commercial product and the support for MASM began to wane. Today, by playing a bunch of games, you can manage to (legally) download MASM for free from the Microsoft web site. You can obtain MASM as part of the Microsoft Developer's Network (MSDN) package, and certain versions of MS Visual C++ include it as well. However, it is not available as a separate product and documentation for the assembler is hard to come by. Last, there are certain people who refuse to use MASM for political reasons (open source/free software and all that stuff). HLA is the most powerful and well-supported Win32-capable assembler beyond MASM32, so it makes a lot of sense to use it.

Of course, there is one additional reason for using HLA in the examples appearing in this book: HLA is the most powerful 80x86 assembler around. Originally, I designed HLA as a tool to teach assembly language programming; so some might get the impression that HLA is not going to be as powerful as an assembler like MASM since HLA was designed for beginners and MASM was designed for professionals. This, however, is not true. While I certainly designed HLA to be easy to learn and use, that design required a lot of sophistication behind the scenes. Also, I'm a pretty demanding assembly language programmer and I wanted to make sure that

HLA was a suitable tool for programmers like myself. Having written well over a hundred thousand lines of HLA code (at the time I write this), I can assure you that HLA is quite a bit more powerful than MASM (which is, incidentally, quite a bit more powerful than most of the other assemblers out there).

One interesting aspect to the HLA versus MASM32 question, which this book will explore in greater detail a little later, is that it's not an either/or situation. There is nothing stopping you from using both assemblers on a single project. A valid complaint with using HLA to develop Win32 applications in assembly language is that there is an abundance of MASM32 source code and utilities. By using HLA one seems to give up all this existing material. However, this is not the case. You can link HLA and MASM code together exactly as you would link HLA and HLA code or MASM and MASM code (this should be obvious, since in one mode of operation HLA emits MASM source code for processing by MASM into an object file). So if you've got some library routines written for MASM32, they're easily called from HLA. Likewise, if you've got a "wizard" code generator (like the ProStart program that comes with the MASM32 package) then that code can easily invoke HLA functions. Although HLA will not compile MASM source (and vice versa), you can easily merge the object files that these two assemblers produce. Therefore, MASM and HLA coexist quite well when developing Win32 assembly applications.

1.6: The Ground Rules

This text teaches "old-fashioned" Windows programming using direct calls to the Win32 API. Petzold points out that there are benefits to programming this way (rather than using MFC, VB, Delphi, or one of the newer coding systems that make Win32 programming so much easier). I'll leave it up to Petzold to make that argument. We're going to program Win32 the "old-fashioned" way because that's currently the only way to do it in assembly. When tools like Delphi become available for assembly programmers, I'd strongly recommend you take a look at those tools. Until then, the Win32 API interface is the only way to write assembly code in Windows, so talking about these other schemes is a waste of time. The RadASM IDE package does simplify certain things (like creating forms and the like), so you don't have to create everything by hand, but the bottom line is that there is no "Delphi for ASM" at this time.

Before actually writing this book I made several aborted "first attempts" before settling on the organization I've chosen here. Early on, I'd decided that the naming convention that Microsoft uses (with conventions inherited from C) was absolutely horrible. So I designed a new naming convention for Windows API function, data structure, variable, and constant names that was a lot more readable. Unfortunately, I ultimately realized that such an approach would not serve my readers well because there is no way a single book can teach you everything you need to know about Windows programming. Unfortunately, if the Win32 names that this book doesn't match the names that C programmers (and, therefore, the vast amount of Win32 programming literature) use, it's going to be difficult to take advantage of that plethora of standard Windows documentation. Originally, I had thought that I could get away with renaming things because Borland's Delphi did this. Ultimately, however, I realized that although I was gaining something important (a much better naming scheme), I was losing more than I was gaining by not adopting standard Windows names.

Once I had convinced myself that using Windows' identifiers was the right way to go, I went in and started revising a bunch of code I'd written to using the standard Windows naming scheme. I quickly realized that it is not possible to completely take this approach. First of all, several standard Windows identifiers are reserved words in an assembly language program (or conflict with standard identifiers you'll typically find in an assembly language program). Also, there is the issue of case sensitivity. C/C++ is a purely case sensitive language and many programmers have taken advantage of this fact to create several different identifiers whose only lexical difference is the use of upper or lower case in the identifier (e.g., you'd find symbols like "someid" and "SOMEID" in use in the same program). Even ignoring the horrible programming style such usage represents, using such identifiers in a language like HLA isn't even possible. Ultimately, I settled on using standard Windows identifiers

whenever possible and practical, and switching to a more reasonable name when there were conflicts (e.g., two different identifiers whose spelling differs only by case). Fortunately, such conflicts don't occur frequently, so most of the Windows identifiers you'll find in this text are the standard symbols the Windows SDK (software development kit) and most of the documentation on this planet use. This book carefully documents the exceptions that occur.

This text does not attempt to teach assembly language programming nor does it attempt to teach the HLA language. See my other books and documentation (e.g., *The Art of Assembly Language* and the HLA documentation) for that purpose. This book assumes that you've properly set up HLA and you've managed to compile and execute some simple console applications (e.g., a console version of the venerable "Hello World" program).

It would be nice to write a book on Windows assembly language that doesn't rely on the reader knowing anything other than assembly language. With one exception, this book does not assume that you've got experience with any language other than assembly language (and HLA, in particular). However, as noted earlier, there is a tremendous body of Windows programming knowledge out in the real world that is squarely aimed at the C/C++ programmer. Since this book can only hope to present a fraction of the available Windows programming information, there are going to be times when you'll need to seek an answer from some other source. If you've got an experienced Win32 assembly language programmer or two handy, you can ask them (e.g., on the Win32 Assembly Community Board). However, there are going to be times when you need an answer quick and a guru won't be available. In those instances, your best solution is to consult other Windows programming documentation, even if it is directed at C/C++ programmers. In such instances, it's real handy if you know C or C++ so that other documentation will make sense to you. This book spends one chapter (the next chapter, in fact), discussing how to read C/C++-based documentation and mentally translate it into assembly language. Even if you don't know C, C++, Java, or some other C-like language, reading that chapter may prove helpful should you find yourself in a position where you absolutely must consult some C-based Windows programming documentation. However, none of the remaining material in this book depends upon the information in that chapter, so you can skip it if you don't want to work with C (or if you're already well-versed in Win32/C programming).

1.7: Using Make/NMake

Although RadASM provides a true IDE for HLA that supports projects, browsing, and other nice features, the best way to manage your Win32 assembly projects (even within RadASM) is via a *makefile*. Although *The Art of Assembly Language Programming* goes into detail about using a makefile, this book does not make the assumption that you've read that book (you could have learned HLA and assembly language programming elsewhere). Since the use of *make* is going to be a fundamental assumption in this book (e.g., most examples will include a makefile), it's probably wise to discuss the use of *make* here for those who may be unfamiliar with this program.

The main purpose of a program like make (or nmake, if you're using Microsoft's version of the program) is to automatically manage the compilation and linking of a multi-module project. Although it is theoretically possible to write a single, self-contained, assembly language source file that assembles directly to an executable file, in practice this is rarely done[1]. Instead, programs are usually broken up into separate source files by logical func-

1. The SpAsm assembler, for example, works exactly this way. While SpAsm is fast and flexible, it also has some severe limitations that prevent its use for most people who write practical Win32 applications in assembly language; in particular, SpAsm doesn't allow the linking of external library modules. This means that SpAsm forces the programmer to include every little utility routine in their source file when creating Win32 applications. This is unacceptable to most assembly programmers. However, if the thought of not being able to link in third-party code doesn't bother you, you should take a look at the SpAsm system because it does support some other, interesting, features. Unfortunately, this book cannot make use of SpAsm because this book teaches solid software engineering and the ability to link in separately compiled object modules is an important requirement for code reuse and modularity.

tion. In order to save time during development, you don't always have to recompile every source file that makes up the application. Instead, you need only recompile those source files that have been changed (or depend upon changes in other source files). This can save a considerable amount of time during development if your project consists of many different source files that you're compiling and linking together and you make a single change to one of these source files (because you will only have to recompile the file you've changed rather than all files in the system).

Although using separate compilation reduces assembly time and promotes code reuse and modularity, it is not without its own drawbacks. Suppose you have a program that consists of two modules: *pgma.hla* and *pgmb.hla*. Also suppose that you've already compiled both modules so that the files *pgma.obj* and *pgmb.obj* exist. Finally, you make changes to *pgma.hla* and *pgmb.hla* and compile the *pgma.hla* file *but forget to compile the pgmb.hla file.* Therefore, the *pgmb.obj* file will be *out of date* since this object file does not reflect the changes made to the *pgmb.hla* file. If you link the program's modules together, the resulting executable file will only contain the changes to the *pgma.hla* file, it will not have the updated object code associated with *pgmb.hla*. As projects get larger they tend to have more modules associated with them, and as more programmers begin working on the project, it gets very difficult to keep track of which object modules are up to date.

This complexity would normally cause someone to recompile *all* modules in a project, even if many of the object files are up to date, simply because it might seem too difficult to keep track of which modules are up to date and which are not. Doing so, of course, would eliminate many of the benefits that separate compilation offers. Fortunately, the make program can solve this problem for you. The make program, with a little help, can figure out which files need to be reassemble and which files have up to date .OBJ files. With a properly defined *make file*, you can easily assemble only those modules that absolutely must be assembled to generate a consistent program.

A make file is a text file that lists compile-time dependencies between files. An .EXE file, for example, is *dependent* on the source code whose assembly produce the executable. If you make any changes to the source code you will (probably) need to reassemble or recompile the source code to produce a new executable file[2].

Typical dependencies include the following:

- An executable file generally depends only on the set of object files that the linker combines to form the executable.
- A given object code file depends on the assembly language source files that were assembled to produce that object file. This includes the assembly language source files (.HLA) and any files included during that assembly (generally .HHF files).
- The source files and include files generally don't depend on anything.

A make file generally consists of a dependency statement followed by a set of commands to handle that dependency. A dependency statement takes the following form:

```
dependent-file : list of files
```

Example :

```
pgm.exe: pgma.obj pgmb.obj              --Windows/nmake example
```

This statement says that "pgm.exe" is dependent upon *pgma.obj* and *pgmb.obj*. Any changes that occur to *pgma.obj* or *pgmb.obj* will require the generation of a new *pgm.exe* file.

---

2. Obviously, if you only change comments or other statements in the source file that do not affect the executable file, a recompile or reassembly will not be necessary. To be safe, though, we will assume *any* change to the source file will require a reassembly.

The make program uses a *time/date stamp* to determine if a dependent file is out of date with respect to the files it depends upon. Any time you make a change to a file, the operating system will update a *modification time and date* associated with the file. The make program compares the modification date/time stamp of the dependent file against the modification date/time stamp of the files it depends upon. If the dependent file's modification date/time is earlier than one or more of the files it depends upon, or one of the files it depends upon is not present, then make assumes that some operation must be necessary to update the dependent file.

When an update is necessary, make executes the set of commands following the dependency statement. Presumably, these commands would do whatever is necessary to produce the updated file.

The dependency statement *must* begin in column one. Any commands that must execute to resolve the dependency must start on the line immediately following the dependency statement and each command must be indented one tabstop. The *pgm.exe* statement above would probably look something like the following:

```
pgm.exe: pgma.obj pgmb.obj
   hla -e:pgm.exe pgma.obj pgmb.obj
```

(The -e:pgm.exe option tells HLA to name the executable file pgm.exe.)

If you need to execute more than one command to resolve the dependencies, you can place several commands after the dependency statement in the appropriate order. Note that you must indent all commands one tab stop. The *make* program ignores any blank lines in a *make* file. Therefore, you can add blank lines, as appropriate, to make the file easier to read and understand.

There can be more than a single dependency statement in a *make* file. In the example above, for example, executable (*pgm.exe*) depends upon the object files (*pgma.obj* and *pgmb.obj*). Obviously, the object files depend upon the source files that generated them. Therefore, before attempting to resolve the dependencies for the executable, make will first check out the rest of the make file to see if the object files depend on anything. If they do, make will resolve those dependencies first. Consider the following *make* file:

```
pgm.exe: pgma.obj pgmb.obj
   hla -e:pgm.exe pgma.obj pgmb.obj

pgma.obj: pgma.hla
   hla -c pgma.hla

pgmb.obj: pgmb.hla
   hla -c pgmb.hla
```

The *make* program will process the first dependency line it finds in the file. However, the files that *pgm.exe* depends upon themselves have dependency lines. Therefore, make will first ensure that *pgma.obj* and *pgmb.obj* are up to date before attempting to execute HLA to link these files together. Therefore, if the only change you ve made has been to *pgmb.hla*, make takes the following steps (assuming *pgma.obj* exists and is up to date).

- The *make* program processes the first dependency statement. It notices that dependency lines for *pgma.obj* and *pgmb.obj* (the files on which *pgm.exe* depends) exist. So it processes those statements first.
- The *make* program processes the *pgma.obj* dependency line. It notices that the *pgma.obj* file is newer than the *pgma.hla* file, so it does *not* execute the command following this dependency statement.
- The *make* program processes the *pgmb.obj* dependency line. It notes that *pgmb.obj* is older than *pgmb.hla* (since we just changed the *pgmb.hla* source file). Therefore, make executes the command following on the next line. This generates a new *pgmb.obj* file that is now up to date.

- Having processed the *pgma.obj* and *pgmb.obj* dependencies, make now returns its attention to the first dependency line. Since make just created a new *pgmb.obj* file, its date/time stamp will be newer than *pgm.exe*s. Therefore, make will execute the HLA command that links *pgma.obj* and *pgmb.obj* together to form the new *pgm.exe* file.

Note that a properly written make file will instruct the *make* program to assemble only those modules absolutely necessary to produce a consistent executable file. In the example above, make did not bother to assemble *pgma.hla* since its object file was already up to date.

There is one final thing to emphasize with respect to dependencies. Often, object files are dependent not only on the source file that produces the object file, but any files that the source file includes as well. In the previous example, there (apparently) were no such include files. Often, this is not the case. A more typical make file might look like the following:

```
pgm.exe: pgma.obj pgmb.obj
   hla -e:pgm.exe pgma.obj pgmb.obj

pgma.obj: pgma.hla pgm.hhf
   hla -c pgma.hla

pgmb.obj: pgmb.hla pgm.hhf
   hla -c pgmb.hla
```

Note that any changes to the *pgm.hhf* file will force the make program to recompile both *pgma.hla* and *pgmb.hla* since the *pgma.obj* and *pgmb.obj* files both depend upon the *pgm.hhf* include file. Leaving include files out of a dependency list is a common mistake programmers make that can produce inconsistent executable files.

Note that you would not normally need to specify the HLA Standard Library include files. the Standard Library .lib files, or any Windows library files (e.g., *kernel32.lib*) in the dependency list. True, your resulting executable file does depend on this code, but this code rarely changes, so you can safely leave it out of your dependency list. Should you make a modification to the Standard Library, simply delete any old executable and object files to force a reassembly of the entire system.

The make program, by default, assumes that it will be processing a make file named *makefile*. When you run the *make* program, it looks for *makefile* in the current directory. If it doesn t find this file, it complains and terminates[3]. Therefore, it is a good idea to collect the files for each project you work on into their own subdirectory and give each project its own *makefile*. Then to create an executable, you need only change into the appropriate subdirectory and run the *make* program.

The make program will only execute a single dependency in a make file, plus any other dependencies referenced by that one item (e.g., the *pgm.exe* dependency line in the previous example depends upon *pgma.obj* and *pgmb.obj*, both of which have their own dependencies). By default, the make program executes the first dependency it finds in the makefile plus any dependencies that are subservient to this first item. In particular, if a dependency line exists in the makefile that is not referenced (directly or indirectly) from the main dependency item, then make will ignore that dependency item unless you explicitly request it's execution.

If you want to execute some dependency other than the first dependency in the make file, you can specify the dependency on the make command line when running make from the Windows' command prompt. For example, a common convention in make files is to create a "clean" dependency that cleans up all the files the compile creates. A typical "clean" dependency line for an HLA compilation might look like the following:

---

3. There is a command line option that lets you specify the name of the makefile. See the nmake documentation in the MASM manuals for more details.

```
clean:
   del *.obj
   del *.inc
   del *.bak
```

The first thing you'll notice is that the "clean" item doesn't have a dependency list. When an item like "clean" appears without a dependency list, make will always execute the commands that follow. Another peculiarity to the "clean" dependency is that there (usually) isn't a file named *clean* in the current directory whose date/time stamp the make program can check. If a file doesn't exist, then make will assume that the file is always out of date. A common convention is to specify non-existent filenames (like *clean*) in a makefile as commands that someone would explicitly execute from within make. Of course, such usage (generally) assumes that you don't actually build a file named "clean" (or whatever name you choose to use).

Since, by default, you typically don't want to execute a command line "clean" when running make, you wouldn't usually place the *clean* dependency first in the make file (nor would you typically refer to *clean* within some other dependency list). Since make doesn't normally execute any dependency items that aren't "reachable" from the first dependency item in the make file, you might wonder how you'd tell make to execute the *clean* command. To specify the execution of some dependency other than the first (default) item in the make file, all you need to is specify the target you want to create (e.g., "clean") on the make command line. For example, to execute the *clean* command, you'd using a Windows command prompt statement like the following:

```
make clean
```

This command does not tell make to use a different make file. It will still open and use the file named "makefile" in the current directory[4]; however, instead of executing the first dependency it finds in *makefile*, it will search for the target "clean" and execute that dependency.

By convention, most programmers use the first dependency in a make file to build the executable based on the current build state of the program (that is, it will compile and link only those files necessary to create an up-to-date executable). Most programmers, by convention, will also include a "clean" target in their make file. The *clean* command deletes all object and intermediate files that the compiler generates; this ensures that the next build of the program will recompile every source file in the project, even if the original objects (and other targets) were up-to-date already. Doing a clean before building the application is useful when you've changed something that is not listed in the dependency lists but on which the final executable still depends (like the HLA Standard Library). Doing a *clean* is also a good way to do a sanity check when you're running into problems and you suspect that the dependency lists aren't completely correct.

Beyond *clean* there aren't too many "standard" target definitions you'll see programmers using in their make files, though it's common for different make files to have some additional commands beyond building the default target and cleaning up temporary compiler files. Throughout this book, the make files associated with each project will generally provide the following dependency/commands (these roughly correspond to make options in the RadASM package):

build: This will be the default command (i.e., the first command appearing in the makefile). It will build an executable by building any out-of-date files and linking everything together. A typical *build* dependency will look like this:

```
build: pgm.exe
```

4. You can tell make to use a different file by specifying the "-f" command line option. Check out make's documentation for more details.

This tells *make* to go execute the dependency for *pgm.exe* (which would normally be the default dependency in the file).

buildall: This command will rebuild the entire application. It begins by doing a clean, and then it does a build. This command generally takes the following form:

```
buildall: clean pgm.exe
```

compileRC: This command will compile any resource files into .RES files. Though the current example does not have any resource files, a typical entry in the make file might look like the following:

```
compileRC: pgm.rc
   rc pgm.rc
```

syntax: This command will compile any HLA into .ASM files just to check their syntax. Using the *pgma.hla/pgmb.hla* example given earlier, a typical compile dependency line might look like the following:

```
syntax:
   hla -s pgma.hla pgmb.hla
```

run: This command will build the executable (if necessary) and then run it. The dependency line typically looks like the following:

```
run: pgm.exe
   pgm <<any necessary command line parameters>>
```

clean: This is the command that deletes any compiler/assembler/linker produced temporary files, backup files, and the executable file. A typical clean command is

```
clean:
   del *.obj
   del *.inc
   del *.bak
   del pgm.exe
```

The purpose behind these various make commands will become clear in the chapter on the RadASM integrated development environment. Their main purpose is to interface with RadASM creating an HLA IDE (Integrated Development Environment).

One nice feature that a standard *make* program provides is *variables*. The make program allows you to create textual variables in a make file using the following syntax:

```
identifier=<<text>>
```

All text beyond the equals sign (=) to the end of the physical line [5] is associated with the identifier and the make program will substitute that text whenever it encounters $(identifier) in your text file. This behavior is quite similar to `TEXT` constants in the HLA language. As an example, consider the following *make* file fragment:

5. If you need more text than will physically fit on a single line, place a backslash at the end of the line to tell make that the line continues on the next physical line in the make file. The make program removes the new line characters between the two lines and continues processing.

```
sources= pgma.hla pgmb.hla
executable= pgm.exe

$(executable): $(sources)
   hla -e:$(executable) $(sources)
```

Because of the textual substitution that takes place, this is equivalent to the following *make* file fragment:

```
pgm.exe: pgma.hla pgmb.hla
   hla -e:pgm.exe pgma.hla pgmb.hla
```

You can even assign variable names from the make command line using syntax like the following:

```
make executable=pgm.exe sources="pgma.hla pgmb.hla"
```

This is an important fact we ll use because it allows us to create a generic makefile that RadASM can use to compile a given project by simply supplying the file names on the command line.

One problem with using makefile variables in this manner is that you lose one of the major advantages of using separate compilation: faster processing. If you specify a variable such as $(sources) as a parameter to the HLA.EXE command in a makefile, HLA will assemble all the source files, not just the ones that are older than their corresponding object files. This obviates the whole purpose for using a makefile in the first place (may as well use a batch file if we re going to force the recompilation of all the files in the project). What we need is a smarter generic make file that only compiles those files necessary in a project while allowing a controlling process (i.e., RadASM) to specify exactly what files to use.

Various versions of the *make* program provide a facility to deal with this problem: implicit rules. Unfortunately, various versions of *make* have minor syntactical differences that hinder the creation of a universal makefile that will work with every version of *make* out there. Nonetheless, with a little care and the use of some advanced make features, it is possible to come up with a make file that works reasonably well with a wide range of different make programs. The examples you re about to see work fine with Borland s *make.exe* program and Microsoft s *nmake.exe* program, they should also work fine with FSF/GNU s *make* program[6].

Implicit rules in a make file describe how to convert files of one type (e.g., .hla files) into files of another type (e.g., .obj files). An implicit rule uses the following syntax:

```
.fromExt.toExt:
   command
```

Where *.fromExt* is the file type (extension) of the source file and *.toExt* is the file type of the destination file to produce. The *.fromExt* must begin in column one. The command is a command line operation that (presumably) translates the source (target) file into the destination (dependent) file. There must be at least one tab in front of this command.

The important thing to note about implicit rules is that they operate on a single file at a time. For example, if you supply the implicit rule .hla.obj to tell *make* how to convert HLA files into object files, and your project includes the files *a.hla*, *b.hla*, and *c.hla*, then this implicit rule will execute the command(s) you specify on each of these files. There is one question, though, how does the command know what file the implicit rule wants it to process? The command can determine this by using a couple of special *make* variables[7]: $@ and $< . When-

6. If you do not have some version of make available, you can obtain a copy of Borlands make.exe from their website (http://www.borland.com). You may download a newer version of Borland s make as part of their BC++ v5.5 trial edition.

ever make encounters the special variable $@ within a command attached to an implicit rule, it substitutes the target name and extension for this variable. So if the implicit rule in this example is processing the files *a.hla, b.hla,* and *c.hla*, then make will expand $@ to *a.obj, b.obj,* and *c.obj*, respectively, when processing these three source files. The special make variable $< expands into the source filename for the given command. Therefore, when individually processing the *a.hla, b.hla,* and *c.hla* files, the $< variable expands to *a.hla, b.hla,* and *c.hla*, respectively. So consider the following implicit make rule:

```
.hla.obj:
   hla -c $<
```

Given the list of files *a.hla, b.hla,* and *c.hla,* this single command is equivalent to the following:

```
a.obj: a.hla
   hla -c a.hla

b.obj: b.hla
   hla -c b.hla

c.obj: c.hla
   hla 0c c.hla
```

Note that make will only execute an implicit rule if you don t supply an explicit rule to handle the file conversion. Consider the following make file fragment:

```
.hla.obj:
   hla -c $<

a.obj: a.hla
   hla -s a.hla
   ml -c -Zi a.asm
```

Given the three files *a.hla, b.hla,* and *c.hla,* this makefile sequence will use the implicit rule to process files *b.hla* and *c.hla* but it will use the explicit rule given in this example to process the *a.hla* file.

Most versions of *make* include a pre-defined set of implicit rules that they will use if you don t explicitly supply some file dependencies in your make file. It is a real good bet, however, that few of these *make* programs are aware of HLAs existence, so it is unlikely that the *make* program will recognize HLA files, by default. Worse, many of these *make* programs do understand various file types (e.g., *.asm* and *.obj*) that HLA produces and they may react adversely to the presence of these files in your project directory. Therefore, we need some way of disabling the existing implicit rules while enabling our new ones. The special make `.SUFFIXES` variable takes care of this. The `.SUFFIXES` variable is a string that contains a list of all the suffixes (file types) that make recognizes for implicit rules. Our first task is to clear this string variable (to dissociate all the old types) and then add the file types (extensions) that we want *make* to recognize. This is done in two steps:

```
# Note: makefile comments begin with a "#" and continue to the end of the line.
#
# Clear out the existing .SUFFIXES list:

.SUFFIXES:
```

---

7. By the way, *make* variables are often called macros by the various make vendors.

```
.SUFFIXES: .hla .obj .exe
```

After clearing and setting up the *.SUFFIXES* variable, make will be capable of processing *.hla* files.

If you don t supply any explicit rules in a make file, then make will process every file in the current subdirectory that an implicit rule can handle. If you supply at least one explicit rule, then make will process only that explicit rule (or the first such explicit rule appearing in the file). Because we ll generally want to control which files that *make* processes by supplying a file list on the command line, we need to add at least one rule to a makefile to give *make* this kind of control. Consider the following (complete) makefile:

```
.SUFFIXES:

.SUFFIXES: .hla .obj .exe

.hla.obj:
   hla -c $<

$(executable): $(objects) $(libraries)
   hla -e:$(executable) $(objects) $(libraries)
```

A typical command-line invocation of make using this makefile might look like this:

```
make executable=a.exe "objects=a.obj b.obj c.obj"
```

(Because we don t define the symbol *libraries*, it expands to an empty string in the makefile.)

There are a couple of problems with this makefile we ve created thus far. First of all, *make* chokes on this file if you don t supply command line definitions for *objects* and *executable*. Second of all, it requires that you specify the object file names (e.g., *a.obj, b.obj,* and *c.obj*) rather than the source file names; this is a minor point, but the user of the make file is more likely to supply the full source file names rather than the object file names.

The first problem is easily handled by adding a couple of conditional statements to the makefile. Many variants of *make* (including Borland and Microsoft s versions) support *!ifndef* and *!endif* to let you test for a variable definition within a *make* file (and take some action if the symbol is not defined)[8]. Here s a possible extension to the current make file we re developing that takes this into account:

```
.SUFFIXES:

.SUFFIXES: .hla .obj .exe

!ifndef files
files=Error_you_need_to_supply_a_files_command_line_parameter.hla
!endif

!ifdef executable
files=Error_you_need_to_supply_an_executable_command_line_parameter.exe
!endif


.hla.obj:
   hla -c $<
```

8. If your version of make doesn t support this feature (or anything like it), you re probably better off using batch files rather than *make* files to build an HLA project.

```
$(executable): $(objects) $(libraries)
   hla -e:$(executable) $(objects) $(libraries)
```

If you type make all by itself on the command line and you fail to supply the files=... or executable=... command line parameter, this makefile will use the strings appearing in this example as these filenames. This invariably produces an error when HLA cannot find the specified files, or HLA will produce a peculiar executable name. In either case, it should be fairly obvious what went wrong.

You might be tempted to try something like the following:

```
!ifndef files
files:
   echo Error you need to supply a 'files=...' command line parameter
!endif
```

This approach will always produce an error message and abort execution of the makefile if you do not supply a files=... command line parameter to the *make* program. However, it turns out that there are some times when we don t want to supply such a command line parameter, consider the following makefile:

```
.SUFFIXES:

.SUFFIXES: .hla .obj .exe

!ifndef files
files=Error_you_need_to_supply_a_files_command_line_parameter.hla
!endif

!ifdef executable
files=Error_you_need_to_supply_an_executable_command_line_parameter.exe
!endif


.hla.obj:
   hla -c $<

$(executable): $(objects) $(libraries)
   hla -e:$(executable) $(objects) $(libraries)

clean:
   del *.obj
   del *.inc
   del *.asm
   del *.link
```

If the user types make clean at the command line hoping to delete all the existing object, assembly, include, and link files, this particular make file will work just fine. However, were you to substitute the `!ifndef` directive with the *type* command, then make clean would always trigger that `!ifndef` and *make* would simply print the error message and quit without processing the clean request. By actually defining files to contain some value (that will generate an error later on), this makefile allows someone to enter a command like make clean and *make* will process the request, as desired.

Although this section discusses the make program in sufficient detail to handle most projects you will be working on, keep in mind that the make program provides considerable functionality that this chapter does not

discuss. To learn more about the *nmake.exe* program, consult the appropriate documentation. This section has briefly touched upon some rather advanced uses of the *make* program. In fact, make has many other features that you might want to take advantage of. For more details, consult the vendor s documentation accompanying the version of make that you re using. This book will assume that you re using Borland s *make* (version 4.0 or later) or some version of Microsoft s *nmake*. Every make file in this book has been tested with both of these versions of make. These make files may work with other versions of *make* as well, but given the advanced features this makefile uses, it s fairly certain this makefile will not work with all versions of *make* out there. If you don t already have a copy of make, note that you can download Borland s make as part of the Borland C++ 5.5 compiler (see the directions for downloading this file earlier in this chapter).

Because of the variations in the way different make programs work, the makefiles appearing in this book will be relatively simple, not taking advantage of too many special make features. The generic makefile we ll usually start with looks like this:

```
# Note: "<<file>>" represents the project/file name. You will replace this text by
# the appropriate name when creating a custom makefile for your project.

build: <<file>>.exe

buildall: clean <<file.exe>>

# The following entry must be modified if the project needs to compile a resource file.

compilerc:
   echo No Resource Files to Process

syntax:
   hla -s <<file>>.hla

run: <<file>>.exe
   <<file>>

clean:
   del /F /Q tmp
   del /F /Q *.exe
   del /F /Q *.obj
   del /F /Q *.link
   del /F /Q *.inc
   del /F /Q *.asm
   del /F /Q *.map

<<file>>.exe: <<file>>.hla wpa.hhf
   hla $(debug) -p:tmp -w -c <<file>>
```

The `$(debug)` variable will be defined on the make command line if the makefile is to create a debug version of the software (we'll discuss how to do this later).

1.8: The HLA Integrated Development Environment

This book uses Ketil Olsen's RadASM integrated development environment for HLA/Win32 program development. The RadASM package provides a programmer's text editor, a project manager, a build subsystem, and several other tools you will find useful. Note, however, that you can use another IDE system that works with HLA (e.g., the UeMake add-on for Ultra-Edit32) or you could use your favorite text editor and run HLA from the

command line. The choice is up to you. A little bit later, this book will describe the RadASM IDE for HLA for those who want to use an IDE but don't have a particular preference.

1.9: Debugging HLA Programs Under Windows

This book recommends the use of Oleh Yuschuk's OllyDbg debugger for Windows when debugging assembly language programs written in HLA. A whole chapter on the use of this tool will appear later in this book.

1.10: Other Tools of Interest

There are many generic programs that a Windows assembly language programmer might find interesting. For example, Steve Hutchesson's MASM32 package contains several utilities a Windows assembly language programmer will find useful. A quick search on the internet will locate many other utilities you'll find useful as well. This book won't spend too much time on tools other than those that have already been mentioned (there are some space restrictions here!), but that shouldn't stop you from locating and using programs you find useful.

1.11: Windows Programming Documentation

As this chapter mentions, there is no way a single book of any practical size is going to tell you everything you need to know about Windows assembly language programming. Unfortunately, documentation on writing Windows applications in assembly is difficult to come by. There are, however, several sources of information you may find helpful. We'll take a look at a couple of them here.

First, you should get a copy of Petzold and Yao's "Programming Windows..." book. Yes, it's written for C programmers, not assembly programmers. However, much of the information that appears in the book is language-independent and it's easy enough to mentally translate the C descriptions into assembly language (e.g., see the next chapter). No serious Win32 API programmer should be without this text, regardless of which language they use.

Microsoft also provides a large database of Win32 programming information with their MSDN (Microsoft Developer's Network). The MSDN information is available on CD-ROM with nearly every Microsoft programming tool. You can also purchase this information separately from Microsoft and it's also available free on-line at www.msdn.com. Among other things, the MSDN database describes the semantics of each and every Win32 API call that you can make; it's written for C programmers (like Petzold's book), but when you want to know what a particular Win32 API function does, the MSDN data base is a good place to look.

On the CD-ROM accompanying this textbook, you'll find some documentation describing most of the Win32 API class from an HLA perspective. This information is effectively the same as what you'll find in the MSDN system, but it's targeted to HLA users rather than C users.

The Iczelion Tutorials also provide a lot of useful information for Windows assembly language programmers. The Iczelion Tutorials were originally written for MASM32 users, though you can find HLA translations of many of those tutorials on the CD-ROM accompanying this book (the MASM32 versions also appear on the CD-ROM.). There are also several web sites dedicated to Win32 programming in assembly language. Searching the internet via your favorite search engine for "Win32 assembly" should turn up a lot of hits. You should also visit Webster at http://webster.cs.ucr.edu and check on the "links" page. There are links to several good Win32 assembly pages from Webster.

Chapter 2: Advanced HLA Programming

Note: This text is being made available in alpha form for proofreading and evaluation purposes only. This book contains known problems, use the information in this book at your own discretion. Please do not distribute this book. Send all corrections to rhyde@cs.ucr.edu.

2.1: Using Advanced HLA Features

This book uses several advanced features in the HLA language. Chances are pretty good that unless you ve spent a lot of time studying HLA prior to reading this book, you re not going to be familiar with many of these features. Now before you complain about the choice of using these features, and how this book should have avoided them in order to make the material more accessible to less than expert HLA programmers, just keep in mind that many of these advanced features were added to the HLA language specifically to support Windows programming. Therefore, avoiding these features would prove to be counter-productive.

Much of the advanced HLA programming we re going to wind up doing involves HLA compile-time language statements (including the macro processor), data declarations, high-level control structures, and high-level procedure call syntax. While *The Art of Assembly Language* covers each of these topics, this book isn t going to make the assumption that you ve carefully read those sections of *The Art of Assembly Language* (or even read AoA at all).Whether or not you ve studied this material in that book, you ll definitely want to carefully read this chapter. It contains important information that the rest of this book is going to use, with an emphasis on those HLA features that are important to Windows programmers.

2.2: HLA's High-Level Data Structure Facilities

One of the primary differences between high level languages and traditional assemblers is the support for *abstract data types* in high level languages versus only supporting low-level machine data types (e.g., bytes, words, and double words) in assembly language. One of the hallmarks of a high level assembler (like HLA, MASM, or TASM) is support for abstract data types and user-defined data types. The HLA assembler provides about the richest set of built-in data types you ll find in any assembler plus the ability to easily create new data types as the need arises. Because Windows uses sophisticated data structures in calls to Win32 API functions, you ll want to familiarize yourself with some of HLA s structured data types in order to write code that interfaces better with Windows.

2.2.1: Basic Data Types

Whereas a traditional assembler provides a few machine-level data types based on the sizes of those types, and then leaves it up to the assembly language programmer to manually interpret those types in a program, HLA provides a rich set of basic data types that denote the purpose of the object in addition to its size. For example, with a traditional assembler you might declare a variable to be a byte object; within your assembly language source code it is up to you to decide whether you want to treat that byte as a character variable, an unsigned integer, a boolean object, a signed integer, an enumerated data type, or some other value that you can represent with eight bits. With HLA, you can actually declare your variables to be `char, boolean, enum, int8, uns8,` etc., and make the intent of that variable s usage much clearer; HLA will even do some type checking when you use these data types to help find logical errors in your programs.

The first thing to note about HLAs abstract data types is that HLA *is an assembly language*. At the machine instruction level, the 80x86 CPU works with bytes, words, double-words, and other such primitive machine data types. Once you ve loaded an `int8` object into AL, for example, it is up to you to choose the correct machine instructions to treat that object as a signed integer. There is nothing stopping you from treating this as an unsigned integer object, as a character value, as a boolean value, etc. Do not, however, get the impression that HLAs abstract data typing facilities serve little purpose beyond providing documentative services that make your programs a little easier to read. HLA provides high-level control structures and procedure calling syntax, as well as a built-in compile-time language, that can take advantage of HLAs type-checking facilities. To achieve maximum benefit from HLA, therefore, you should choose the proper data types for your variables, constants, and other objects in HLA.

Although there are some very good reasons for using abstract (high-level) data types in an assembly language program, HLA doesn t lose sight of the fact that it is an assembler. Sometimes the use of high-level data abstractions interferes with the assembly language programming paradigm. In such situations, you ll want to use low-level machine data types to make your code more efficient. To support such situations, HLA supports the full range of low-level machine data types you ll find on the 80x86 CPU, including bytes, words, double words, quad words, ten-byte words, and long words (128-bit objects/16-byte objects). HLA uses the following type names for these primitive data types:

byte	Eight-byte (one byte) objects.
word	16-bit (two byte) objects
dword	32-bit (four byte) double word objects
qword	64-bit (eight byte) quad word objects
tbyte	80-bit (ten byte) ten byte objects
lword	128-bit (16 byte) long word objects

Note that some individuals might consider HLAs `real32, real64,` and `real80` data types to be primitive because the 80x86 CPU directly supports these types. However, HLA treats these types as abstract data types and performs type checking on them because of the special nature of their data.

The low-level data types provide a type-escape mechanism for HLA. The only type checking that HLA does on these machine level types is to ensure that the size of the object is appropriate for its current use (e.g., HLA will report an error if you try to load a byte object into a word-sized register). Other than the size check, which you can override if you desire, HLA will allow the use of any similarly-sized scalar object in place of one of these data types and you can use an object that has one of these data types in place of a similarly sized scalar object.

Note the use of the term scalar in the previous paragraph. What this means is that you can substitute any like-size single variable for a byte, word, dword, qword, tbyte, or lword object in an HLA statement. You may not, however, directly substitute an array, record, union, class, or other composite type in place of one of these data types, even if the size of the composite object is the same as the primitive data type. For example, you cannot directly substitute a two-byte array in place of a word object (though, using coercion, even this is possible in HLA; you ll find out more about coercion in just a little bit).

2.2.1.1: Eight Bit Data Types

HLA supports the following eight-bit data types, all of which are directly compatible with the `byte` type:

- boolean
- enum (HLA, by default, uses a byte to hold enum objects)
- uns8
- byte
- int8
- char

Directly compatible means that you may supply a byte object anywhere one of these other types is required and you may also supply any one of these other types wherever a byte type is required. Note that these data types are not compatible amongst themselves; that is, you cannot always supply, say, a `char` data type where an `uns8` object is required. The `byte` type is the only data type that is universally compatible with all of these types.

Note that HLA assigns the `byte` type to the 80x86 eight-bit registers, AL, AH, BL, BH, CL, CH, DL, and DH. Therefore, these registers are automatically compatible with any byte-sized object using one of HLAs single byte types.

HLA uses the `boolean` type to represent true/false values. Technically, a `boolean` variable requires only a single bit to represent the two possible values; however, the 80x86 CPU can only efficiently access objects that are multiples of eight bits. Therefore, HLA sets aside a whole byte for a `boolean` object. HLA represents `false` with the value zero and `true` with the value one. Traditionally, programmers have used zero for `false` and any non-zero value to represent `true`; therefore, HLAs definition is upwards compatible to the traditional (i.e., C) definition. In general, when using boolean values, you should always store a zero or one into a boolean variable and test for zero/not zero when checking `boolean` values. This will provide the maximum compatibility with all uses of this data type. Note that arithmetic expressions involving `boolean` operands may produce strange results when combining operands using the AND and OR operators. The next chapter discusses this issue in detail. Just be aware of this for the time being.

Enumerated data types are data types that associate sequential numeric values with symbolic constants. Internally, HLA treats `enum` objects as unsigned integers (e.g., for purposes of comparison), though `enum` objects are not directly compatible with `uns8` objects. An enumerated data type in HLA consists of a list of identifiers to which HLA assigns sequential unsigned integer values, starting with zero. For example, consider the following HLA type declaration:

```
type
   color_t :enum{ red, green, blue }:
```

HLA internally assigns the value zero with `red`, the value one with `green`, and the value two with `blue`. It is important to note, however, that `red, green,` and `blue` are not `uns8` objects. They are of type `color_t` and this will make a difference in certain circumstances. For more details on enumerated data types in HLA, please consult the *HLA Reference Manual*.

HLA uses the `uns8` data type to represent unsigned integer values in the range 0..255. This data type is fundamentally equivalent to the `byte` type except that `uns8` objects are not automatically compatible with the other eight-bit data types. In boolean expressions, HLA always defaults to using unsigned comparisons and conditional jump instructions when the operands are `uns8` operands.

HLA uses the int8 type to represent signed integers in the range -128..+127. In boolean expressions, HLA always uses signed comparisons and conditional jumps with at least one of the operands is signed (note that if you mix an unsigned and signed operand across a relational operator, HLA defaults to use a signed comparison).

Internally, HLA treats the *char* data type as an eight-bit unsigned integer value. Though not directly compatible with the *uns8* type, HLA uses unsigned comparisons and conditional jumps whenever it encounters a char operand in a run-time boolean expression.

As noted earlier, the *byte* type is fundamentally compatible with all other scalar eight-bit types. When two byte types appear in a run-time boolean expression around the same operand, HLA compares the two values using an unsigned comparison. If a byte operand appears with another eight-bit type around some relational operator in a run-time boolean expression, then HLA uses whatever type of comparison is appropriate for that other operand (i.e., unsigned for *enum, char, boolean,* and *uns8* operands or signed for *int8* operands).

There are three major areas where HLA differentiates the types of eight-bit objects: in run-time boolean expressions, in procedure call parameter lists, and in compile-time (constant) expressions. We ll address each of these areas separately in later sections of this chapter.

2.2.1.2: Sixteen-Bit Data Types

HLA supports the following 16-bit data types, all of which are directly compatible with the *word* type:

- int16
- uns16
- wchar
- word

HLA uses *int16* objects to represent signed values in the range -32768..+32767. Whenever HLA encounters an *int16* object in a run-time boolean expression, it uses signed comparisons and conditional jump instructions, even if the operand opposite the *int16* object in a relational expression is an unsigned operand.

HLA uses *uns16* objects to represent unsigned values in the range 0..65535. Whenever HLA encounters an uns16 object in a run-time boolean expression, and the corresponding object opposite the relational operator is not an *int16* value, HLA uses unsigned comparison and jump instructions.

HLA uses *wchar* objects to represent unicode characters. These are unsigned objects (though not directly compatible with *uns16* values) so HLA uses unsigned comparisons and conditional jump instructions whenever you compare *wchar* objects in a run-time boolean expression.

Objects of type *word* are directly compatible with any of the other 16-bit ordinal data types. If an object of type *word* appears in a run-time boolean expression, then HLA will emit code that uses the type of comparison required by the other operand in the relational expression. If the other operand is also of type *word*, HLA uses an unsigned comparison and jump to compare the two operands.

2.2.1.3: Thirty-Two-Bit Data Types

HLA supports the following 32-bit data types, all of which are directly compatible with the *dword* data type:

- dword
- int32
- uns32
- pointer types
- procedure pointer types

HLA also supports the following 32-bit data types, though these types are not directly compatible with the *dword* type:

- string (this is a pointer type)
- unicode (this is a pointer type)
- real32

The behavior of the *uns32* and *int32* types, with respect to the dword word, is similar to the relationship between *uns8/int8* and *byte*, and *uns16/int16* and *word*. Take a look at the previous sections for more details.

HLA generally treats pointer types as *dword* objects within most run-time expressions. In particular, HLA will perform an unsigned comparison when comparing two pointer objects.

Procedure pointer types are also treated just like *dwords* by HLA with one major exception: HLA provides implicit syntax for calling a procedure indirectly using HLAs high level procedure call syntax. So although you can treat a procedure pointer just like a dword object within a run-time boolean/relational expression, you cannot directly call a procedure via a pointer to a procedure in a *dword* variable using HLAs high-level procedure call syntax. Such activity is only available when using procedure pointers (or when specifying an explicit coercion operator). We ll take another look at HLAs procedure pointer objects in a few sections.

Although HLA string variables are pointers, HLA treats strings as a unique data type for the purposes of type checking. In particular, *dword* and *string* objects aren t always interchangeable nor are they even type compatible in all cases. To confuse the issue even farther, certain Win32 API function calls allow you to pass a small integer value (16 bits or less) in place of a string pointer as a function parameter. Therefore, in certain cases, HLA will allow you to substitute a small integer constant in place of an actual string pointer when calling a function (this use, however, is being depreciated, so don t count on it in future versions of HLA). Generally, though, you can use a string variable (not a constant) wherever a *dword* object is expected; you may not, however, always use a *dword* object where a *string* object is expected. This same discussion applies to the *unicode* data type (which is a string of unicode characters).

Because floating point values on the 80x86 are fundamentally different than integer values, HLA does not allow you to mix *real32* and *dword* objects in the same expression. Indeed, HLA doesn t even allow *real32* objects in a boolean/relational expression (though you may pass *real32* objects as procedure parameters). For that reason, we ll not consider *real32* objects here. See *The Art of Assembly Language* for more details concerning floating point arithmetic on the 80x86.

2.2.1.4: Sixty-Four-Bit Data Types

HLA supports a couple of 64-bit data types:

- int64
- real64
- uns64
- qword

As with the 32-bit data types, *qword* is directly compatible with *int64* and *uns64* types. The issue of *int64* and *uns64* compatibility isn t as much of an issue because the 80x86 doesn t directly support 64-bit comparisons

with the integer instruction set[1]. Also like the 32-bit data types, the *real64* type is not directly compatible with the *qword* type, nor may you compare real64 objects within an HLA run-time boolean expression.

2.2.1.5: Eighty-Bit Data Types

HLA supports two 80-bit data types - the *tbyte* type and the *real80* type. Neither type is directly compatible with the other and HLA doesn t allow the comparison of 80-bit objects within a run-time boolean expression. *Tbyte* objects usually hold BCD values and *real80* objects hold floating point values. We ll not consider these two types any farther, here.

2.2.1.6: One Hundred Twenty-Eight Bit Data Types

HLA supports several 128-bit data types, they are:

- cset
- int128
- lword
- real128[2]
- uns128

Though HLA does not support the use of *int128, lword,* and *uns128* objects in a run-time boolean expression (because the 80x86 CPU doesn t support the direct comparison of 128-bit objects using the integer instruction set), you can still pass these objects as procedure parameters and use them in compile-time expressions, so it s worth mentioning that the relationship they share is similar to the *dword/uns32/int32* relationship sans the ability to compare them. The *cset* data type is, technically, a composite data type (character sets can be thought of as an array of 128 bits); therefore, there is no fundamental relationship between an *lword* and a *cset* other than their size. similarly, *real128* data types are also composite objects (it s an array of two real64 objects) so they aren t directly compatible with *lwords*, either. Because Windows doesn t use 128-bit data types very often, we ll not consider these data types any further.

2.2.2: Composite Data Types

HLA supports a fair number of *composite* data types as well as scalar data types. As the name suggests, a composite data type is one that is composed (or made up) of other data types. HLA provides direct support for the following composite data types: arrays, records (structures), unions, classes, pointers, procedure pointers, character sets, and thunks. Character sets, pointers, and procedure pointers are a special case - HLA provides direct support for these composite data types so that you can often treat these types as scalar (though not ordinal) types. Though it s sometimes convenient to treat these objects as scalars, there are times when it s more convenient to think of them as composite objects. Thus we ll consider them again in the following subsections.

---

1. We ll ignore MMX and SSE operands because they require special instructions that HLA s run-time boolean expressions do not support.
2. Real128 is actually a composite data type, specifically a record, that HLA supports for SSE/2 compatibility. In theory, it is not a scalar data type. However, we ll mention it here just for completeness.

2.2.2.1: HLA Array Types

HLA uses a high-level-like syntax for the declaration of arrays in your assembly language programs. A typical (run-time) array variable declaration might look like the following:

```
static
   runTimeArray :int32[ 16];
```

This declaration sets aside storage for a sequence of 16 32-bit integers in memory. The name *runTimeArray* refers to the address of the first element of this array. The next variable immediately following this array in memory will be at an address that is (at least) 64 bytes later (16 array elements times four bytes for each element is 64 bytes). You can even declare multi-dimensional arrays in HLA using a high level like syntax, e.g.,

```
type
   two_dim_array : char[  4, 16 ]; // 64 bytes of storage
   three_dim_array : dword[  4, 2, 2 ]; // Also 64 bytes of storage
   four_dim_array : real32[  2, 2, 2, 2 ]; // Also 64 bytes of storage
```

Note that multi-dimensional arrays consume as much storage as the product of their dimensions, also multiplied by the size of a single array element.

As is true in all assembly languages, indexes into an array start at zero and the last element (of a particular dimension or index) ends at the array bounds minus one. For example, the elements of the *runTimeArray* example given earlier range from index zero through 15.

One very fundamental point, which is the cause of much confusion to assembly language programmers and especially to HLA programmers, is that you do not index into an array using the same syntax and concepts as you do in a high level language. The fact that HLA uses a high-level language-like syntax for array declarations falsely propagates this idea. The fact that HLA's compile-time language works the way high level languages do, in direct contrast to the run-time (i.e., 80x86 assembly) language also aids in this confusion. *Don't get caught in this trap!*

Although both *The HLA Reference Manual* and *The Art of Assembly Language Programming* discuss accessing elements of arrays, emails and posts to assembly language related newsgroups indicate that this is one area that causes problems for assembly programmers (especially beginning assembly programmers). So it's worthwhile repeating some of that information here.

The most fundamental mistake people make when indexing into array in assembly language code is that they forget that they must multiply the index into the array by the size of an array element. More often than not, a beginning assembly language programmer will translate code like the following:

```
eax = runTimeArray[ i ]; //  C  example
```

into assembly code that looks like this:

```
   mov( i, ebx ); // Must load array index into a 32-bit register to use indexed addressing
   mov( runTimeArray[  ebx ], eax );
```

Nice try, but completely incorrect. The problem is that the 80x86 indexed addressing modes compute *byte offsets from some base address*, not element indexes into an array. That is, the runTimeArray[ebx] addressing mode computes the effective address obtained by adding the current value in EBX to the address associated with the *runTimeArray* label. If EBX contains five and the memory address associated with the *runTimeArray* label is $40_0000, then this addressing mode references the object at address $40_0005 in memory. Note that this is *not*

the address of the sixth element of the *runTimeArray* array. Each element of the *runTimeArray* object consumes four bytes, so the elements appear in memory as follows:

runTimeArray[0]- $40_0000
runTimeArray[1]- $40_0004
runTimeArray[2]- $40_0008
runTimeArray[3]- $40_000C
runTimeArray[4]- $40_0010
runTimeArray[5]- $40_0014
runTimeArray[6]- $40_0018
runTimeArray[7]- $40_001C
runTimeArray[8]- $40_0020
runTimeArray[9]- $40_0024
runTimeArray[10]- $40_0028
runTimeArray[11]- $40_002C
runTimeArray[12]- $40_0030
runTimeArray[13]- $40_0034
runTimeArray[14]- $40_0038
runTimeArray[15]- $40_003C

The address $40_0005 (obtained as the effective address of the expression *runTimeArray[ebx]* when EBX contains five) is actually the address if the second byte of the *runTimeArray[1]* element. This probably isn t what the programmer had in mind.

As you can see in this list, each element of *runTimeArray* is located four bytes in memory apart. This makes perfect sense because each element of the array consumes four bytes of memory and we d like the array elements to occupy adjacent memory cells (that is, each element of the array should immediately follow the previous element of the array in memory). In order to achieve this, you must multiply the array index by the size of each array element (in bytes). Because each element of the array is a double word (four bytes), we have to multiply the index into the array by four and add the base address of the array (that is, the address of the first element) to obtain the actual address of the desired element. In the current example, if we want to access the element at index five (that is, the sixth element of the array), then we need to multiply the index (five) by four before adding in the base address. Five times four is 20 ($14), adding this to the base address produces $40_0014 in the current example. If you ll look at the list for *runTimeArray* element addresses, you ll find that *runTimeArray[5]* does, indeed, sit at address $40_0014 in memory.

Because programs commonly access elements of arrays whose element sizes are one, two, four, or eight bytes long, the 80x86 CPU provides a special set of addressing modes, the scaled indexed addressing modes, that let you easily compute the index into these arrays. These addressing modes use syntax like the following:

```
byteArray[ eax*1 ] -- note that this is functionally identical to byteArray[ ebx ]
wordArray[ ebx*2 ]
dwordArray[ ecx*4 ]
qwordArray[ edx*8 ]
```

The register you specify in the scaled indexed addressing mode must be a general-purpose 32-bit integer register; see *The Art of Assembly Language* for more details.

If you need to access an element of an array whose element sizes are not one, two, four, or eight bytes, then you ve got to manually multiply the index by the size of an array element (typically using a `shl` or `intmul` instruction). For example, if you ve got an array of `real80` objects you ll need to multiply the index into the array by 10 (10 bytes = 80 bits) before accessing the particular element. Here s some sample code that demonstrates this:

```
// Push element "i" of real80Array onto the FPU stack:

   intmul( 10, i, ecx ); // ecx = 10 * i
   fld( real80Array[ ecx ] );
```

Note that you do not use a scaled indexed addressing mode when you manually multiply the index by the element size. You ve already done the multiplication to obtain a byte offset into the array.

Accessing an element of a multidimensional array requires the use of a more complex calculation. In the Windows API world, all two-dimensional arrays are stored in a form known as *row-major order*. Rather than get into the complexities of row-major (versus column-major) addressing here, we ll just use the HLA Standard Library `array.index` function that does multi-dimensional array computations for you automatically. Those who are interested in the low-level implementation of multi-dimensional array access should take a look at the chapter covering arrays in *The Art of Assembly Language Programming*.

The HLA Standard Library `array.index` macro automatically computes the index into an array (with any number of dimensions). Because this is a macro, it directly generates code that is almost as good as hand-written assembly language code (in a few special cases you might be able to generate slightly better code by hand, but most of the time this macro generates exactly the same code you d manually write). To use the `array.index` macro, you must either include *stdlib.hhf* at the beginning of your source file or, at least, include *arrays.hhf*. The `array.index` macro uses the following invocation syntax:

array.index(<32-bit register>, <array name>, <<list of array indices>>);

The first argument must be a 32-bit general purpose integer register (i.e., EAX, EBX, ECX, EDX, ESI, EDI, EBP, or ESP). Generally, you would not use EBP or ESP here (because they have special purposes and they generally aren t available for use in accessing array elements).

The second argument to the `array.index` macro is the name of the array whose elements you want to access. HLA actually supports dynamically allocated multi-dimensional arrays in the arrays.hhf module, we ll not worry about those here (see the HLA Standard Library documentation for more details); instead, this book will simply assume that you ve supplied the name of an array you ve declared in a `static, readonly, storage,` or `var` declaration section, or as a procedure pass by value parameter. Any of the array examples appearing in this section would work fine, for example.

The third (through n^{th}) parameter(s) specify the constants, variables, or registers that you want to use as the indexes into this array. The index objects must all be 32-bit integer values (presumably unsigned, though the macro accepts signed values too).

The `array.index` macro computes the address of the specified array element and leaves that address sitting in the 32-bit register you specify as the first argument in the `array.index` argument list. To access the specified array element, all you need to do is reference the memory location pointed at by that 32-bit register after the `array.index` invocation. Note that you do not have to multiply this value by the size of the array element (that s already done for you) nor do you have to add in the base address of the array (that s also done for you). Here are some examples of accessing array elements using the `array.index` macro:

```
// j = runTimeArray[  i ];

   array.index( ebx, runTimeArray, i );
   mov( [ebx], eax );
   mov( eax, j );

// c = two_dim_array[  i, j ];

   array.index( edx, two_dim_array, i, j );
   mov( [edx], al );
   mov( al, c );

// esi = three_dim_array[  i, j, k ]

   array.index( ecx, three_dim_array, i, j, k );
   mov( [ecx], edx );

// push four_dim_array[  eax, ebx, ecx, edx ] onto the FPU stack:

   array.index( edi, four_dim_array, eax, ebx, ecx, edx );
   fld( (type real32 [edi]) );
```

The HLA compile-time language provides some built-in compile-time functions that return values of interest to an HLA programmer using arrays in an HLA program. During compilation, HLA replaces each of these compile-time functions with a constant that is the result of computing the compile-time function s result. You may use these compile-time functions anywhere a constant is legal in an HLA program. See Table 2-1 for a list of the applicable functions.

Table 2-1: HLA Compile-Time Functions That Provide Useful Array Information

Function	Sample Call	Description
@size	@size(array_name)	Returns the size, in bytes, of the entire array (the run-time size of the array).
@elements	@elements(array_name)	Returns the total number of elements declared for an array. If the array is a multi-dimensional array, this returns the product of all the array bounds for that array (i.e., the total number of elements).
@elementsize	@elementsize(array_name)	Returns the run-time size, in bytes, of a single element of the array.
@arity	@arity(array_name)	Returns the number of dimensions of the array.
@dim	@dim(array_name)	Returns a compile-time constant array where each element of the array specifies the bounds for one of the dimensions of the array.
	@dim(array_name)[index]	Returns the number of elements of a particular dimension of an array (note that index is zero-based).

You can use the compile-time functions in Table 2-1 to automate much of your array calculations. This can make your programs much easier to maintain by avoiding hard-coded constants in your program. For example, suppose you have an array of some type (*myType_t*) whose definition is subject to change as you modify the program. Computing an index into this array is a bit of a problem; for example, if you know that the size of an element in the array is six bytes, you could (manually) compute an index into that array using code like the following:

```
intmul( 6, index, ebx );
lea( eax, myArray[ ebx] );  // Load eax with the address of myArray[ index] .
```

The problem with this approach is that it fails if you decide to change the definition of *myType_t* (*myArray's* type) without manually adjusting this code to deal with the fact that *myArray's* size is no longer six bytes. Now consider the following implementation of this code sequence:

```
intmul( @elementsize( myArray ), index, ebx );
lea( eax, myArray[  ebx ] );
```

This code sequence automatically recomputes the appropriate index into the array regardless of *myArray's* size (because HLA automatically substitutes the correct size for *@elementsize*). About the only drawback to this scheme is that it hides the fact that certain indexes (1, 2, 4, and 8) might be better computed using an 80x86 scaled indexed addressing mode, though it is possible to fix this problem by using a macro (the HLA Standard Library *array.index* function, for example, automatically checks for these special element sizes and adjusts the code it outputs in an appropriate fashion for these special cases).

HLA is interesting insofar as it allows you to declare and use compile-time constant arrays as well as run-time memory arrays. Few other languages, much less assemblers, provide the concept of a compile-time array. HLAs compile-time language makes extensive use of these compile-time arrays and you can use compile-time arrays (also known as array constants) to initialize run-time arrays at program load time (i.e., arrays you declare in the *static* and *readonly* sections). There is, however, one important issue of which you need to be aware - HLA compile-time arrays/array constants use semantics that are appropriate for the HLA compile-time language (which is a high level language). These semantics are different than the semantics for a run-time (assembly language) array. In particular, indexes into array constants are actual array indexes, not byte offsets. Therefore, you do not multiply the index of a constant array by the element size. Furthermore, HLA automatically computes the row-major index into a constant array for you, you do not have to use array.index or manually compute the index yourself. We ll return to this subject a little later in this chapter when we discuss the HLA compile-time language. Just be aware of the fact that compile-time array constants are fundamentally different that run-time assembly language arrays.

2.2.2.2: HLA Union Types

A discriminant union is a data type that combines several different objects so that they use the same storage in memory. In theory, access to each of the variables in a discriminant union are mutually exclusive; because the variables share the same storage in memory, storing a value into one of these variables disturbs the values of the other variables sharing that same memory address. Some high level language programmers use discriminant union types as a sneaky way to retype data in memory. Obviously, this feature is of little value in an assembly language program where is trivially easy to recast data in memory as some other type. However, there are two primary purposes for using a discriminate union, both of which apply in assembly language as well as in high level languages: memory conservation and the creation of variant types. Because many assembler authors perceive discriminant unions as a type casting trick, very few assemblers support the union data type. Unfortunately

(for those assemblers), many Windows data types use unions and this makes using such assemblers inconvenient when working with these Windows data structures. Fortunately, HLA provides full support for discriminant union data types, so this isn t a problem at all in HLA.

HLA implements the discriminant union type using the *union..endunion* reserved words. The following HLA type declaration demonstrates a *union* declaration:

```
type
    allInts:
        union
            i8  :int8;
            i16 :int16;
            i32 :int32;
        endunion;
```

Each entry variable declaration appearing between the *union* and endunion keywords is a *field* of that *union*. All fields in a *union* have the same starting address in memory. The size of a *union* object is the size of the largest field in the union. The fields of a *union* may have any type that is legal in a variable (HLA *var*) declaration section.

Given a *union* object, say *i* of type *allInts*, you access the fields of the *union* by specifying the *union* variable name, a period (dot), and the field name. The following 80x86 *mov* instructions demonstrate how to access each of the fields of an *i* variable of type *allInts*:

```
mov( i.i8, al );
mov( i.i16, ax );
mov( i.i32, eax );
```

A good example of a *union* type in action is the Windows *w.CHARTYPE* data type:

```
type
    CHARTYPE:
        union
            UnicodeChar: word;
            AsciiChar: byte;
        endunion;
```

The Windows *w.CHARTYPE* is a perfect example of a union data structure that encapsulates a couple of mutually exclusive values (that is, you wouldn t use the *UnicodeChar* and *AsciiChar* fields of this union simultaneously). Most Windows applications use either ASCII (ANSI) characters or Unicode characters; few applications would attempt to use both character sets in the same program (other than to convert one form to the canonical form that the application uses). Windows, as a general rule, provides two sets of API functions that deal with characters - one for ASCII and one for Unicode. By playing games with C preprocessor macros and accompanying data types (like *w.CHARTYPE*), Windows allows C programmers to call either set of functions while passing the same set of parameters. Of course, the programmer has to know which character type to supply (i.e., whether to store their data in the *Unicode* or *AsciiChar* fields), but the interface is the same either way.

Most of the unions you ll find in Windows data structures represent mutually exclusive data fields, like the fields belonging to *w.CHARTYPE*. Depending on your circumstances (e.g., whether your program works with ASCII or Unicode characters) you use one or the other of these two fields in your program and you will probably not use the other field at all. Another common use of discriminant union types is to create *variant* objects. A variant object is a variable that has a *dynamic type*, that is, the program determines the type of the object at run-time and, in fact, the variable s type can change under program control. Dynamically typed objects are great when you

cannot anticipate exactly what type of data the user will enter while the program is running. The typical implementation of a variant data type consists of two components: a *tag value* that describes the current (dynamic) type of the object and a discriminant union value that can hold any of the possible types. During program execution, the software uses a *switch/case* statement (or similar logic) to execute appropriate code based on the current type of the dynamically typed object. Although the use of variant types is interesting and useful, Windows doesn t make much use of this advanced feature, so we won t discuss it any farther here.

2.2.2.3: HLA Record (Structure) Types

Records (structures) are probably the most important high level data structure an assembler can provide for Win32 assembly language programmers. In fact, if an assembler does not provide decent, built-in, capabilities for declaring and using records, you simply don t want to use that assembler for developing Win32 programs. Windows APIs make extensive use of records (C structs) and attempting to write assembly code with an assembler that doesn t properly support records is going to be a burden. Fortunately, HLA is not one of these assemblers - HLA provides excellent support for records. In this section we ll explore HLA s support for this important user-definable data type.

The record data type provides a mechanism for collecting together different variables into a physical unit so the program can treat this disparate collection of objects as a single entity. At the most basic level an array is also such a data type. However, the difference between an array and a record is the fact that all the elements of an array must have the same type, this restriction does not exist for the elements (*fields*) of a record. Another difference between an array and a record is that you reference the elements of an array using a numeric index whereas you access the fields of a record by the fields names.

HLA s records allow programmers to create data types whose fields can be different types. The following HLA type declaration defines a simple record with four fields:

```
type
    Planet:
        record
            x       :int32;
            y       :int32;
            z       :int32;
            density :real64;
        endrecord;
```

Objects of type *Planet* will consume 20 bytes of storage at run-time.

The fields of a *record* may be of any legal HLA data type including other composite data types. Like unions, anything that is legal in a *var* section is a legal field of a *record*. Also like unions, you use the dot-notation to access fields of a record object. For example, you could use the following 80x86 instructions to access an object *plnt* of type *Planet*:

```
mov( plnt.x, eax );
mov( edx, plnt.y );
add( plnt.z, ebx );
fld( plnt.density );
```

There are several advantages to using records versus simply creating separate variables for each of the fields of the record. First of all, a record type provides a template that makes it very easy to create new instances (copies) of that record type. For example, if you need nine planets, you can easily create them in an HLA *static* section as follows:

```
static
    Mercury  :Planet;
    Venus    :Planet;
    Earth    :Planet;
    Mars     :Planet;
    Jupiter  :Planet;
    Saturn   :Planet;
    Uranus   :Planet;
    Neptune  :Planet;
    Pluto    :Planet;
```

Imagine the mess you would have if you had to create four separate variables for each of these nine planets (e.g., *Mercury_x, Mercury_y, Mercury_z, Mercury_density*, etc.).

Another solution, of course, is to create an array of records, with one array element for each planet. You can easily do this as follows:

```
static
   Planets :Planet[ 9];
```

Each element of the array will have all the fields associated with the *Planet* data type. Because HLA treats an array name as though it were the first element of the array, you can access the fields of the *Planets* array using the same dot-notation syntax you use to access fields of a single *Planet* object, e.g.,

```
   mov( Planets.x, eax ); // Moves field "x" of Planets[ 0] into eax.
```

If you want to access a field of some element other than the first element of an array of records, you tack on the indexed addressing mode and supply an appropriate index into the array, e.g.,

```
   intmul( @elementsize( Planets ), index, ebx );
   mov( Planets.x[ ebx], eax );
```

Note that the indexed addressing mode specification follows the entire variable s name, unlike most high level languages, like C, where you would inject the array index into the name immediately after *Planets*. Keep in mind that you can also use HLA s array.index macro to index into an array of records, though you will need to coerce the resulting pointer to the *Planet* type in order to access the fields of that record, e.g.,

```
   array.index( ebx, Planets, index );
   mov( (type Planet [ ebx]).x, eax );
```

In addition to saving you the effort of creating separate variables for each field, records also let you encapsulate related data, making it easier to copy record variables as single entities, manipulate record objects via pointers, and pass records as parameters to procedures. For example, it s much easier to pass a single parameter of type *Planet* to some procedure rather than passing the four separate fields that the *Planet* type encapsulates (if you don t think passing four parameters is a big deal, just keep in mind that many Win32 structures, that often appear as procedure parameters, have *dozens* of fields).

Beyond the issues of convenience and efficiency, there is one other important reason for using records in your assembly language programs: they re easier to read and maintain. Assembly language code is hard enough to read as it is; every opportunity you get to produce more readable assembly code is an opportunity you should take.

Record types may *inherit fields* from other record types. Consider the following two HLA type declarations:

```
type
    Pt2D:
        record
            x: int32;
            y: int32;
        endrecord;

    Pt3D:
        record inherits( Pt2D )
            z: int32;
        endrecord;
```

In this example, *Pt3D* (point 3-D) inherits all the fields from the *Pt2D* (point 2-D) type. The *inherits* keyword tells HLA to copy all the fields from the specified record (*Pt2D* in this example) to the beginning of the current record declaration (*Pt3D* in this example). Therefore, the declaration of *Pt3D* above is equivalent to:

```
type
    Pt3D:
        record

            x: int32;
            y: int32;
            z: int32;

        endrecord;
```

In some special situations you may want to override a field from a previous field declaration. For example, consider the following record declarations:

```
type
    BaseRecord:
        record
            a: uns32;
            b: uns32;
        endrecord;

    DerivedRecord:
        record inherits( BaseRecord )
            b: boolean;  // New definition for b!
            c: char;
        endrecord;
```

Normally, HLA will report a duplicate symbol error when attempting to compile the declaration for *DerivedRecord* since the *b* field is already defined via the *inherits(BaseRecord)* option. However, in certain cases it s quite possible that the programmer wishes to make the original field inaccessible in the derived class by using a different name. That is, perhaps the programmer intends to actually create the following record:

```
type
    DerivedRecord:
        record
            a: uns32;    // Derived from BaseRecord
            b: uns32;    // Derived from BaseRecord, but inaccessible here.
```

```
            b: boolean;  // New definition for b!
            c: char;
        endrecord;
```

HLA allows a programmer explicitly override the definition of a particular field by using the *overrides* keyword before the field they wish to override. So while the previous declarations for *DerivedRecord* produce errors, the following is acceptable to HLA:

```
type
    BaseRecord:
        record
            a: uns32;
            b: uns32;
        endrecord;

    DerivedRecord:
        record inherits( BaseRecord )
            overrides b: boolean;  // New definition for b!
            c: char;
        endrecord;
```

Normally, HLA aligns each field on the next available byte offset in a record. If you wish to align fields within a record on some other boundary, you may use the *align* directive to achieve this. Consider the following record declaration as an example:

```
type
    AlignedRecord:
        record
            b:boolean;              // Offset 0
            c:char;                 // Offset 1
            align(4);
            d:dword;                // Offset 4
            e:byte;                 // Offset 8
            w:word;                 // Offset 9
            f:byte;                 // Offset 11
        endrecord;
```

Note that field *d* is aligned at a four-byte offset while *w* is not aligned. We can correct this problem by sticking another *align* directive in this record:

```
type
    AlignedRecord2:
        record
            b:boolean;              // Offset 0
            c:char;                 // Offset 1
            align(4);
            d:dword;                // Offset 4
            e:byte;                 // Offset 8
            align(2);
            w:word;                 // Offset 10
            f:byte;                 // Offset 12
        endrecord;
```

Be aware of the fact that the align directive in a *record* only aligns fields in memory if the record object itself is aligned on an appropriate boundary. For example, if an object of type *AlignedRecord2* appears in memory at an odd address, then the *d* and *w* fields will also be misaligned (that is, they will appear at odd addresses in memory). Therefore, you must ensure appropriate alignment of any record variable whose fields you re assuming are aligned.

Note that the *AlignedRecord2* type consumes 13 bytes. This means that if you create an array of *AlignedRecord2* objects, every other element will be aligned on an odd address and three out of four elements will not be double-word aligned (so the *d* field will not be aligned on a four-byte boundary in memory). If you are expecting fields in a record to be aligned on a certain byte boundary, then the size of the record must be an even multiple of that alignment factor if you have arrays of the record. This means that you must pad the record with extra bytes at the end to ensure proper alignment. For the *AlignedRecord2* example, we need to pad the record with three bytes so that the size is an even multiple of four bytes. This is easily achieved by using an *align* directive as the last declaration in the record:

```
type
    AlignedRecord2:
        record
            b:boolean;              // Offset 0
            c:char;                 // Offset 1
            align(4);
            d:dword;                // Offset 4
            e:byte;                 // Offset 8
            align(2);
            w:word;                 // Offset 10
            f:byte;                 // Offset 12
            align(4)                // Ensures we're padded to a multiple of four bytes.
        endrecord;
```

Note that you should only use values that are integral powers of two in the *align* directive.

If you want to ensure that all fields are appropriately aligned on some boundary within the record, but you don t want to have to manually insert *align* directives throughout the record, HLA provides a second alignment option to solve your problem. Consider the following syntax:

```
type
    alignedRecord3 : record[ 4]
        << Set of fields >>
    endrecord;
```

The [4] immediately following the *record* reserved word tells HLA to start all fields in the record at offsets that are multiples of four, regardless of the object s size (and the size of the objects preceding the field). HLA allows any integer expression that produces a value in the range 1..4096 inside these parenthesis. If you specify the value one (which is the default), then all fields are packed (aligned on a byte boundary). For values greater than one, HLA will align each field of the record on the specified boundary. For arrays, HLA will align the field on a boundary that is a multiple of the array element s size. The maximum boundary HLA will round any field to is a multiple of 4096 bytes.

Note that if you set the record alignment using this syntactical form, any *align* directive you supply in the record may not produce the desired results. When HLA sees an *align* directive in a record that is using field alignment, HLA will first align the current offset to the value specified by *align* and then align the next field s offset to the global record align value.

Nested record declarations may specify a different alignment value than the enclosing record, e.g.,

```
type
    alignedRecord4 : record[ 4]
        a:byte;
        b:byte;
        c:record[ 8]
            d:byte;
            e:byte;
        endrecord;
        f:byte;
        g:byte;
    endrecord;
```

In this example, HLA aligns fields *a, b, f,* and *g* on double word boundaries, it aligns *d* and *e* (within *c*) on eight-byte boundaries. Note that the alignment of the fields in the nested record is true only within that nested record. That is, if *c* turns out to be aligned on some boundary other than an eight-byte boundary, then *d* and *e* will not actually be on eight-byte boundaries; they will, however be on eight-byte boundaries relative to the start of *c*.

In addition to letting you specify a fixed alignment value, HLA also lets you specify a minimum and maximum alignment value for a record. The syntax for this is the following:

```
type
    recordname : record[ maximum : minimum]
        << fields >>
    endrecord;
```

Whenever you specify a maximum and minimum value as above, HLA will align all fields on a boundary that is at least the minimum alignment value. However, if the object s size is greater than the minimum value but less than or equal to the maximum value, then HLA will align that particular field on a boundary that is a multiple of the object s size. If the object s size is greater than the maximum size, then HLA will align the object on a boundary that is a multiple of the maximum size. As an example, consider the following record:

```
type
    r: record[  4:1 ];
        a:byte;             // offset 0
        b:word;             // offset 2
        c:byte;             // offset 4
        d:dword;[ 2]        // offset 8
        e:byte;             // offset 16
        f:byte;             // offset 17
        g:qword;            // offset 20
    endrecord;
```

Note that HLA aligns *g* on a double word boundary (not quad word, which would be offset 24) since the maximum alignment size is four. Note that since the minimum size is one, HLA allows the *f* field to be aligned on an odd boundary (since it s a byte).

If an array, record, or union field appears within a record, then HLA uses the size of an array element or the largest field of the record or union to determine the alignment size. That is, HLA will align the field without the outermost record on a boundary that is compatible with the size of the largest element of the nested array, union, or record.

HLA s sophisticated record alignment facilities let you specify record field alignments that match that used by most major high level language compilers. This lets you easily access data types used in those HLLs without

resorting to inserting lots of *ALIGN* directives inside the record. We ll take a look at this feature in the next chapter when we discuss how to translate C structs into HLA records.

By default, the first field of a record is assigned offset zero within that record. If you would like to specify a different starting offset, you can use the following syntax for a record declaration:

```
type
    Pt3D:
        record := 4;
            x: int32;
            y: int32;
            z: int32;
        endrecord;
```

The constant expression specified after the assignment operator (:=) specifies the starting offset of the first field in the record. In this example *x, y,* and *z* will have the offsets 4, 8, and 12, respectively.

Warning: setting the starting offset in this manner does not add padding bytes to the record. This record is still a 12-byte object. If you declare variables using a record declared in this fashion, you may run into problems because the field offsets do not match the actual offsets in memory. This option is intended primarily for mapping records to pre-existing data structures in memory. Only really advanced assembly language programmers should use this option.

Note: Windows carefully defines the data fields in most of the record data structures so that you don t have to worry about field alignment. As long as you ensure that a record variable is sitting at a double-word aligned address in memory, you can be confident that the record s fields will be aligned at an appropriate address within that record.

2.2.3: Nested and Anonymous Unions and Records

The fields of a record or union may take on any legal data type. Such fields can be primitive types (as has been the case in the examples up to this point), arrays, or even other record and union types. Support for nestable (*recursive*) data types in an assembly language is very important to Win32 assembly programmers because many Win32 data structures employ this scheme.

Consider, for example, the following record type:

```
type
    recWrecAndUnion:
        record
            r1Field: byte;
            r2Field: word;
            r3Field: dword;
            rRecField:
                record
                    rr4Field: byte[ 2] ;
                    rr5Field: word[ 3] ;
                endrecord;
            r6Field: byte;
            rUnionField:
                union
                    r7Field:dword;
                    r8Field:real64;
                endunion;
```

```
        endrecord;
```

To access the nested *record* and *union* fields found within the *recWrecAndUnion* type, you use an extended form of the dot-notation . For example, if you have a variable, *r*, of type *recWrecAndUnion*, you can access the various fields of this variable using the following identifiers:

```
r.r1Field
r.r2Field
r.r3Field
r.rRecField.rr4Field
r.rRecField.rr5Field
r.r6Field
r.rUnionField.r7Field
r.rUnionField.r8Field
```

In a similar fashion, unions may contain records and other unions as fields, e.g.,

```
type
    unionWrecord:
        union
            u1Field: byte;
            u2Field: word;
            u3Field: dword;
            urField:
                record
                    u4Field: byte[ 2] ;
                    u5Field: word[ 3] ;
                endrecord;
            u6Field: byte;
        endunion;
```

Again, you would use the extended dot-notation syntax to access the fields of this *union*. Assuming that you have a variable *u* of type *unionWrecord*, you would access *u's* fields as follows:

```
u.u1Field
u.u2Field
u.u3Field
u.urField.u4Field
u.urField.u5Field
u.u6Field
```

Whenever a *record* appears within a *union*, the total size of that *record* is what HLA uses to determine the largest data object in the *union* for the purposes of determining the *union's* size. Also note that the fields of a *record* within a *union* all begin at separate offsets; it is the *record* object in the *union* that begins at the same offset as the other fields in the *union*. The fields of the *record*, however, begin at unique offsets within that *record*, just as for any *record*.

Unions also support a special field type known as an *anonymous record* . The following example demonstrates the syntax for an anonymous *record* in a *union*:

```
type
    unionWrecord:
        union
```

```
            u1Field: byte;
            u2Field: word;
            u3Field: dword;
            record
                u4Field: byte[ 2];
                u5Field: word[ 3];
            endrecord;
            u6Field: byte;
        endunion;
```

Fields appearing within the anonymous record do not necessarily start at offset zero in the data structure. In the example above, *u4Field* starts at offset zero while u5Field immediately follows it two bytes later. The fields in the *union* outside the anonymous *record* all start at offset zero. If the size of the anonymous record is larger than any other field in the *union*, then the record s size determines the size of the *union*. This is true for the example above, so the *union's* size is 16 bytes since the anonymous record consumes 16 bytes.

To access a field of an anonymous record in a *union*, you use the standard dot-notation syntax. Because the anonymous record does not have an explicit field name associated with the *record*, you access the fields of the *record* using a single-level dot notation, just like the other fields in the *union*. For example, if you have a variable named *uwr* of type *unionWrecord*, you d access the fields of *uwr* as follows:

```
uwr.u1Field
uwr.u2Field
uwr.u3Field
uwr.u4Field -- a field of the anonymous record
uwr.u5Field -- a field of the anonymous record
uwr.u6Field
```

You may also declare anonymous unions within a *record*. An anonymous union is a *union* declaration without a field name associated with the *union*, e.g.,

```
type
    DemoAU:
        record
            x: real32;
            union
                u1:int32;
                r1:real32;
            endunion;
            y:real32;
        endrecord;
```

In this example, *x, u1, r1,* and *y* are all fields of *DemoAU*. To access the fields of a variable *D* of type *DemoAU*, you would use the following names: *D.x, D.u1, D.r1,* and *D.y*. Note that *D.u1* and *D.r1* share the same memory locations at run-time, while *D.x* and *D.y* have unique addresses associated with them.

2.2.4: Pointer Types

HLA allows you to declare a pointer to some other type using syntax like the following:

```
pointer to base_type
```

The following example demonstrates how to create a pointer to a 32-bit integer within the type declaration section:

```
type pi32: pointer to int32;
```

HLA pointers are always 32-bit pointers.

HLA also allows you to define pointers to existing procedures using syntax like the following:

```
procedure someProc( parameter_list );
<< procedure options, followed by @external, @forward, or procedure body>>
      .
      .
      .
type
   p : pointer to procedure someProc;
```

The *p* procedure pointer inherits all the parameters and other procedure options associated with the original procedure. This is really just shorthand for the following:

```
procedure someProc( parameter_list );
<< procedure options, followed by @external, @forward, or procedure body>>
      .
      .
      .
type
   p : procedure ( Same_Parameters_as_someProc ); <<same options as someProc>>
```

The former version, however, is easier to maintain since you don t have to keep the parameter lists and procedure options in sync.

Note that HLA provides the reserved word null (or NULL, reserved words are case insensitive) to represent the nil pointer. HLA replaces NULL with the value zero. The NULL pointer is compatible with any pointer type (including strings, which are pointers).

2.2.5: Thunk Types

A thunk is an eight-byte variable that contains a pointer to a piece of code to execute and an execution environment pointer (i.e., a pointer to an activation record). The code associated with a thunk is, essentially, a small procedure that (generally) uses the activation record of the surround code rather than creating its own activation record. HLA uses thunks to implement the iterator *yield* statement as well as pass by name and pass by lazy evaluation parameters. In addition to these two uses of thunks, HLA allows you to declare your own thunk objects and use them for any purpose you desire. Windows also uses the term thunk but the Windows meaning is different than the HLA meaning. Fortunately, most Windows thunks involve calling 16-bit code in older versions of Windows. In modern Windows systems, you use thunks much less often than in older versions of Windows.

For more information about HLA thunks, please consult the *HLA Reference Manual*. Thunks are mentioned here only because of the possible confusion that might exist with Windows thunks. This book will not use HLA thunks, so there is little need to discuss this subject further.

2.2.6: Type Coercion

One HLA feature that many established assembly language programmers tend to have a problem with is the fact that HLA enforces a fair amount of type checking on its operands. Traditional assemblers generally treat variable names in a program as a memory address and nothing more; it is up to the programmer to choose an instruction that operates on the data type at that address. More modern assemblers, like MASM and TASM, enforce a small amount of type checking by storing the size of a memory object along with its address in the assembler s *symbol table* (a database where the assembler keeps track of information related to symbols during assembly). HLA takes this concept even farther. Although HLA is nowhere near as strongly typed as some high level languages (e.g., Ada), HLA does enforce type checking more strongly than most assemblers. While there are some well-established software engineering benefits associated with strong type checking, assembly programmers often need to access a object in a way that may be incompatible with its type. To achieve this in HLA, you use HLA s *type coercion operator.*

HLA s type coercion operator replaces the type of a memory or register operand. This syntax for this operator is

(type *newtype register_or_memory_operand*)

where *newtype* is a valid HLA type specification and *register_or_memory_operand* is a register name or a memory address. For register operands, the size of the new type must be compatible with the register s size; that is, if the register is an eight-bit register, then the type must be a one-byte type. This restriction does not apply to memory operands[3]. Here are some examples of HLA s type coercion operator:

```
(type byte uns8Var)
(type int32 eax) // Treat eax as a signed integer rather than unsigned (the default)
(type string esi) // esi is string object
(type recType [ ebx] ) // ebx is a pointer to a record of type recType
(type recType [ ebx] ).someField // Access field "someField" pointed at by EBX
(type procWith2Parms edx)(parm1, parm2); //Call procedure pointed at by edx
```

Note that there is a big difference between (type string esi) and (type string [esi]) . The string type is a pointer type, so (type string esi) tells HLA that ESI contains a string variable s value (that is, a pointer to character string data). The construct (type string [esi]) tells HLA that ESI points at a string variable (which is, itself, a pointer to character string data).

The important thing to note about HLA s type coercion operator is that you can treat everything inside the parentheses as though it were are single memory object (or register object) of that particular type. Therefore, you can use the dot-operator to reference fields of records, unions, and class, you can attach a parameter list to procedure pointer types, etc. Note, by the way, that there is a difference between the following two HLA statements:

```
(type procWithTwoParms edx)( parm1, parm2 );
(type procWithTwoParms [ edx] )( parm1, parm2 );
```

The first statement above calls the procedure whose address EDX contains. The second statement calls the procedure whose address is contained in the double word at which EDX points.

3. The restriction on registers exists because it doesn t make sense to cast a register as a size that is different than the register s actual size. Memory operands, on the other hand, may consume fewer or more bytes that the variable s original declaration because the 80x86 CPU can access addresses after the variable without any problems.

HLA's machine types (*byte, word, dword, qword, tbyte*, and *lword*) are generally "type-free". The only type checking HLA does on operands of these types is to ensure that the value you store into one of these objects is the same size as the destination operand. The 80x86's integer registers, for example, have the types byte, word, and dword associated with them, so they are compatible with most (ordinal/scalar) objects that are one, two, or four bytes long. HLA's *real32* type is a special case - HLA requires that you cast *real32* objects to *dword* if you want manipulate them as a 32-bit integer/ordinal object (e.g., in an integer register).

For those who complain about all the extra type involved with using HLA's type coercion operator, note that you can use an HLA TEXT constant or macro to replace some common type coercion with a single identifier, e.g.,

```
const
   recEBX :text := "(type recType [ ebx] )";
      .
      .
      .
   mov( recEBX.field, eax ); // copies (type recType [ ebx] ).field into EAX
```

As a general rule, you want to be careful about using HLA's type coercion operator. The overuse of the coercion operator is a sure sign of bad program design, particularly if you're constantly recasting memory variables (as opposed to constructs like [ebx] that truly require type casting). If you really need to access an object using one of several different types, you might consider using a *union* type rather than the type coercion operator. Using a *union* type is a little bit more structured (and less typing if you pick an appropriate union name) and helps document your intent a little better.

2.3: HLA High-Level Control Structures

An assumption that this book makes is that you are reasonably familiar with HLA and already a competent assembly/HLA programmer. Because HLA's high level control structures were originally intended as a device to help beginners learn assembly language programming via HLA, you might question why a chapter on advanced HLA usage would consider these statements. Indeed, because it's probably safe to assume that Win32 assembly programming is for advanced assembly language programmers, you might argue that a discussion of HLA's high-level control structures doesn't even belong in this book - advanced assembly language programmers already know the low-level way to do control structures.

A typical assembly programmer who has learned assembly language programming via HLA typically goes through three phases in their assembly education. In phase one (beginner), the programmer uses HLA's high-level control structures as a "crutch" to leverage their existing high level language programming knowledge in order to quickly learn assembly language. In phase two, the beginning assembly language programmer discovers that these statements aren't "true" assembly language and they learn the "proper" way to write control flow in assembly language using comparisons, tests, and conditional branches. Once an HLA programmer masters the low-level control flow and gains some experience using assembly language, that programmer enters a third phase where they gain a complete understanding of how HLA converts those high-level control structures into machine instructions and the programmer realizes that sometimes it's beneficial to go ahead and use those high level control statements in order to make their programs more readable.

The vast majority of Win32 assembly code written today is written with a high level assembler and the common convention is to prefer high level control structures over the low-level "compare and branch" sequences that low-level assemblers provide. That is to say, most Win32 assembly programmers have entered that "third phase" of their assembly programming careers where they begin to make intelligent choices concerning the use of high level control structures in order to make their programs more readable.

Windows assembly language programming is sufficiently complex that any tool we can employ to reduce the effort needed to write applications in assembly, that doesn t destroy the efficiency of the resulting application, is well worth using. Tools like HLAs high level control structures (and high level procedure definitions that we ll look at in the next section) provide a tremendous boost to assembly language productivity. So their use is worth serious consideration by Windows assembly programmers.

Long-time die-hard assembly programmers may find the use of anything high-level troublesome, even disgusting, in code that claims to be assembly language. There are two perceived problems with using high-level-like control structures in assembly code: inefficient compilation and inefficient paradigm.

The inefficient compilation issue concerns the fact that few assemblers (if any) provide statement-level optimization facilities such as those found in high-level language compilers (e.g., C). This means that high level compilers are theoretically capable of producing better code than a high level assembler because the high level language compiler will attempt to optimize the code it produces whereas no high level assembler does that (today, anyway). This isn t quite as bad as it seems, though, because 80x86 high level assemblers (including HLA) generally place sufficient restrictions on their high level control statements that enable them to generate fairly decent code. Still, do keep in mind that high level assemblers (and HLA in particular) don t optimize the code generated for high level control structures.

Of the two problems, the inefficient paradigm problem is probably the bigger problem of the two. Programmers who write assembly code using high level control structures often think in high level terms rather than in assembly terms (i.e., they write C code using mov statements). While this is a very real problem, this book will make the tacit assumption that a good assembly language programmer always considers the code a high-level control structure will generate and will avoid the use of such code if it leads to something that is overly inefficient.

The purpose of this section is to describe how HLA generates machine code for various high level control structures so that an assembly language programmer can make an educated decision concerning their use. While it s easy enough to get lazy and use a high level control structure where it s inappropriate, this book assumes that someone who cares about efficiency will not succumb to the temptation to overuse these control structures.

One other fact is worth pointing out concerning program efficiency and the need to optimize an assembly language sequence to squeeze out every last cycle - most Win32 programs spend the majority of their time waiting for program events. It is perfectly possible for a Win32 assembly programmer to make the application s code run ten times faster yet the user of that software won t perceive any difference in performance. This can happen because the program spends 95% of its time in the Windows kernel (or some other spot outside of the application) and all you achieve by speeding up the application code by a factor of ten is to shift 4.5% of the time that the application used to consume into Windows. The user will not notice a practical difference in the execution time of such a program improvement[4]. This is not to suggest that improved program efficiency isn t something to strive for; it simply means that a good software engineer carefully considers the need for optimization and weighs whether the benefits of such optimization are worth the cost (e.g., the production of less readable code by using low-level control structures in assembly language).

2.3.1: Boolean Control Expressions

A big reason why HLA generates reasonably good machine code for its high level control structures is the fact that HLA places several restrictions on the boolean expression you can use with these statements. In this section we ll review HLAs boolean expressions and then discuss how to write the most efficient forms of these expressions.

4. Obviously, this does not apply in every case, some applications are compute bound and consume an inordinate amount of CPU time within the application. Most Win32 applications, however, are not compute bound.

The primary limitation of HLA's *if* and other HLL statements has to do with the conditional expressions allowed in these statements. These expressions must take one of the following forms:

```
operand1 relop operand2

register in constant .. constant
register not in constant .. constant

memory in constant .. constant
memory not in constant .. constant

reg8 in CSet_Constant
reg8 in CSet_Variable

reg8 not in CSet_Constant
reg8 not in CSet_Variable

register
!register

memory
!memory

Flag

( boolean_expression )
!( boolean_expression )

boolean_expression && boolean_expression

boolean_expression || boolean_expression
```

For the first form, operand1 *relop* operand2 , *relop* is one of:

```
=  or ==       (either one, both are equivalent)
<> or !=       (either one)
<
<=
>
>=
```

Operand1 and *operand2* must be operands that would be legal for a *cmp(operand1, operand2);* instruction. In fact, HLA simply translates these expressions into a *cmp* instruction a a conditional jump instruction (the exact conditional jump instruction is chosen by the operator and the type of operands). For the *if* statement, HLA emits a *cmp* instruction with the two operands specified and an appropriate conditional jump instruction that skips over the statements following the *then* reserved word if the condition is false. For example, consider the following code:

```
if( al = 'a' ) then

   stdout.put( "Option 'a' was selected", nl );

endif;
```

HLA translates this to:

```
cmp( al, 'a' );
jne skip_if;

   stdout.put( "Option 'a' was selected", nl );

skip_if:
```

HLA efficiently translates expressions of this form into machine code. As any decent assembly programmer knows, you ll get more efficient code if one (or both) of the operands are an 80x86 register. As the cmp instruction does not allow two memory operands, you may not compare one memory operand to another using the relational expression.

Unlike the conditional branch instructions, the six relational operators cannot differentiate between signed and unsigned comparisons (for example, HLA uses < for both signed and unsigned less than comparisons). Since HLA must emit different instructions for signed and unsigned comparisons, and the relational operators do not differentiate between the two, HLA must rely upon the types of the operands to determine which conditional jump instruction to emit.

By default, HLA emits unsigned conditional jump instructions (i.e., *ja, jae, jb, jbe,* etc.). If either (or both) operands are signed values, HLA will emit signed conditional jump instructions (i.e., *jg, jge, jl, jle*, etc.) instead. For example, suppose *sint32* is a signed integer variable and *uint32* is an unsigned 32-bit integer, consider the code that HLA will generate for the following expressions:

```
// sint32 < 5

   cmp( sint32, 5 );
   jnl skip;
      .
      .
      .
// uint32 < 5

   cmp( uint32, 5 );
   jnb skipu;
```

HLA considers the 80x86 registers to be *unsigned*. This can create some problems when using the HLA *if* statement. Consider the following code:

```
if( eax < 0 ) then

   << do something if eax is negative >>

endif;
```

Since neither operand is a signed value, HLA will emit the following code:

```
      cmp( eax, 0 );
      jnb SkipThenPart;
      << do something if eax is negative >>
SkipThenPart:
```

Unfortunately, it is never the case that the value in EAX is below zero (since zero is the minimum unsigned value), so the body of this *if* statement never executes. Clearly, the programmer intended to use a signed comparison here. The solution is to ensure that at least one operand is signed. However, as this example demonstrates, what happens when both operands are intrinsically unsigned?

The solution is to use coercion to tell HLA that one of the operands is a signed value. In general, it is always possible to coerce a register so that HLA treats it as a signed, rather than unsigned, value. The *if* statement above could be rewritten (correctly) as

```
if( (type int32 eax) < 0 ) then

   << do something if eax is negative >>

endif;
```

HLA will emit the *jnl* instruction (rather than *jnb*) in this example. Note that if either operand is signed, HLA will emit a signed condition jump instruction. Therefore, it is not necessary to coerce both unsigned operands in this example.

The second form of a conditional expression that the *if* statement accepts is a register or memory operand followed by in and then two constants separated by the .. operator, e.g.,

```
   if( al in 1..10 ) then ...
```

This code checks to see if the first operand is in the range specified by the two constants. The constant value to the left of the .. must be less than the constant to the right for this expression to make any sense. The result is true if the operand is within the specified range. For this instruction, HLA emits a pair of compare and conditional jump instructions to test the operand to see if it is in the specified range, e.g.,

```
   cmp( al, 1 );
   jb notInRange;
   cmp( al, 10 );
   ja notInRange;
```

Once again, HLA uses a signed or unsigned comparison based on the type of the memory or register operand present.

HLA also allows a exclusive range test specified by an expression of the form:

```
   if( al not in 1..10 ) then ...
```

In this case, the expression is true if the value in AL is outside the range 1..10. HLA generates code like the following for this expression:

```
   cmp( al, 1 );
   jb IsInRange;
   cmp( al, 10 );
   jb notInRange;
IsInRange:
```

In addition to integer ranges, HLA also lets you use the *in* operator with character set constants and variables. The generic form is one of the following:

```
reg8 in CSetConst
reg8 not in CSetConst
reg8 in CSetVariable
reg8 not in CSetVariable
```

For example, a statement of the form *if(al in {'a'..'z'}) then ...* checks to see if the character in the AL register is a lower case alphabetic character. Similarly,

```
if( al not in {'a'..'z', 'A'..'Z'}) then...
```

checks to see if AL is not an alphabetic character. This form generates some particularly ugly code, so you want to be careful about using it. The code it generates looks something like the following:

```
push( eax );
movzx( al, eax );
bt( eax, compilerGeneratedCset );
pop( eax );
jnc notInSet;
```

On top of this code, HLA also generates a 16-bit character set constant object containing the members specified by the character set constant in the boolean expression.

The fifth form of a conditional expression that the *if* statement accepts is a single register name (eight, sixteen, or thirty-two bits). The *if* statement will test the specified register to see if it is zero (false) or non-zero (true) and branches accordingly. If you specify the not operator (!) before the register, HLA reverses the sense of this test.

The sixth form of a conditional expression that the *if* statement accepts is a single memory location. The type of the memory location must be boolean, byte, word, or dword. HLA will emit code that compares the specified memory location against zero (false) and generate an appropriate branch depending upon the value in the memory location. If you put the not operator (!) before the variable, HLA reverses the sense of the test.

The seventh form of a conditional expression that the *if* statement accepts is a Flags register bit or other condition code combination handled by the 80x86 conditional jump instructions. The following reserved words are acceptable as *if* statement expressions:

```
@c, @nc, @o, @no, @z, @nz, @s, @ns, @a, @na, @ae, @nae, @b, @nb, @be,
@nbe, @l, @nl, @g, @ne, @le, @nle, @ge, @nge, @e, @ne
```

These items emit an appropriate jump (of the opposite sense) around the *then* portion of the *if* statement if the condition is false.

If you supply any legal boolean expression in parentheses, HLA simply uses the value of the internal expression for the value of the whole expression. This allows you to override default precedence for the &&, ||, and ! operators.

The !(boolean_expression) evaluates the expression and does just the opposite. That is, if the interior expression is false, then !(boolean_expression) is true and vice versa. This is mainly useful with conjunction and disjunction since all of the other interesting terms already provide a logically negated form. Note that in general, the ! operator must precede some parentheses. You cannot say ! AX < BX , for example.

Originally, HLA did not include support for the conjunction (&&) and disjunction (||) operators. This was explicitly left out of the design so that beginning students would be forced to rethink their logical operations in assembly language. Unfortunately, it was so inconvenient not to have these operators that they were eventually added.

The conjunction and disjunction operators are the operators && and ||. They expect two valid HLA boolean expressions around the operator, e.g.,

```
eax < 5 && ebx <> ecx
```

Since the above forms a valid boolean expression, it, too, may appear on either side of the && or || operator, e.g.,

```
eax < 5 && ebx <> ecx || !dl
```

HLA gives && higher precedence than ||. Both operators are left-associative so if multiple operators appear within the same expression, they are evaluated from left to right if the operators have the same precedence. Note that you can use parentheses to override HLA s default precedence.

When generating code for these expressions, HLA employs *short-circuit evaluation*. Short-circuit evaluation means that HLA can generate code that uses control flow to maintain the current state (true or false) of the boolean expression rather than having to save the result somewhere (e.g., in a register). Because HLA promises not to disturb register values in its HLL-like code, short-circuit evaluation is the best way to proceed (it won t require pushing and popping registers as was necessary in the character set test earlier). For the current example, HLA would probably generate code like the following:

```
// eax < 5 && ebx <> ecx || !dl

   cmp( eax, 5 );
   jnb tryDL;
   cmp( ebx, ecx );
   jne isTrue;
tryDL:
   test( dl, dl );
   jnz isFalse;
isTrue:
```

One wrinkle with the addition of && and || is that you need to be careful when using the flags in a boolean expression. For example, eax < ecx && @nz hides the fact that HLA emits a compare instruction that affects the Z flag. Hence, the @nz adds nothing to this expression since EAX must not equal ECX if eax<ecx. So take care when using && and ||.

If you would prefer to use a less abstract scheme to evaluate boolean expressions, one that lets you see the low-level machine instructions, HLA provides a solution that allows you to write code to evaluate complex boolean expressions within the HLL statements using low-level instructions. Consider the following syntax:

```
if
(#{
   <<arbitrary HLA statements >>
}#) then

   << "True" section >>
```

```
else //or elseif...

   << "False" section >>

endif;
```

The #{ and }# brackets tell HLA that an arbitrary set of HLA statements will appear between the braces. HLA will *not* emit any code for the *if* expression. Instead, it is the programmer s responsibility to provide the appropriate test code within the #{---}# section. Within the sequence, HLA allows the use of the boolean constants *true* and *false* as targets of conditional jump instructions. Jumping to the *true* label transfers control to the true section (i.e., the code after the *then* reserved word). Jumping to the *false* label transfers control to the false section. Consider the following code that checks to see if the character in AL is in the range a .. z :

```
   if
   (#{
      cmp( al, 'a' );
      jb false;
      cmp( al, 'z' );
      ja false;
   }#) then

      << code to execute if AL in {'a'..'z'} goes here >>

   endif;
```

With the inclusion of the #{---}# operand, the *if* statement becomes much more powerful, allowing you to test any condition possible in assembly language. Of course, the #{---}# expression is legal in the *elseif* expression as well as the *if* expression.

2.3.2: The HLA IF..ENDIF Statement

HLA provides a limited *if..then..elseif..else..endif* statement that can help make your programs easier to read. For the most part, HLA s *if* statement provides a convenient substitute for a *cmp* and a conditional branch instruction pair (or chain of such instructions when employing *elseif's*).

The generic syntax for the HLA if statement is the following:

```
if( conditional_expression ) then

   << Statements to execute if expression is true >>

endif;


if( conditional_expression ) then

   << Statements to execute if expression is true >>

else

   << Statements to execute if expression is false >>
```

```
endif;

if( expr1 ) then

   << Statements to execute if expr1 is true >>

elseif( expr2 ) then

   << Statements to execute if expr1 is false
       and expr2 is true >>

endif;


if( expr1 ) then

   << Statements to execute if expr1 is true >>

elseif( expr2 ) then

   << Statements to execute if expr1 is false
       and expr2 is true >>

else

   << Statements to execute if both expr1 and
       expr2 are false >>

endif;
```

Note: HLAs if statement allows multiple *elseif* clauses. All *elseif* clauses must appear between *if* clause and the *else* clause (if present) or the *endif* (if an *else* clause is not present).

For simple boolean expressions (see the previous section) HLA generally emits code that is comparable to what an expert assembly language programmer would generate for these statements. There are, however, two issues to keep in mind when using HLAs high-level *if..elseif..else..endif* control structures: the cost of branching and inefficiencies associated with nested *if* statements.

High level control structures abstract away the low-level machine code implementation. Most of the time, this is a good thing. The whole purpose of the high level control structures is to hide what s going on underneath and make the code easier to read and understand. Sometimes, however, this abstraction can hide inefficiencies. Consider the following HLA *if* statement and its low-level implementation:

```
   if( eax < ebx ) then

      << do this if eax < ebx >>

   else

      << do this if eax >= ebx >>

   endif;

// Low-level equivalent:

   cmp( eax, ebx );
```

```
   jnb eaxGEebx;

      << do this if eax < ebx >>

      jmp ifIsDone;

eaxGEebx:
      << do this if eax >= ebx >>

ifIsDone:
```

This looks relatively straight-forward and simple. Probably the code that most people would have written if HLA s high-level *if* statement was not available. By carefully studying this code, you will note that there are two execution paths in this code, one path taken if the expression evaluates true and one path taken if the expression evaluates false. There is a transfer of control instruction that is taken along either path (the *jmp* instruction along the true path and the *jnb* instruction along the false path). Because jump instructions can execute slowly on modern CPU architectures, this particular code sequence may run slower than necessary because a jump is always taken.

An expert assembly programmer might consider the frequency with which the boolean expression in this *if* statement evaluates true or false. If the result of this boolean expression is random (that is, there is a 50/50 chance it will be true or false) then there is little you can do to improve the performance. However, in most if statements like this one, it s usually the case that one value occurs more frequently than the other (often, a boolean expression tends to evaluate true more often than false because most programmers tend to put the most common case in the true section of an *if..else..endif* statement). If that is the case, and the *if* statement is sitting in some section of time-critical code, an expert assembly programmer will often convert this *if* statement to something like the following:

```
   cmp( eax, ebx );
   jnb eaxGEebx;
      << do this if eax < ebx >>
returnHere:
      .
      .
      .
// Somewhere convenient:

eaxGEebx:
   << do this if eax >=ebx >>
   jmp returnHere;
```

By moving the *else* section to some other point in the code (presumably, after a control transfer instruction) this code provides a straight-through path (i.e., no *jmp*) for the true section of the code. The *else* section, on the other hand, now has to execute two jump instructions (the condition jump and the *jmp* at the end of the else sequence). However, if the *else* section executes less frequently than the *true* section of the *if* statement, this code sequence will run a little faster on many CPUs. One problem with the high-level *if..else..endif* statement is that it obscures the presence of that *jmp* instruction that transfers control over the *else* section. So you need to remember this fact if you re writing some highly time-critical code. Most of the time, this type of change (spaghetti programming) will not make a noticeable difference in the execution time of your program. Under the right circumstances, however, it will make a difference.

In general, any single instance of an inefficiency like these two is going to have a negligible impact on the performance or size of your programs. However, keep in mind that if you write assembly code using a high level

language mindset, inefficiencies like this will begin to creep in all over the place and may begin to have an impact on the execution speed of your applications. Therefore, you should carefully consider each high level control statement you use in HLA to verify that HLA will be able to generate reasonable code for the sequence you re supplying.

2.3.3: The HLA WHILE..ENDWHILE Statement

The *while..endwhile* statement allows the following syntax:

```
while( boolean_expression ) do

   << while loop body>>

endwhile;


while( boolean_expression ) do

   << while loop body>>

welse

   << Code to execute when boolean_expression is false >>

endwhile;


while(#{ HLA_statements }#) do

   << while loop body>>

endwhile;

while(#{ HLA_statements }#) do

   << while loop body>>

welse

   << Code to execute when expression is false >>

endwhile;
```

The *while* statement allows the same boolean expressions as the HLA *if* statement. Like the HLA *if* statement, HLA allows you to use the boolean constants *true* and *false* as labels in the #{...}# form of the *while* statement above. Jumping to the *true* label executes the body of the *while* loop, jumping to the *false* label exits the while loop.

For the *while(expr) do* forms, HLA moves the test for loop termination to the bottom of the loop and emits a jump at the top of the loop to transfer control to the termination test. For example, consider the following *while* statement:

```
while( eax < ebx ) do
```

```
      << statements to execute while eax < ebx >>

   endwhile;
```

Typical hand conversion of this *while* loop:

```
whlLabel:
      cmp( eax, ebx );
      jnb endWhileLabel;

         << statements to execute while eax < ebx >>

      jmp whlLabel;
endWhileLabel:
```

Here s the code that HLA actually emits for this *while* loop:

```
      jmp doWhile;
whlLabel:
         << statements to execute while eax < ebx >>

doWhile:
      cmp( eax, ebx );
      jb whlLabel;
```

The advantage of rotating the test to the bottom of the loop is that it eliminates a jump out of the body s path. That is, each iteration of this *while* loop executes one less *jmp* instruction than the typical conversion. As long as the loop s body executes one or more times (on the average), this conversion scheme is slightly more efficient. Note, however, that if the loop body doesn t execute the majority of the time, then the former conversion runs slightly faster.

For the *while(#{stmts}#)* form, HLA compiles the termination test at the top of the emitted code for the loop (i.e., the standard conversion). Therefore, the standard *while* loop may be slightly more efficient (in the typical case) than the hybrid form; just something to keep in mind.

The HLA *while* loop supports an optional *welse* (while-else) section. The *while* loop will execute the code in this section only when then the expression evaluates false. Note that if you exit the loop via a *break* or *breakif* statement the *welse* section does not execute. This provides logic that is sometimes useful when you want to do something different depending upon whether you exit the loop via the expression going false or by a *break* statement. Here s a quick example that demonstrates how HLA translates a *while..welse..endwhile* statement into low-level code:

```
   while( eax < ebx ) do

      << statements to execute while eax < ebx >>
      breakif( ecx = 0 );
      << more statements to execute >>

   welse

      << statements to execute when the boolean expression evaluates false >>

   endwhile;
```

```
// Typical HLA conversion to low-level code:

      jmp whlExpr;
whlLoop:
         << statements to execute while eax < ebx >>

         // breakif conversion:

         test( ecx, ecx );
         jz whileDone;

         << more statements to execute >>

whlExpr:
      cmp( eax, ebx );
      jb whlLoop;

// code for the welse section:

      << statements to execute when the boolean expression evaluates false >>

whileDone:
```

2.3.4: The HLA REPEAT..UNTIL Statement

HLAs *repeat..until* statement uses the following syntax:

```
repeat

   << statements to execute repeatedly >>

until( boolean_expression );


repeat

   << statements to execute repeatedly >>

until(#{  HLA_statements } #);
```

The body of the loop always executes at least once and the test for loop termination occurs at the bottom of the loop. The *repeat..until* loop (unlike C/C++ s *do..while* statement) terminates loop execution when the expression is true (that is, *repeat..until* repeats while the expression is false).

As you can see, the syntax for this is very similar to the *while* loop. About the only major difference is the fact that jump to the *true* label in the #{---}# sequence exits the loop while jumping to the *false* label in the #{---}# sequence transfers control back to the top of the loop. Also note that there is no equivalent to the *welse* clause in a *repeat..until* loop.

If you take away the jmp that HLA emits at the top of a (standard) *while* loop, you ll have a pretty good idea of the type of code that HLA generates for a *repeat..until* loop.

As a general rule, *repeat..until* loops are slightly more efficient than *while* loops because they don t execute that extra *jmp* instruction. Otherwise, the performance characteristics are virtually identical to that for the *while* loop.

2.3.5: The HLA FOR Loops

The HLA *for..endfor* statement is very similar to the C/C++ *for* loop. The *for* clause consists of three components:

```
for( initialize_stmt; boolean_expression; increment_statement ) do
```

The *initialize_statement* component is a single machine instruction. This instruction typically initializes a loop control variable. HLA emits this statement before the loop body so that it executes only once, before the test for loop termination.

The *boolean_expression* component is a simple boolean expression. This expression determines whether the loop body executes. Note that the *for* statement tests for loop termination before executing the body of the loop, just like the *while* statement (though the compiler will move this test to the bottom of the loop for efficiency reasons, the semantics are the same as though the test were physically at the beginning of the loop).

The *increment_statement* component is a single machine instruction that HLA emits at the bottom of the loop, just before jumping back to the top of the loop. This instruction is typically used to modify the loop control variable.

The syntax for the HLA *for* statement is the following:

```
for( initStmt; BoolExpr; incStmt ) do

   << loop body >>

endfor;

-or-

for( initStmt; BoolExpr; incStmt ) do

   << loop body >>

felse

   << statements to execute when BoolExpr evaluates false >>

endfor;
```

Semantically, this statement is identical to the following *while* loop:

```
initStmt;
while( BoolExpr ) do
   << loop body >>
   incStmt;
endwhile;

-or-
```

```
initStmt;
while( BoolExpr ) do
   << loop body >>
   incStmt;

welse

   << statements to execute when BoolExpr evaluates false >>
endwhile;
```

Note that HLA does not include a form of the *for* loop that lets you bury a sequence of statements inside the boolean expression. Use the *while* loop if you want to do that. If this is inconvenient, you can always create your own version of the *for* loop using HLA's macro facilities.

The *felse* section in the *for..felse..endfor* loop executes when the boolean expression evaluates false. Note that the *felse* section does not execute if you break out of the *for* loop with a *break* or *breakif* statement. You can use this fact to do different logic depending on whether the code exits the loop via the boolean expression going false or via some sort of *break*.

The code that HLA generates for the *for* loop is identical to the code that HLA generates for the converted *while* example. Therefore, efficiency concerns about the *for* loop are identical to those for the *while* loop.

HLA's *forever..endfor* loop creates an infinite loop[5]. The *endfor* efficiently compiles into a single *jmp* instruction that transfers control to the top of the loop (i.e., to the *forever* clause). Typically, you'd use a *break* or *breakif* statement to exit a *forever..endfor* at some point in the middle of the loop. This makes *forever..endfor* loops slightly less efficient that a standard *while..endwhile* or *repeat..until* loop because you'll always execute two jump instructions in such a loop - one that breaks out of the loop and one at the bottom of the loop that transfers control back to the top of the loop. For example, consider the following:

```
   forever
      << Code to execute before the test >>
      breakif( eax = ecx );
      << Code to execute after the test >>
   endfor;

// conversion to "pure" assembly language

   forLabel:
      << Code to execute before the test >>
      cmp( eax, ecx );
      jne exitFor;
      << Code to execute after the test >>
   jmp forLabel;
   exitFor:
```

An experienced assembly language programmer would probably rewrite this as a loop that tests for loop termination at the bottom of the loop using code like the following:

```
// conversion to slightly more efficient assembly language

   jmp IntoLoop
```

5. Though you may use HLA's *break, breakif, exit,* or *exitif* statements to escape from such a loop.

```
   forLabel:
      << Code to execute after the test >>
   IntoLoop:
      << Code to execute before the test >>
      cmp( eax, ecx );
      je forLabel;
```

HLA provides a third *for* loop, the *foreach* loop, that lets you create user-defined loops via HLAs *iterator* facility. We re not going to discuss the code generation for the *foreach* loop here, but be aware that HLA generates some fairly sophisticated code for the *foreach* loop and the corresponding *iterator*. Though the code is not particularly inefficient, you should note that using a *foreach* loop where a simple *for* or *while* loop will suffice is always going to be less efficient. Please see the HLA documentation for more details concerning the use of iterators and the *foreach* statement.

2.3.6: HLA's BREAK, CONTINUE, and EXIT Statements

The *break* and *breakif* statements allow you to exit a loop at some point other than the normal test for loop termination. These two statements allow the following syntax:

```
break;
breakif( boolean_expression );
breakif(#{ stmts }#);
```

The *continue* and *continueif* statements allow you to restart a loop. These two statements allow the following syntax:

```
continue;
continueif( boolean_expression );
continueif(#{ stmts }#);
```

Restarting a loop means jumping to the loop termination test for *while, repeat..until*, and *for* loops, it means jumping to the (physical) top of the loop for *forever* and *foreach* loops.

Note that the *break*, *breakif, continue,* and *continueif* statements are legal only inside *while, for, forever, foreach*, and *repeat* loops. HLA does not recognize loops you ve coded yourself using discrete assembly language instructions (of course, you can probably write a macro to provide a *break* function for your own loops). Note that the *foreach* loop pushes data on the stack that the *break* statement is unaware of. Therefore, if you break out of a *foreach* loop, garbage will be left on the stack. The HLA *break* statement will issue a warning if this occurs. It is your responsibility to clean up the stack upon exiting a *foreach* loop if you break out of it. Using a continue within a *foreach* loop is perfectly fine.

The *begin..end* statement block provides a structured goto statement for HLA. The *begin* and *end* clauses surround a group of statements; the *exit* and *exitif* statements allow you to exit such a block of statements in much the same way that the *break* and *breakif* statements allow you to exit a loop. Unlike *break* and *breakif*, which can only exit the loop that immediately contains the *break* or *breakif*, the exit statements allow you to specify a *begin* label so you can exit several nested contexts at once. The syntax for the *begin..end, exit*, and *exitif* statements is as follows:

```
begin contextLabel ;

   << statements within the specified context >>
```

```
end contextLabel;

exit contextLabel;
exitif( boolean_expression ) contextLabel;
exitif(#{ stmts }#) contextLabel;
```

The *begin..end* clauses do not generate any machine code (although *end* does emit a label to the assembly output file). The *exit* statement simply emits a *jmp* to the first instruction following the *end* clause. The *exitif* statement emits a compare and a conditional jump to the statement following the specified end.

If you break out of a *foreach* loop using the *exit* or *exitif* statements, there will be garbage left on the stack. It is your responsibility to be aware of this situation (i.e., HLA doesn t warn you about it) and clean up the stack, if necessary.

You can nest *begin..end* blocks and *exit* out of any enclosing *begin..end* block at any time. The *begin* label provides this capability. Consider the following example:

```
program ContextDemo;

#include( "stdio.hhf" );

static
   i:int32;

begin ContextDemo;

   stdout.put( "Enter an integer:" );
   stdin.get( i );

   begin c1;

      begin c2;

         stdout.put( "Inside c2" nl );
         exitif( i < 0 ) c1;

      end c2;
      stdout.put( "Inside c1" nl );
      exitif( i = 0 ) c1;
      stdout.put( "Still inside c1" nl );

   end c1;
   stdout.put( "Outside of c1" nl );

end ContextDemo;
```

The *exit* and *exitif* statements let you exit any *begin..end* block; including those associated with a program unit such as a procedure, iterator, method, or even the main program. Consider the following (unusable) program:

```
      program mainPgm;

         procedure LexLevel1;
```

```
        procedure LexLevel2;
        begin LexLevel2;

            exit LexLevel2;   // Returns from this procedure.
            exit LexLevel1;   // Returns from this procedure and
                              //  and the LexLevel1 procedure
                              //  (including cleaning up the stack).
            exit mainPgm;     // Terminates the main program.

        end LexLevel2;

    begin LexLevel1;
        .
        .
        .
    end LexLevel1;

begin mainPgm;
    .
    .
    .
end mainPgm;
```

Note: You may only exit from procedures that have a display and all nested procedures from the procedure you wish to exit from through to the *exit* statement itself must have displays. In the example above, both *LexLevel1* and *LexLevel2* must have displays if you wish to exit from the *LexLevel1* procedure from inside *LexLevel2*. By default, HLA emits code to build the display unless you use the *"@nodisplay"* procedure option.

Note that to exit from the current procedure, you must not have specified the *"@noframe"* procedure option. This applies only to the current procedure. You may exit from nesting (lower lex level) procedures as long as the display has been built. For more information on displays and stack frames, see the upcoming section on procedures and procedure invocations.

2.3.7: Exception Handling Statements in HLA

HLA's exception handling system is an interesting example for those seeking to write efficient code. While using HLA's try..endtry and raise statements is quite a bit more efficient than manually checking for problems in your code, it's also the case that these statements generate a fair number of instructions and their overuse can lead to the creation of some inefficient code. Therefore, it's important to understand the code that HLA generates for these statements so you can write code efficiently by using these statements properly.

HLA uses the *try..exception..endtry* and *raise* statements to implement exception handling. The syntax for these statements is as follows:

```
try
   << HLA Statements to execute >>

<<  unprotected // Optional unprotected section.
   << HLA Statements to execute >>
>>

exception( const1 )

   << Statements to execute if exception const1 is raised >>
```

```
<< optional exception statements for other exceptions >>

<<  anyexception //Optional anyexception section.
   << HLA Statements to execute >>
>>

endtry;


raise( const2 );
```

Const1 and *const2* must be unsigned integer constants. Usually, these are values defined in the *excepts.hhf* header file. Some examples of predefined values include the following:

```
ex.StringOverflow
ex.StringIndexError

ex.ValueOutOfRange
ex.IllegalChar
ex.ConversionError

ex.BadFileHandle
ex.FileOpenFailure
ex.FileCloseError
ex.FileWriteError
ex.FileReadError
ex.DiskFullError
ex.EndOfFile

ex.MemoryAllocationFailure

ex.AttemptToDerefNULL

ex.WidthTooBig
ex.TooManyCmdLnParms

ex.ArrayShapeViolation
ex.ArrayBounds

ex.InvalidDate
ex.InvalidDateFormat
ex.TimeOverflow
ex.AssertionFailed
ex.ExecutedAbstract
```

Windows Structured Exception Handler exception values:

```
ex.AccessViolation
ex.Breakpoint
ex.SingleStep

ex.PrivInstr
ex.IllegalInstr
```

```
ex.BoundInstr
ex.IntoInstr

ex.DivideError

ex.fDenormal
ex.fDivByZero
ex.fInexactResult
ex.fInvalidOperation
ex.fOverflow
ex.fStackCheck
ex.fUnderflow

ex.InvalidHandle
ex.StackOverflow

ex.ControlC
```

This list is constantly changing as the HLA Standard Library grows, so it is impossible to provide a compete list of standard exceptions here. Please see the *excepts.hhf* header file for a complete list of standard exceptions.

The HLA Standard Library currently reserves exception numbers zero through 1023 for its own internal use. User-defined exceptions should use an integer value greater than or equal to 1024 and less than or equal to 65535 ($FFFF). Exception value $10000 and above are reserved for use by Windows Structured Exception Handler and Linux signals.

The *`try..endtry`* statement contains two or more blocks of statements. The statements to protect immediately follow the *`try`* reserved word. During the execution of the protected statements, if the program encounters the first exception block, control immediately transfers to the first statement following the endtry reserved word. The program will skip all the statements in the exception blocks.

If an exception occurs during the execution of the protected block, control is immediate transferred to an exception handling block that begins with the exception reserved word and the constant that specifies the type of exception.

Example:

```
repeat

   mov( false, GoodInput );
   try
      stdout.put( "Enter an integer value:" );
      stdin.get( i );
      mov( true, GoodInput );

   exception( ex.ValueOutOfRange )

      stdout.put( "Numeric overflow, please reenter ", nl );

   exception( ex.ConversionError )

      stdout.put( "Conversion error, please reenter", nl );

   endtry;

until( GoodInput = true );
```

In this code, the program will repeatedly request the input of an integer value as long as the user enters a value that is out of range (+/- 2 billion) or as long as the user enters a value containing illegal characters.

Multiple *try..endtry* statements can be *nested*. If an exception occurs within a nested *try* protected block, the *exception* blocks in the innermost try block containing the offending statement get first shot at the exceptions. If none of the *exception* blocks in the enclosing *try..endtry* statement handle the specified exception, then the next innermost *try..endtry* block gets a crack at the exception. This process continues until some exception block handles the exception or there are no more *try..endtry* statements.

If an exception goes unhandled, the HLA run-time system will handle it by printing an appropriate error message and aborting the program. Generally, this consists of printing Unhandled Exception (or a similar message) and stopping the program. If you include the excepts.hhf header file in your main program, then HLA will automatically link in a somewhat better default exception handler that will print the number (and name, if known) of the exception before stopping the program.

Note that *try..endtry* blocks are dynamically nested, not statically nested. That is, a program must actually execute the *try* in order to activate the exception handler. You should never jump into the middle of a protected block, skipping over the *try*. Doing so may produce unpredictable results.

You should not use the *try..endtry* statement as a general control structure. For example, it will probably occur to someone that one could easily create a switch/case selection statement using *try..endtry* as follows:

```
try
   raise( SomeValue );

   exception( case1_const)
      <code for case 1>

   exception( case2_const)
      <code for case 2>

   etc.
endtry
```

While this might work in some situations, there are two problems with this code.

First, if an exception occurs while using the *try..endtry* statement as a switch statement, the results may be unpredictable. Second, HLAs run-time system assumes that exceptions are rare events. Therefore, the code generated for the exception handlers doesn t have to be efficient. You will get much better results implementing a switch/case statement using a table lookup and indirect jump (see the Art of Assembly) rather than a *try..endtry* block.

Warning: As you ll see in a moment, the *try* statement pushes data onto the stack upon initial entry and pops data off the stack upon leaving the *try..endtry* block. Therefore, jumping into or out of a *try..endtry* block is an absolute no-no. As explained so far, then, there are only two reasonable ways to exit a *try* statement, by falling off the end of the protected block or by an exception (handled by the *try* statement or a surrounding *try* statement).

The *unprotected* clause in the *try..endtry* statement provides a safe way to exit a *try..endtry* block without raising an exception or executing all the statements in the protected portion of the *try..endtry* statement. An unprotected section is a sequence of statements, between the protected block and the first exception handler, that begins with the keyword *unprotected*. E.g.,

```
try
```

```
   << Protected HLA Statements >>

  unprotected

   << Unprotected HLA Statements >>

  exception( SomeExceptionID )

   << etc. >>

endtry;
```

control flows from the protected block directly into the unprotected block as though the *unprotected* keyword were not present. However, between the two blocks HLA compiler-generated code removes the data pushed on the stack. Therefore, it is safe to transfer control to some spot outside the *try..endtry* statement from within the unprotected section.

If an exception occurs in an unprotected section, the *try..endtry* statement containing that section does not handle the exception. Instead, control transfers to the (dynamically) nesting *try..endtry* statement (or to the HLA run-time system if there is no enclosing *try..endtry*).

If you re wondering why the *unprotected* section is necessary (after all, why not simply put the statements in the *unprotected* section after the *endtry*?), just keep in mind that both the protected sequence and the handled exceptions continue execution after the *endtry*. There may be some operations you want to perform after exceptions are released, but only if the protected block finished successfully. The *unprotected* section provides this capability. Perhaps the most common use of the *unprotected* section is to break out of a loop that repeats a *try..endtry* block until it executes without an exception occurring. The following code demonstrates this use:

```
forever

   try

         stdout.put( "Enter an integer: " );
         stdin.geti8();   // May raise an exception.

      unprotected

         break;

      exception( ex.ValueOutOfRange )

         stdout.put( "Value was out of range, reenter" nl );

      exception( ex.ConversionError )

         stdout.put( "Value contained illegal chars" nl );

   endtry;

endfor;
```

This simple example repeatedly asks the user to input an int8 integer until the value is legal and within the range of valid integers.

Another clause in the *try..except* statement is the *anyexception* clause. If this clause is present, it must be the last clause in the *try..except* statement, e.g.,

```
try
   << protected statements >>

<<
   unprotected

      Optional unprotected statements
>>

<< exception( constant ) // Note: may be zero or more of
                            of these.

      Optional exception handler statements
>>

   anyexception
      << Exception handler if none of the others execute >>

endtry;
```

Without the *anyexception* clause present, if the program raises an exception that is not specifically handled by one of the exception clauses, control transfers to the enclosing *try..endtry* statement. The *anyexception* clause gives a *try..endtry* statement the opportunity to handle any exception, even those that are not explicitly listed. Upon entry into the anyexception block, the EAX register contains the actual exception number.

The HLA *raise* statement generates an exception. The single parameter is an 8, 16, or 32-bit ordinal constant. Control is (ultimately) transferred to the first (most deeply nested) *try..endtry* statement that has a corresponding exception handler (including *anyexception*).

If the program executes the *raise* statement within the protected block of a *try..endtry* statement, then the enclosing *try..endtry* gets first shot at handling the exception. If the *raise* statement occurs in an *unprotected* block, or in an exception handler (including *anyexception*), then the next higher level (nesting) *try..endtry* statement will handle the exception. This allows *cascading* exceptions; that is, exceptions that the system handles in two or more exception handlers. Consider the following example:

```
try
   << Protected statements >>

  exception( someException )
   << Code to process this exception >>

   // The following re-raises this exception, allowing
   // an enclosing try..endtry statement to handle
   // this exception as well as this handler.

   raise( someException );

  << Additional, optional, exception handlers >>

endtry;
```

To understand the cost of using HLA's exception handling statements, it's wise to take a look at the code the compiler generates for these statements. Consider the following HLA code and it's conversion to "pure" assembly language:

```
   try
      << Protected Code >>

   exception( ex.ValueOutOfRange )
      << Code to execute if system raises "out of range" exception >>

   exception( ex.StringIndexError )
      << Code to execute if there is a string index exception >>

   endtry;

// "Pure" assembly code HLA produces for the above statements:

   push( &exceptionLabel );
   push( ebp );
   mov( ExceptionPtr__hla_, ebp ); // This is a global symbol HLA defines
   push( &HWexcept__hla_ );        //  "   "   " "  "   "  "   "   "    "
   mov( esp, ExceptionPtr__hla_ );

      << Protected Code >>

   mov( ExceptionPtr__hla_, esp ); // Code that cleans up the stack after
   pop( ExceptionPtr__hla_ );      //  executing the protected code.
   add( 8, esp );
   pop( ebp );
   add( 4, esp );
   jmp endTryLabel;

exceptionLabel:
   cmp( eax, 3 ); // ex.ValueOutOfRange = 3
   jne tryStrIndexEx
      << Code to execute if system raises "out of range" exception >>
   jmp endTryLabel;

tryStrIndexEx:
   cmp( eax, 2 ); // ex.StringIndexError = 2
   jne Raise__hla_; // Re-raises exception in surrounding try..endtry block.

endTryLabel:
```

As you can see by reading through this code, HLA emits five instructions upon encountering the *try* clause and emits six instructions after the protected block (i.e., before the first exception statement). Assuming exceptions don't occur very frequently (that is, they are the exceptions rather than the rule), most of the other statements that HLA emits will not execute frequently. These 11 statements before and after the protected code, however, execute everything the program encounters and leaves a *try..endtry* block. While 11 statements isn't a horrendous number, if you've only got a few statements within the *try..endtry* statement and you execute this sequence within a loop, this 11 statements can have an impact on your program's performance.

If the performance of the *try..endtry* statement is an issue, and you would still like to protect a sequence of statements with *try..endtry*, you can reduce the overhead by placing more statements inside the *try..endtry* protected region. For example, if you've got a *try..endtry* block inside a loop you'll have to execute that

try..endtry sequence on each iteration of the loop. However, if you place these statements around the loop, you only have to execute the 11 statements that the *try..endtry* block generates once for each execution of the loop. The drawback to widening the scope of the *try..endtry* statement is that it becomes more difficult to pinpoint the cause of the exception to just a few statements. For example, if you re reading a set of values into an array within a loop and you place the try..endtry block around the loop, you may not be able to determine which iteration of the loop triggered the exception (note that you cannot count on the registers containing reasonable values whenever someone raises an exception, so if your loop index is in a register, it s lost).

The big problem with exceptions is that they shouldn t occur under normal circumstances; therefore, the execution of those 11 statements is pure overhead, most of the time. Therefore, you want to be judicious with your use of the *try..endtry* block, especially in nested loops and other code sequences that execute frequently. It goes without saying that you should not use exception handling as a normal flow control sequence (that is, you shouldn t attempt to use the *try..endtry* block as a fancy form of a *switch/case* statement). The 11 statement overhead applies to code sequences that execute normally, without raising any exceptions. Processing an exception can require several dozen (or more) instructions. Normally, the overhead associated with processing an actual exception isn t an issue in most applications because exceptions rarely occur. However, if you attempt to use exceptions as a normal execution path, you ll probably be disappointed with the performance (note that the code that executes when an exception occurs is part of the HLA run-time system, it doesn t appear in the code that HLA emits for the *try..endtry* block shown earlier).

2.4: HLA Procedure Declarations and Invocations

HLA provides a very sophisticated system for declaring and calling procedures using a high level syntax. Indeed, HLA s procedure syntax, in many ways, is higher-level than most high level programming languages. We aren t going to get into all the gory details here because, quite frankly, all these features aren t necessary for writing Win32 applications (because C doesn t support these features). Nevertheless, HLA does have many useful features you won t normally find in an assembly language that help make writing Win32 assembly code much easier. We ll discuss those features in this section.

The general syntax for an HLA procedure declaration is:

```
procedure identifier ( optional_parameter_list );  procedure_options
   declarations
begin identifier;
   statements
end identifier;
```

The optional parameter list consists of a list of var-type declarations taking the form:

```
   optional_access_keyword  identifier1 : identifier2 optional_in_reg
```

optional_access_keyword, if present, must be *val, var, valres, result, name,* or *lazy* and defines the parameter passing mechanism (pass by value, pass by reference, pass by value/result [or value/returned], pass by result, pass by name, or pass by lazy evaluation, respectively). The default is pass by value (*val*) if an access keyword is not present. For pass by value parameters, HLA allocates the specified number of bytes according to the size of that object in the activation record. For pass by reference, pass by value/result, and pass by result, HLA allocates four bytes to hold a pointer to the object. For pass by name and pass by lazy evaluation, HLA allocates eight bytes to hold a pointer to the associated thunk and a pointer to the thunk s execution environment. Because Win32 applications typically only use pass by value and pass by reference, those are the two parameter

passing mechanisms we ll discuss in this book. For details on the other HLA parameter passing mechanisms, please consult the *HLA Reference Manual*.

The *optional_in_reg* clause, if present, corresponds to the phrase in *reg* where *reg* is one of the 80x86 s general purpose 8-, 16-, or 32-bit registers. This clause tells HLA to pass the parameter in the specified register rather than on the stack.

HLA also allows a special parameter of the form:

```
var identifier : var
```

This creates an *untyped* reference parameter. You may specify any memory variable as the corresponding actual parameter and HLA will compute the address of that object and pass it on to the procedure without further type checking. Within the procedure, the parameter is given the `dword` type.

The *procedure_options* component above is a list of keywords that specify how HLA emits code for the procedure. There are several different procedure options available: `@noalignstack`, `@alignstack`, `@pascal`, `@stdcall`, `@cdecl`, `@align( int_const)`, `@use reg32`, `@leave`, `@noleave`, `@enter`, `@noenter`, and `@returns( "text" )`. See Table 2-2 for a description of each of these procedure options.

Table 2-2: Procedure Options in an HLA Program

`@noframe`, `@frame`	By default, HLA emits code at the beginning of the procedure to construct a stack frame. The `@noframe` option disables this action. The `@frame` option tells HLA to emit code for a particular procedure if stack frame generation is off by default. HLA also uses these two special identifiers as a compile-time variable to set the default display generation for all procedures. Setting `@frame` to true (or `@noframe` to false) turns on frame generation by default; setting `@frame` to false (or `@noframe` to true) turns off frame generation.
`@nodisplay`, `@display`	By default, HLA emits code at the beginning of the procedure to construct a display within the frame. The `@nodisplay` option disables this action. The `@display` option tells HLA to emit code to generate a display for a particular procedure if display generation is off by default. Note that HLA does not emit code to construct the display if `'@noframe'` is in effect, though it will assume that the programmer will construct this display themselves. HLA also uses these two special identifiers as a compile-time variable to set the default display generation for all procedures. Setting `@display` to true (or `@nodisplay` to false) turns on display generation by default; setting `@display` to false (or `@nodisplay` to true) turns off display generation. Because most Win32 applications do not typically use nested procedures (and, in particular, none of the examples in this book use them), you'll usually see the following statement near the top of most example programs appearing in this book: `?nodisplay := true;`

@noalignstack, @alignstack	By default (assuming frame generation is active), HLA will an instruction to align ESP on a four-byte boundary after allocating local variables. Win32, Linux, and other 32-bit OSes require the stack to be dword-aligned (hence this option). If you know the stack will be dword-aligned, you can eliminate this extra instruction by specifying the *@noalignstack* option. Conversely, you can force the generation of this instruction by specifying the *@alignstack* procedure option. HLA also uses these two special identifiers as a compile-time variable to set the default display generation for all procedures. Setting *@alignstack* to true (or *@noalignstack* to false) turns on stack alignment generation by default; setting *@alignstack* to false (or *@noalignstack* to true) turns off stack alignment code generation. Because Windows guarantees that the stack will be aligned when it transfers control to a procedure in your application, and because programs in this book never force a misalignment of the stack, you'll commonly see the following statement near the beginning of the HLA example programs in this book: `?@noalignstack := true;`
@pascal, @cdecl, @stdcall	These options give you the ability to specify the parameter passing mechanism for the procedure. By default, HLA uses the *@pascal* calling sequence. This calling sequence pushes the parameters on the stack in a left-to-right order (i.e., in the order they appear in the parameter list). The *@cdecl* procedure option tells HLA to pass the parameters from right-to-left so that the first parameter appears at the lowest address in memory and that it is the user's responsibility to remove the parameters from the stack. The *@stdcall* procedure option is a hybrid of the *@pascal* and *@cdecl* calling conventions. It pushes the parameters in the right-to-left order (just like *@cdecl*) but *@stdcall* procedures automatically remove their parameter data from the stack (just like *@pascal*). Win32 API calls use the *@stdcall* calling convention. In the Win32 examples appearing in this book, you'll see the following conventions adhered to: ¥ All Win32 API functions use the *@stdcall* calling convention (there are some Windows API functions that require the *@cdecl* calling convention, but this book does not call those routines). ¥ All functions in the application that Windows calls (the window procedure and any callback functions) use the *@stdcall* calling convention. ¥ Local procedures in the program (those that only HLA code calls) use the *@pascal* calling convention.
@align(int_constant)	The *@align(int_const)* procedure option aligns the procedure on a 1, 2, 4, 8, or 16 byte boundary. Specify the boundary you desire as the parameter to this option. The default is *@align(1)*, which is unaligned; HLA also uses this special identifier as a compile-time variable to set the default procedure alignment . Setting *@align := 1* turns off procedure alignment while supplying some other value (which must be a power of two) sets the default procedure alignment to the specified number of bytes.

@use reg32	When passing parameters, HLA can sometimes generate better code if it has a 32-bit general purpose register for use as a scratchpad register. By default, HLA never modifies the value of a register behind your back; so it will often generate less than optimal code when passing certain parameters on the stack. By using the *@use* procedure option, you can specify one of the following 32-bit registers for use by HLA: EAX, EBX, ECX, EDX, ESI, or EDI. By providing one of these registers, HLA may be able to generate significantly better code when passing certain parameters. Note that it is not legal for you to specify the EBP or ESP registers after an *@use* option. Procedures only require a single 32-bit register, so if you specify multiple *@use* options, HLA only uses the last register you specify. Although HLA allows the use of ESI and EDI, HLA generates better code in certain circumstances if you specify one of EAX, EBX, ECX or EDX.
@returns("text")	This option specifies the compile-time return value whenever a function name appears as an instruction operand. For example, suppose you are writing a function that returns its result in EAX. You should probably specify a "returns" value of "EAX" so you can compose that procedure just like any other HLA machine instruction. For more details on this option, check out the discussion of "instruction composition" in the *HLA Reference Manual*.

@leave, @noleave	These two options control the code generation for the standard exit sequence. If you specify the *@leave* option then HLA emits the x86 *leave* instruction to clean up the activation record before the procedure returns. If you specify the *@noleave* option, then HLA emits the primitive instructions to achieve this, e.g.,
	mov(ebp, esp);
	pop(ebp);
	
	The manual sequence is faster on some architectures, the *leave* instruction is always shorter.
	
	Note that *@noleave* occurs by default if you've specified *@noframe*. By default, HLA assumes *@noleave* but you may change the default using these special identifiers as a compile-time variable to set the default *leave* generation for all procedures. Setting *@leave* to true (or *@noleave* to false) turns on *leave* generation by default; setting *@leave* to false (or *@noleave* to true) turns off the use of the *leave* instruction.

@enter, @noenter	These two options control the code generation for a procedure's standard entry sequence. If you specify the *@enter* option then HLA emits the x86 *enter* instruction to create the activation record. If you specify the *@noenter* option, then HLA emits the primitive instructions to achieve this. The manual sequence is always faster, using the *enter* instruction is usually shorter. Note that *@noenter* occurs by default if you've specified *@noframe*. By default, HLA assumes *@noenter* but you may change the default using these special identifiers as a compile-time variable to set the default *enter* generation for all procedures. Setting *@enter* to true (or *@noenter* to false) turns on *enter* generation by default; setting *@enter* to false (or *@noenter* to true) turns off the use of the *enter* instruction. The enter instruction is primarily of interest to programmers using nested procedures and displays (see the @nodisplay option). Because most Win32 applications don't use nested procedures and a display, you'll rarely see this option used in a Win32 program.

The *@returns* option and instruction composition in HLA deserve a special mention. Instruction composition is a feature in HLA that allows you to substitute an HLA instruction (or other statement) in place of certain instruction operands. For example, consider the following perfectly legal HLA statement:

```
mov( mov( 0, eax ), ebx );
```

To process a statement like this, HLA associates a compile-time return value with every instruction. The return value is a string (text) value and is usually the destination operand of the instruction. For example, the destination operand of the nested instruction in this example is eax so HLA substitutes EAX for the nested instruction after emitting the code for that instruction. In other words, this particular sequence is equivalent to the following:

```
mov( 0, eax );
mov( eax, ebx );  // First operand was the result of instruction composition.
```

You may also supply procedure invocations as instruction operands. When HLA encounters a procedure invocation in an instruction operand, it emits the code for that procedure call and then substitutes the returns string for that procedure invocation (the returns string is the string operand you supply in the *@returns* procedure option). The following example demonstrates how the *@returns* option works:

```
program returnsDemo;
#include( "stdio.hhf" );

   procedure eax0; @returns( "eax" );
   begin eax0;

      mov( 0, eax );

   end eax0;

begin returnsDemo;

   mov( eax0(), ebx );
   stdout.put( "ebx=", ebx, nl );

end returnsDemo;
```

If you do not explicitly supply an *@returns* procedure option, then HLA will use the empty string as the *@returns* value for that procedure. Obviously, if you compose such a procedure invocation inside some other instruction you ll probably create a syntax error.

You ll rarely see code that uses procedure invocations in an instruction as in these examples. However, instruction composition can take place in other areas too. For example, the operands of a boolean expression in an HLA *if* statement also support instruction composition, allowing you to specify statements like the following:

```
if( eax0() <> 0 ) then
   .
   .
   .
endif;
```

HLA will emit the code for the call to the procedure (*eax0* in this example) and then substitute the procedure s *@returns* value for the procedure invocation in the boolean expression (i.e., eax in this example).

To help those who insist on constructing the activation record themselves, HLA declares two local constants within each procedure: *_vars_* and *_parms_*. The *_vars_* symbol is an integer constant that specifies the number of local variables declared in the procedure. This constant is useful when allocating storage for your local variables. The *_parms_* constants specifies the number of bytes of parameters. You would normally supply this constant as the parameter to a *ret()* instruction to automatically clean up the procedure s parameters when it returns. Here s an example of these two constants in use:

```
procedure hasParmsAndLocals( p1:dword; p2:int32; p3:uns32 ); @noframe; @nodisplay;
var
   localA :dword;
   localB :dword;
   localC :dword;
   localD :dword;
begin hasParmsAndLocals;

   push( ebp ); // we have to build the activation record ourselves!
   mov( esp, ebp );
   sub( _vars_, esp ); // make room for local variables (16 bytes, as it turns out).
      .
      .
      .
   mov( ebp, esp );  // Clean up the activation record on return
   pop( ebp );       // Note that we could also use "leave" here.
   ret( _parms_ );   // Return and remove parameters from the stack (12 bytes).
```

If you do not specify *@nodisplay*, then HLA defines a run-time variable named *_display_* that is an array of pointers to activation records. As Win32 applications generally do not use nested procedures and displays, we will not discuss the use of the @nodisplay option (or displays) in this book. For more details on displays and the @nodisplay (or @display) procedure option, please consult the *HLA Reference Manual*.

You can also declare *@external* procedures (procedures defined in other HLA units or written in languages other than HLA) using the following syntaxes:

```
procedure externProc1 (optional parameters) ; @returns( "text" ); @external;

procedure externProc2 (optional parameters) ;
   @returns( "text" ); @external( "different_name" );
```

As with normal procedure declarations, the parameter list and *@returns* clause are optional.

The first form is generally used for HLA-written functions. HLA will use the procedure s name (*externProc1* in this case) as external name.

The second form lets you refer to the procedure by one name (*externProc2* in this case) within your HLA program and by a different name (*different_name* in this example) in the MASM generated code. This second form has two main uses: (1) if you choose an external procedure name that just happens to be a MASM reserved word, the program may compile correctly but fail to assemble. Changing the external name to something else solves this problem. (2) When calling procedures written in external languages you may need to specify characters that are not legal in HLA identifiers. For example, Win32 API calls often use names like *WriteFile@24* containing illegal (in HLA) identifier symbols. The string operand to the external option lets

you specify any name you choose. Of course, it is your responsibility to see to it that you use identifiers that are compatible with the linker and MASM, HLA doesn t check these names.

By default, HLA does the following:

- Creates a display for every procedure.
- Emits code to construct the stack frame for each procedure.
- Emits code to align ESP on a four-byte boundary upon procedure entry.
- HLA assumes that it cannot modify any register values when passing (non-register) parameters.
- The first instruction of the procedure is unaligned.

These options are the most general and safest for beginning assembly language programmers. However, the code HLA generates for this general case may not be as compact or as fast as is possible in a specific case. For example, few procedures will actually need a display data structure built upon procedure activation. Therefore, the code that HLA emits to build the display can reduce the efficiency of the program. Most Win32 application programmers, of course, will want to use procedure options like *@nodisplay* to tell HLA to skip the generation of this code. However, if a program contains many procedures and none of them need a display, continually adding the *@nodisplay* option can get really old. Therefore, HLA allows you to treat these directives as pseudo-compile-time-variables to control the default code generation. E.g.,

```
? @display := true; // Turns on default display generation.
? @display := false; // Turns off default display generation.
? @nodisplay := true; // Turns off default display generation.
? @nodisplay := false; // Turns on default display generation.

? @frame := true; // Turns on default frame generation.
? @frame := false; // Turns off default frame generation.
? @noframe := true; // Turns off default frame generation.
? @noframe := false; // Turns on default frame generation.

? @alignstack := true; // Turns on default stk alignment code generation.
? @alignstack := false; // Turns off default stk alignment code generation.
? @noalignstack := true; // Turns off default stk alignment code generation.
? @noalignstack := false; // Turns on default stk alignment code generation.

? @enter := true; // Turns on default ENTER code generation.
? @enter := false; // Turns off default ENTER code generation.
? @noenter := true; // Turns off default ENTER code generation.
? @noenter := false; // Turns on default ENTER code generation.

? @leave := true; // Turns on default LEAVE code generation.
? @leave := false; // Turns off default LEAVE code generation.
? @noleave := true; // Turns off default LEAVE code generation.
? @noleave := false; // Turns on default LEAVE code generation.

?@align := 1; // Turns off procedure alignment (align on byte boundary).
?@align := int_expr; // Sets alignment, must be a power of two.
```

These directives may appear anywhere in the source file. They set the internal HLA default values and all procedure declarations following one of these assignments (up to the next, corresponding assignment) use the specified code generation option(s). Note that you can override these defaults by using the corresponding procedure options mentioned earlier.

2.4.1: Disabling HLA's Automatic Code Generation for Procedures

Before jumping in and describing how to use the high level HLA features for procedures, the best place to start is with a discussion of how to disable these features and write plain old fashioned assembly language code. This discussion is important because procedures are the one place where HLA automatically generates a lot of code for you and many assembly language programmers prefer to control their own destinies; they don t want the compiler to generate any excess code for them. Though most Win32 assembly language programmers use HLAs high level procedure features, many may want to disable some or all of HLAs code generation features for procedures. So disabling HLAs automatic code generation capabilities is a good place to start this discussion.

By default, HLA automatically emits code at the beginning of each procedure to do five things: (1) Preserve the pointer to the previous activation record (EBP); (2) build a display in the current activation record; (3) allocate storage for local variables; (4) load EBP with the base address of the current activation record; (5) adjust the stack pointer (downwards) so that it points at a dword-aligned address.

When you return from a procedure, by default HLA will deallocate the local storage and return, removing any parameters from the stack.

To understand the code that HLA emits, consider the following simple procedure:

```
procedure p( j:int32 );
var
    i:int32;
begin p;
end p;
```

Here is a dump of the symbol table that HLA creates for procedure p[6]:

```
p   <0,proc>:Procedure type (ID=?1_p)
    --------------------------------
    _vars_          <1,cons>:uns32, (4 bytes)  =4
    i               <1,var >:int32, (4 bytes, ofs:-12)
    _parms_         <1,cons>:uns32, (4 bytes)  =4
    _display_       <1,var >:dword, (8 bytes, ofs:-4)
    j               <1,valp>:int32, (4 bytes, ofs:8)
    p               <1,proc>:
-----------------------------------
```

The important thing to note here is that local variable *i* is at offset -12 and HLA automatically created an eight-bit local variable named *_display_* which is at offset -4.

HLA emits code similar to the following for the procedure above[7]:

```
procedure p; @nodisplay; @noframe; @nostackalign;
begin p;
      push( ebp );        //Dynamic link (pointer to previous activation record)
      pushd( [ebp-4] );   //Display for lex level 0
      lea( ebp, [esp+04] );  //Get frame ptr (point EBP at current activation record)
```

6. Note that the symbol table output routines in HLA change from time to time, so this output may not exactly match what HLA produces in the particular version that you re using.
7. Again, code generation is subject to change in HLA, so this may not exactly match what the particular version of HLA that you re using produces.

```
      push( ebp );           //Ptr to this proc's A.R. (part of display construction)
      sub( 4, esp );         //Local storage.
      and( $fffffffc, esp ); //dword-align stack

// Exit point for the procedure:

      mov( ebp, esp );  //Deallocate local variables.
      pop( ebp );       //Restore pointer to previous activation record.
      ret( 4 );         //Return, popping parameters from the stack.
end p;
```

Building the display data structure is not very common in standard assembly language programs. This is only necessary if you are using nested procedures and those nested procedures need to access non-local variables. Because this is a rare situation, many programmers will immediately want to tell HLA to stop emitting the code to generate the display. This is easily accomplished by adding the *@nodisplay* procedure option to the procedure declaration. Adding this option to procedure *p* produces the following:

```
procedure p( j:int32 ); @nodisplay;
var
    i:int32;
begin p;
end p;
```

Compiling this procedures the following symbol table dump:

```
p               <0,proc>:Procedure type (ID=?1_p)
    ---------------------------------
    _vars_          <1,cons>:uns32, (4 bytes)  =4
    i               <1,var >:int32, (4 bytes, ofs:-4)
    _parms_         <1,cons>:uns32, (4 bytes)  =4
    j               <1,valp>:int32, (4 bytes, ofs:8)
    p               <1,proc>:
-----------------------------------
```

Note that the *_display_* variable is gone and the local variable *i* is now at offset -4. Here is the code that HLA emits for this new version of the procedure:

```
procedure p; @nodisplay; @noframe; @nostackalign;
begin p;
      push( ebp );           //Dynamic link (pointer to previous activation record)
      mov( esp, ebp );       //Point EBP at current activation record.
      sub( 4, esp );         //Local storage.
      and( $fffffffc, esp ); //dword-align stack

// Exit point for the procedure:

      mov( ebp, esp );  //Deallocate local variables.
      pop( ebp );       //Restore pointer to previous activation record.
      ret( 4 );         //Return, popping parameters from the stack.
end p;
```

As you can see, this code is smaller and a bit less complex. Unlike the code that built the display, it is fairly common for an assembly language programmer to construct an activation record in a manner similar to this.

Indeed, about the only instruction out of the ordinary in this example is the *and* instruction that dword-aligns the stack (Win32 calls require the stack to be double word aligned, and the system performance is much better if the stack is double word aligned).

This code is still relatively inefficient if you don t pass parameters on the stack and you don t use automatic (non-static, local) variables. Many assembly language programmers pass their few parameters in machine registers and also maintain local values in the registers. If this is the case, then the code above is pure overhead. You can inform HLA that you wish to take full responsibility for the entry and exit code by using the *"@noframe"* procedure option. Consider the following version of *p*:

```
procedure p( j:int32 ); @nodisplay; @noframe;
var
    i:int32;
begin p;
end p;
```

(this produces the same symbol table dump as the previous example).

HLA emits the following code for this version of *p*:

```
procedure p; @nodisplay; @noframe; @nostackalign;
begin p;
end p;
```

Whoa! There s nothing there! But this is exactly what the advanced assembly language programmer wants. With both the *@nodisplay* and *@noframe* options, HLA does not emit any extra code for you. You would have to write this code yourself.

By the way, you *can* specify the *@noframe* option without specifying the *@nodisplay* option. HLA still generates no extra code, but it will assume that you are allocating storage for the display in the code you write. That is, there will be an eight-byte *_display_* variable created and *i* will have an offset of -12 in the activation record. It will be your responsibility to deal with this. Although this situation is possible, it s doubtful this combination will be used much at all.

Note a major difference between the two versions of *p* when *@noframe* is not specified and *@noframe* is specified: if *@noframe* is not present, HLA automatically emits code to return from the procedure. This code executes if control falls through to the *"end p;"* statement at the end of the procedure. Therefore, if you specify the *@noframe* option, you must ensure that the last statement in the procedure is a *ret()* instruction or some other instruction that causes an unconditional transfer of control. If you do not do this, then control will fall through to the beginning of the next procedure in memory, probably with disastrous results.

The *ret()* instruction presents a special problem. It is dangerous to use this instruction to return from a procedure that does not have the *@noframe* option. Remember, HLA has emitted code that pushes a lot of data onto the stack. If you return from such a procedure without first removing this data from the stack, your program will probably crash. The correct way to return from a procedure without the *@noframe* option is to jump to the bottom of the procedure and run off the end of it. Rather than require you to explicitly put a label into your program and jump to this label, HLA provides the *exit procname;* instruction. HLA compiles the *exit* instruction into a *jmp* that transfers control to the clean-up code HLA emits at the bottom of the procedure. Consider the following modification of *p* and the resulting assembly code produced:

```
procedure p( j:int32 ); @nodisplay;
var
    i:int32;
begin p;
    exit p;
    nop();
end p;


procedure p( j:int32 ); @noframe; @nodisplay;
var
    i:int32;
begin p;
    push( ebp )
    mov( esp, ebp )l
    sub( 4, esp );
    and( $fffffffc, esp );
    jmp     _x_2_p;          // This is what the exit statement compiles to
    nop;
_x_2_p:
    mov( ebp, esp );
    pop( ebp );
    ret( 4 );
end p;
```

As you can see, HLA automatically emits a label to the assembly output file (*"_x_2_p"* in this instance) at the bottom of the procedure where the clean-up code starts. HLA translates the *"exit p;"* instruction into a *jmp* to this label.

If you look back at the code emitted for the version of *p* with the *@noframe* option, you ll note that HLA did not emit a label at the bottom of the procedure. Therefore, HLA cannot generate a jump to this nonexistent label, so you cannot use the *exit* statement in a procedure with the *@noframe* option (HLA will generate an error if you attempt this).

Of course, HLA will *not* stop you from putting a *ret()* instruction into a procedure without the *@noframe* option (some people who know exactly what they are doing might actually want to do this). Keep in mind, if you decide to do this, that you must deallocate the local variables (that s what the *"mov esp, ebp"* instruction is doing), you need to restore EBP (via the *"pop ebp"* instruction above), and you need to deallocate any parameters pushed on the stack (the *"ret 4"* handles this in the example above). The following code demonstrates this:

```
procedure p( j:int32 ); @nodisplay;
var
    i:int32;
begin p;

    if( j = 0 ) then

        // Deallocate locals.

        mov( ebp, esp );

        // Restore old EBP

        pop( ebp );
```

```
        // Return and pop parameters

        ret( 4 );

    endif;
    nop();
end p;


procedure p; @nodisplay; @noframe;
begin p;
   push( ebp );
   mov( esp, ebp );
   sub( 4, esp );
   and( $fffffffc, esp );

   cmp( (type dword [ebp+8]), 0 );
   jne _2_false;
   mov( ebp, esp );
   pop( ebp );
   ret( 4 );

_2_false:
   nop;
   mov( ebp, esp );
   pop( ebp );
   ret( 4 );
end p;
```

If real assembly language programmers would generally specify both the *@noframe* and *@nodisplay* options, why not make them the default case (and use *"@frame"* and *"@display"* options to specify the generation of the activation record and display)? Well, keep in mind that HLA was originally designed as a tool to teach assembly language programming to beginning students and this default behavior is safer for beginning programmers. Also, most Win32 applications use these defaults (that is, most Win32 application programmers do not specify @noframe, regardless of how advanced they are) so these defaults make a lot of sense for Win32 applications.

If you are absolutely certain that your stack pointer is aligned on a four-byte boundary upon entry into a procedure, you can tell HLA to skip emitting the *and($ffff_fffc, esp);* instruction by specifying the *@noalignstack* procedure option. Because Win32 apps always keep the stack dword aligned, most Win32 applications set the stack alignment to off via a statement like this one near the beginning of the source file:

```
?@nostackalign := true;
```

Note that specifying *@noframe* also specifies *@noalignstack*.

Note that HLA does provide a few additional procedure options. Please consult the *HLA Reference Manual* for more details on these options. They are not of immediate interest to us here, so this book will ignore them.

2.4.2 Calling HLA Procedures

There are two standard ways to call an HLA procedure: use the call instruction or simply specify the name of the procedure as an HLA statement. Both mechanisms have their pluses and minuses.

To call an HLA procedure using the *call* instruction is exceedingly easy. Simply use either of the following syntaxes:

```
call( procName );
call procName;
```

Either form compiles into an 80x86 *call* instruction that calls the specified procedure. The difference between the two is that the first form (with the parentheses) returns the procedure s returns value, so this form can appear as an operand to another instruction. The second form above always returns the empty string, so it is not suitable as an operand of another instruction. Also, note that the second form requires a statement or procedure label, you may not use memory addressing modes in this form; on the other hand, the second form is the only form that lets you call a statement label (as opposed to a procedure label); this form is useful on occasion.

If you use the *call* statement to call a procedure, then you are responsible for passing any parameters to that procedure. In particular, if the parameters are passed on the stack, you are responsible for pushing those parameters (in the correct order) onto the stack before the call. This is a lot more work than letting HLA push the parameters for you, but in certain cases you can write more efficient code by pushing the parameters yourself.

The second way to call an HLA procedure is to simply specify the procedure name and a list of actual parameters (if needed) for the call. This method has the advantage of being easy and convenient at the expense of a possible slight loss in efficiency and flexibility. This calling method should also prove familiar to most HLL programmers. As an example, consider the following HLA program:

```
program parameterDemo;

#include( "stdio.hhf" );

procedure PrtAplusB( a:int32; b:int32 ); @nodisplay;
begin PrtAplusB;

    mov( a, eax );
    add( b, eax );
    stdout.put( "a+b=", (type int32 eax ), nl );

end PrtAplusB;

static
    v1:int32 := 25;
    v2:int32 := 5;

begin parameterDemo;

    PrtAplusB( 1, 2 );
    PrtAplusB( -7, 12 );
    PrtAplusB( v1, v2 );

    mov( -77, eax );
    mov( 55, ebx );
```

```
    PrtAplusB( eax, ebx );

end parameterDemo;
```

This program produces the following output:

```
a+b=3
a+b=5
a+b=30
a+b=-22
```

As you can see, the calls to *PrtAplusB* in HLA are very similar to calling procedures (and passing parameters) in a high level language like C/C++ or Pascal. There are, however, some key differences between and HLA call and a HLL procedure call. The next section will cover those differences in greater detail. The important thing to note here is that if you choose to call a procedure using the HLL syntax (that is, the second method described here), you can pass the parameters in the parameter list and let HLA push the parameters for you. If you want to take complete control over the parameter passing code, one way to do this is to use the *call* instruction.

2.4.3: Parameter Passing in HLA, Value Parameters

The previous section probably gave you the impression that passing parameters to a procedure in HLA is nearly identical to passing those same parameters to a procedure in a high level language. The truth is, the examples in the previous section were rigged. There are actually many restrictions on how you can pass parameters to an HLA procedure. This section discusses the parameter passing mechanism in detail.

The most important restriction on actual parameters in a call to an HLA procedure is that HLA only allows memory variables, registers, constants, and certain other special items as parameters. In particular, you cannot specify an arithmetic expression that requires computation at run-time (although a constant expression, computable at compile time is okay). The bottom line is this: if you need to pass the value of an expression to a procedure, you must compute that value prior to calling the procedure and pass the result of the computation; HLA will not automatically generate the code to compute that expression for you.

The second point to mention here is that HLA is a strongly typed language when it comes to passing parameters. This means that with only a few exceptions, the type of the actual parameter must exactly match the type of the formal parameter. If the actual parameter is an int8 object, the formal parameter had better not be an int32 object or HLA will generate an error. The only exceptions to this rule are the *byte, word,* and *dword* types. If a formal parameter is of type *byte*, the corresponding actual parameter may be any one-byte data object. If a formal parameter is a *word* object, the corresponding actual parameter can be any two-byte object. Likewise, if a formal parameter is a *dword* object, the actual parameter can be any four-byte data type. Conversely, if the actual parameter is a *byte, word,* or *dword* object, it can be passed without error to any one, two, or four-byte actual parameter (respectively). Programmers who are really lazy make all their parameters *bytes, words*, or *dwords* (at least, wherever possible). Programmers who care about the quality of their code use untyped parameters cautiously.

If you want to use the high level calling sequence, but you don t like the inefficient code HLA sometimes produces when generating code to pass your parameters, you can always use the #{...}# sequence parameter to override HLAs code generation and substitute your own code for one or two parameters. Of course, it doesn t make any sense to pass all the parameters is a procedure using this trick, it would be far easier just to use the call instruction. Here s an example of using HLAs manual parameter passing facility within a HLL style procedure call:

```
   PrtAplusB
   (
      #{
         mov( i, eax );   // First parameter is "i+5"
         add( 5, eax );
         push( eax );
      }#,
      5
   );
```

HLA will automatically copy an actual value parameter into local storage for the procedure, regardless of the size of the parameter. If your value parameter is a one million byte array, HLA will allocate storage for 1,000,000 bytes and then copy that array in on each call. C/C++ programmers may expect HLA to automatically pass arrays by reference (as C/C++ does) but this is not the case. If you want your parameters passed by reference, you must explicitly state this.

The code HLA generates to copy value parameters, while not particularly bad, certainly isn t optimal. If you need the fastest possible code when passing parameters by value on the stack, it would be better if you explicitly pushed the data yourself. Another alternative that sometimes helps is to use the use reg_{32} procedure option to provide HLA with a hint about a 32-bit scratchpad register that it can use when building parameters on the stack. For more details on this facility, please see the *HLA Reference Manual* or *The Art of Assembly Language*.

2.4.4: Parameter Passing in HLA: Reference Parameters

The one good thing about pass by reference parameters is that they are always four byte pointers, regardless of the size of the actual parameter. Therefore, HLA has an easier time generating code for these parameters than it does generating code for pass by value parameters.

Like high level languages, HLA places a whopper of a restriction on pass by reference parameters: they can only be memory locations. Constants and registers are not allowed since you cannot compute their address. Do keep in mind, however, that any valid memory addressing mode is a valid candidate to be passed by reference; you do not have to limit yourself to static and local variables. For example, *"[eax]"* is a perfectly valid memory location, so you can pass this by reference (assuming you type-cast it, of course, to match the type of the formal parameter). The following example demonstrate a simple procedure with a pass by reference parameter:

```
program refDemo;

#include( "stdio.hhf" );

    procedure refParm( var a:int32 );
    begin refParm;

        mov( a, eax );
        mov( 12345, (type int32 [eax]));

    end refParm;

    static
        i:int32:=5;

begin refDemo;

    stdout.put( "(1) i=", i, nl );
```

```
    mov( 25, i );
    stdout.put( "(2) i=", i, nl );
    refParm( i );
    stdout.put( "(3) i=", i, nl );


end refDemo;
```

The output produced by this code is

```
(1) i=5
(2) i=25
(3) i=12345
```

As you can see, the parameter a in *refParm* exhibits pass by reference semantics since the change to the value *a* in *refParm* changed the value of the actual parameter (*i*) in the main program.

Note that HLA passes the address of *i* to *refParm*, therefore, the *a* parameter contains the address of *i*. When accessing the value of the *i* parameter, the *refParm* procedure must deference the pointer passed in *a"*. The two instructions in the body of the *refParm* procedure accomplish this.

Take a look at the code that HLA generates for the call to *refParm*:

```
        pushd( &i );
        call refParm;
```

As you can see, this program simply computed the address of *i* and pushed it onto the stack. Now consider the following modification to the main program:

```
program refDemo;

#include( "stdio.hhf" );

    procedure refParm( var a:int32 );
    begin refParm;

        mov( a, eax );
        mov( 12345, (type int32 [ eax] ));

    end refParm;

    static
        i:int32:=5;

    var
        j:int32;

begin refDemo;

    mov( 0, j );
    refParm( j );
    refParm( i );
    lea( eax, j );
    refParm( [ eax] );
```

```
end refDemo;
```

This version emits the following code:

```
mov( 0, (type dword [ ebp-8] ) );
push( eax );
lea( eax, (type dword [ ebp-8] ));
xchg( eax, [ esp] );
call refParm;

pushd( &i );
call refParm;

lea( eax, (type dword [ ebp-8] ));
push( eax );
push( eax );
lea( eax, (type dword [ eax+0] ));
mov( eax, [ esp+4] );
pop( eax );
call refParm;
```

As you can see, the code emitted for the last call is pretty ugly (we could easily get rid of three of the instructions in this code). This call would be a good candidate for using the *call* instruction directly. Also see Hybrid Parameters a little later in this book. Another option is to use the *"@use reg32"* option to tell HLA it can use one of the 32-bit registers as a scratchpad. Consider the following:

```
procedure refParm( var a:int32 ); use eax;
      .
      .
      .
    lea( ebx, j );
    refParm( [ ebx] );
```

This sequence generates the following code (which is a little better than the previous example):

```
lea( ebx, j );
lea( eax, (type dword [ ebx+0] ));
push( eax );
call refParm;
```

As a general rule, the type of an actual reference parameter must exactly match the type of the formal parameter. There are a couple exceptions to this rule. First, if the formal parameter is *dword*, then HLA will allow you to pass any four-byte data type as an actual parameter by reference to this procedure. Second, you can pass an actual *dword* parameter by reference if the formal parameter is a four-byte data type. There are some other exceptions that we ll explore in a later section of this chapter.

2.4.5: Untyped Reference Parameters

HLA provides a special formal parameter syntax that tells HLA that you want to pass an object by reference and you don t care what its type is. Consider the following HLA procedure:

```
procedure zeroObject( var object:byte; size:uns32 );
begin zeroObject;
   << code to write "size" zeros to "object" >
end zeroObject;
```

The problem with this procedure is that you will have to coerce non-byte parameters to a byte before passing them to *zeroObject*. That is, unless you re passing a byte parameter, you ve always got to call *zeroObject* thusly:

```
   zeroObject( (type byte NotAByte), sizeToZero );
```

For some functions you call frequently with different types of data, this can get painful very quickly.

The HLA untyped reference parameter syntax solves this problem. Consider the following declaration of *zeroObject*:

```
procedure zeroObject( var object:var; size:uns32 );
begin zeroObject;
   << code to write "size" zeros to "object" >
end zeroObject;
```

Notice the use of the reserved word *var* instead of a data type for the object parameter. This syntax tells HLA that you re passing an arbitrary variable by reference. Now you can call *zeroObject* and pass any (memory) object as the first parameter and HLA won t complain about the type, e.g.,

```
   zeroObject( NotAByte, sizeToZero );
```

Note that you may only pass untyped objects by reference to a procedure.

Note, and this is very important, when you pass an actual parameter to a procedure that has an untyped reference parameter in that slot, HLA will always take the address of the object you pass it. This is true even if the actual parameter is a pointer to some data (which is in direct contrast to what HLA normally does with pointer objects that you pass as reference parameters). For more details, read the section on Passing Pointers and Values as Reference Parameters on page 91.

2.4.6: Hybrid Parameter Passing in HLA

HLA s automatic code generation for parameters specified using the high level language syntax isn t always optimal. In fact, sometimes it is downright inefficient. This is because HLA makes very few assumptions about your program. For example, suppose you are passing a word parameter to a procedure by value. Since all parameters in HLA consume an even multiple of four bytes on the stack, HLA will zero extend the word and push it onto the stack. It does this using code like the following:

```
   pushw( 0 );
   pushw( Parameter );
```

Clearly you can do better than this if you know something about the variable. For example, if you know that the two bytes following *Parameter* are in memory (as opposed to being in the next page of memory that isn t allocated, and access to such memory would cause a protection fault), you could get by with the single instruction:

```
push( (type dword Parameter) );
```

Unfortunately, HLA cannot make these kinds of assumptions about the data because doing so could create malfunctioning code (admittedly, in very rare circumstances).

One solution, of course, is to forego the HLA high level language syntax for procedure calls and manually push all the parameters yourself and call the procedure via the *call* instruction. However, this is a major pain that involves lots of extra typing and produces code that is difficult to read and understand. Therefore, HLA provides a hybrid parameter passing mechanism that lets you continue to use a high level language calling syntax yet still specify the exact instructions needed to pass certain parameters. This hybrid scheme works out well because HLA actually does a good job with most parameters (e.g., if they are an even multiple of four bytes, HLA generates efficient code to pass the parameters; it s only those parameters that have a weird size that HLA generates less than optimal code for).

If a parameter consists of the #{ and }# brackets with some corresponding code inside the brackets, HLA will emit the code inside the brackets in place of any code it would normally generate for that parameter. So if you wanted to pass a 16-bit parameter efficiently to a procedure named *Proc* and you re sure there is no problem accessing the two bytes beyond this parameter, you could use code like the following:

```
Proc( #{ push( (type dword WordVar) ); }# );
```

Whenever you pass a non-static[8] variable as a parameter by reference, HLA generates the following code to pass the address of that variable to the procedure:

```
push( eax );
push( eax );
lea( eax, Variable );
mov( eax, [ esp+4] );
pop( eax );
```

It generates this particular code to ensure that it doesn t change any register values (after all, you could be passing some other parameter in the EAX register). If you have a free register available, you can generate slightly better code using a calling sequence like the following (assuming EBX is free):

```
HasRefParm
(
   #{
         lea( ebx, Variable );
         push( ebx );
   }#
);
```

2.4.7: Parameter Passing in HLA, Register Parameters

HLA provides a special syntax that lets you specify that certain parameters are to be passed in registers rather than on the stack. The following are some examples of procedure declarations that use this feature:

8. Static variables are those you declare in the static, readonly, and storage sections. Non-static variables include parameters, VAR objects, and anonymous memory locations.

```
procedure a( u:uns32 in eax ); forward;
procedure b( w:word in bx ); forward;
procedure d( c:char in ch ); forward;
```

Whenever you call one of these procedures, the code that HLA automatically emits for the call will load the actual parameter value into the specified register rather than pushing this value onto the stack. You may specify any general purpose 8-bit, 16-bit, or 32-bit register after the *in* keyword following the parameter s type. Obviously, the parameter must fit in the specified register. You may only pass reference parameters in 32-bit registers; you cannot pass parameters that are not one, two, or four bytes long in a register. Also note that HLA does not support passing parameters in FPU, MMX, or XMM registers (at least, using the high level syntax).

You can get in to trouble if you re not careful when using register parameters, consider the following two procedure definitions:

```
procedure one( u:uns32 in eax; v:dword in ebx ); forward;
procedure two( a:uns32 in eax );
begin two;

   one( 25, a );

end two;
```

The call to *one* in procedure *two* looks like it passes the values 25 and whatever was passed in for *a* in procedure *two*. However, if you study the HLA output code, you will discover that the call to *one* passes 25 for both parameters. They reason for this is because HLA emits the code to load 25 into EAX in order to pass 25 in the *u* parameter. Unfortunately, this wipes out the value passed into *two* in the *a* variable, hence the problem. Be aware of this if you use register parameters often.

2.4.8: Passing Pointers and Values as Reference Parameters

When it comes to pass by reference parameters, HLA relaxes the type checking rules a little bit in order to make calling procedures with reference parameters a bit more convenient. For the most part, this relaxation (which you ll see in a moment) is intuitive and works well. However, there are a few special cases where the lack of strict type checking creates some surprising results; so we ll discuss this issue in detail here.

To understand the motivation for HLA s relaxation of strict type-checking, consider the following procedure declaration:

```
procedure uns8RefParm( var b:uns8 );
begin uns8RefParm;
    .
    .
    .
end uns8RefParm;
```

Whenever you call this procedure, you have to pass it an actual parameter that is either an *uns8* typed object or a *byte* object (remember, you can pass a byte object as a parameter whenever a byte-sized reference parameter is required). HLA will emit the code to take the address of that one-byte object in memory and pass this address on to the procedure. Note that within the *uns8RefParm* procedure, like any reference parameter, HLA treats b as though it were a dword object (because pointers always require 32 bits, regardless of the size of the object at which they point). Here is a typical call to the *uns8RefParm* procedure:

```
static
   uns8Var :uns8;
      .
      .
      .
   uns8RefParm( uns8Var );
```

Here s the code HLA might generate for this procedure call:

```
   pushd( &uns8Var );
   call uns8RefParm;
```

Now consider the following code fragments:

```
type
   pUns8 : pointer to uns8;

static
   pUns8Var :pUns8;
   uns8Var  :uns8;
      .
      .
      .
   mov( &uns8Var, pUns8Var ); // Initialize pUns8Var with the address of uns8Var
      .
      .
      .
   uns8RefParm( pUns8Var );
```

Where HLA to strictly enforce type checking of reference parameters, this call to *uns8RefParm* would generate an error. After all, this code is trying to pass a four-byte pointer object by reference to a procedure that is expecting an *uns8* reference parameter. The two variables in memory are not at all compatible even though the pointer object, itself, points at an *uns8* variable in memory. In a typical high level language, it makes a lot of sense to report this as an error and reject the source file as uncompilable. In assembly language, however, it is very common to refer to objects (even static ones) using pointers to those objects that you generate on the fly (e.g., pointer values that you load into registers and possibly manipulate in those registers). As a result, you ve often got the address of a variable sitting around in a register or a pointer and you d like to pass that address on through to the procedure rather than taking the address of the object. After all, any assembly language programmer realizes that pass by reference parameters are just syntactical sugar for saying take the address of the actual parameter because the procedure is expecting us to pass it an address. If you ve already got the address somewhere, why bother computing it again?

If the address is sitting in a 32-bit register, you can always pass the address using a memory operand like [ebx] . HLA will, nicely, assume that an untyped memory operand like this actually points at a correct object. So you could call *uns8RefParm* as follows:

```
   uns8RefParm( [ebx] );
```

Theoretically, it would be better for you to write this as

```
   uns8RefParm( (type uns8 [ebx]) );
```

However, experience shows that requiring the type coercion here is just too inconvenient to consider. So HLA relaxes the requirement a bit and allows the use of anonymous memory references (those involving an indirect address in a 32-bit register without any other type or name information) as legal arguments for any pass by reference parameter.

Now consider the following call to *uns8RefParm*:

```
uns8RefParm( pUns8Var );
```

The *pUns8Var* argument should, technically, be illegal. This is not an object of byte *uns8* (or *byte*) and, therefore, HLA should generate a type mismatch error in response to this procedure invocation. As it turns out, however, it is very common in assembly language programs (especially in Win32 assembly programs) to want to pass the pointer to a memory object in a reference parameter slot. One could easily do this by explicitly passing the value of the pointer as the parameter, e.g.,

```
   push( pUns8Var );
   call uns8RefParm;

// or

   uns8RefParm( #{ push( pUns8Var ); }# );
```

However, the need to pass such a pointer variable s value in place of taking the address of some variable occurs so frequently that HLA relaxes the type checking of a reference parameter to allow any pointer variable that points at the base type of the formal parameter s type. That is, it is legal to pass a pointer to an *uns8* variable as a pass by reference parameter whose type is *uns8*. Rather than take the address of the pointer variable, such a procedure invocation would copy the value of that pointer variable (which is the address of some other object) as the actual parameter data, e.g.,

```
   uns8RefParm( uns8Var );  // Legal, of course
   uns8RefParm( pUns8Var ); // Also legal

// The above two statements emit the following code:

   push( &uns8Var );
   call uns8RefParm;

   push( pUns8Var );  // Pushes the value of pUns8Var, not it's address!
   call uns8RefParm;
```

Note that untyped reference parameters always take the address of the actual parameter to pass on to the procedure, even if the actual parameter is a pointer. This is an unfortunately gotcha that you re going to have to remember. Consider the following code as an example of this problem:

```
procedure untypedParm( var v:var );
      .
      .
      .
procedure typedParm( var v:char );
      .
      .
```

```
      .
static
   p :pointer to char;
      .
      .
      .
   typedParm( p ); // Legal, passes value of p to typedParm.
   untypedParm( p ); // Legal passes the *address* of p to untypedParm.

// code equivalent to the above two calls:

   pushd( p );   // Note that this pushes p's value, not its address.
   call typedParm;

   pushd( &p );  // Notice that this pushes the address of p, not p's value.
   call untypedParm;
```

This behavior isn t entirely intuitive. As a result, many programmers who ve gotten in the habit of passing a pointer variable as a reference parameter wind up getting burned when the formal parameter is an untyped reference parameter. This problem crops up enough that it s quite possible that HLA will drop support for automatically passing the value of a pointer variable through a reference parameter in some future version of the compiler. Therefore, you should avoid passing a pointer variable as a reference parameter, expecting HLA to automatically push the value (rather than the address) of the pointer variable onto the stack.

Although you should avoid having HLA automatically pass a pointer variable s value through a reference parameter, the need to pass such a value exists. Recent versions of HLA have made use of HLA s val reserved word as a prefix to an actual parameter to tell HLA to pass that parameter s value as the address the reference parameter requires. To do this, simply prefix the actual parameter with the *val* keyword, e.g.,

```
untypedParm( ptrVar );      // Passes the address of ptrVal
untypedParm( val ptrVar );  // Passes ptrVar's value.
```

The *val* keyword actually works with any *dword* or pointer variable type, not just pointers to the underlying type, e.g.,

```
typedParm( val dwordVar );
```

In HLA, strings are special cases of pointer objects. If you pass a string variable as a parameter that is an untyped reference parameter, HLA will actually pass the address of the string variable (that is, the address of the pointer) rather than the address of the string data (which is where the string is pointing). Several Win32 API functions, for example, expect you to pass them the address of a sequence of characters (i.e., a string) and you ll probably have that data sitting in an HLA string variable in your code. However, if you directly pass the string variable as the parameter to the Win32 API function, HLA will actually take the address of the string pointer variable rather than pass the address of the character data on through to the procedure. You can counteract this behavior by prefixing the string variable with the *val* keyword in the actual procedure call, e.g.,

```
procWithUntypedParm( val strVariable );
```

Because strings are pointers, you may also use the *val* keyword to pass an arbitrary 32-bit numeric value for a string parameter. This is useful in certain Win32 API calls where you pass either a pointer to a zero-terminated sequence of characters (i.e., a string) or a small integer ATOM value, e.g.,

```
proceWithStrParm( val 12345 );  // Normally requires a string parameter.
```

Whenever you're in doubt as to what HLA is doing with your code, you should take a look at the assembly language output file that HLA produces. Although the syntax is different than HLA, you should be able to figure out what code the compiler is generating for your procedure calls. If this isn't the code you want, you can always pass the parameters manually or apply some operator like `val` to an actual parameter.

2.5: The HLA Compile-Time Language

One of HLA's more powerful features, and a feature we'll use quite a bit in this book, is the HLA *compile-time language*. The concept of a compile-time language may seem pretty weird when you first consider it, but the truth is this term is really just a new label for something that's been around a long time: a built-in macro processor. The main reason HLA calls its macro processor a compile-time language is because HLA's macro facilities are quite a bit more powerful than those available with most assemblers.

Actually, to equate the term "compile-time language" with "macro processor" is a bit of an injustice. A compile-time language includes several things that many people often think of as being separate from macro processing, including conditional assembly/compilation, string processing (e.g., of macro operands), repetitive code emission, and so on. Most languages that provide compile-time language features provide a minimal set of such constructs in the overall language. Not so with HLA. It's actually possible to write complete applications using nothing more than the HLA compile-time language; such usage isn't entirely appropriate or efficient, but it is possible to create simple applications within HLA's compile-time language facility.

The real purpose of the HLA compile-time language is to automate the creation of certain common code sequences. A simple macro is a good, well-understood, example of this. By invoking a macro, you can expand some sequence of machine instructions over and over again in an assembly language file, thus "automating" the creation of the code sequences those macros produce. Macros aren't the only constructs that can automate the generation of code at compile-time, however. HLA also provides loops, declarations, assignments, I/O, and other facilities we can use to automate code generation.

Before explaining how to use HLA's compile-time language, perhaps it's a good idea to explicitly define what a compile-time language and a run-time language are and how you would use them. The run-time language is the language that you use to create programs that run after compilation completes; in the case of HLA, the run-time language is 80x86 assembly language and a run-time program is the 80x86 machine code resulting from an HLA compilation. The run-time language includes all the 80x86 machine instructions, data declaration statements, procedure declarations, and so on). It is not physically possible for the run-time program to execute concurrently with its production by the HLA compiler (obviously, you can run a previous instance of that program while HLA is running, but we're talking about the machine code that HLA produces during compilation here). The compile-time language, on the other hand, consists of a handful of statements that HLA executes, via interpretation, while it is compiling the run-time program. The two languages are not entirely distinct. For example, constant and value declarations in the HLA declaration section are usable by both the run-time and compile-time programs. Despite the overlap, it's important to realize that the two languages are distinct and the presence of compile-time language statements in an HLA source file may not produce equivalent execution semantics in a run-time program.

The best place to start when discussing the compile-time language is with the HLA declaration section. The compile-time and run-time languages share the HLA declaration sections (though the two languages sometimes interpret the declarations differently). So let's look at each of the declaration sections that appear in an HLA source file and compare their effects on both the compile-time and run-time programs in that source file.

The HLA *const* declaration section defines constant values in both the compile-time and run-time languages. Any declaration appearing in an HLA *const* section is available in both languages in the same source file (subject, of course, to the scoping rules that apply to that identifier). The only issue is that HLA *text* constants are only usable at compile-time.

The HLA *val* declaration section also declares constants as far as the run-time language is concerned. However, this is where you declare compile-time *variables*. The difference between HLA *const* values and HLA *val* values is the fact that (during assembly) you can change the value of a *val* object. Hence, at compile-time, *val* objects are variables. Note that the HLA compile-time ? (assignment) statement also creates *val* objects that the run-time program can treat as a constant value during the application s execution. We ll return to this statement in a moment.

The HLA type declaration section is also applicable to both compile-time and run-time programs. That is, you can define types that are usable by both compile-time and run-time programs in this declaration section. Note, however, that there are a couple of types that don t make sense at compile-time. Specifically you cannot create compile-time *class* objects, *thunks*, and a few other types. As noted for *const* and *val* objects, if you define a new type based on HLA s built-in *text* type, that type is only usable in a compile-time program.

The *static, readonly, storage,* and *var* sections create run-time variables. HLA s compile-time language provides compile-time function that let you test the compile-time attributes of these variable declarations (e.g., determine their type, the number of array elements, stuff like that), but the compile-time program cannot access the run-time values of these variables (because run-time occurs long after the compile-time program terminates). For *static* and *readonly* objects the compile-time language can specify the initial value loaded into memory for these run-time variables, but the compile-time language really has no idea what value a run-time variable will have at a given point during program execution.

HLA procedures, methods, and iterators are purely run-time entities. Like run-time variables, the HLA compile-time language can test certain *procedure/iterator/method* name attributes, but the compile-time language cannot call this subroutines nor determine much else about them.

An HLA namespace is an encapsulation of a declaration section. Namespaces may encapsulate run-time as well as compile-time declarations, so namespaces are equally applicable to both types of programs. Accessing fields of the namespace is done the same way in both compile-time and run-time programs (e.g., using dot-notation).

HLA classes are mainly a run-time construct. However, classes, like namespaces, open up a new declaration section in which you can declare constants, values, and variables. So although you cannot create *class* constants at compile-time, a *class* may contain several compile-time language elements (e.g., *const, val,* and macro declarations). Though this book use classes and objects in a few sample programs, such usage is going to be rare enough that we can effectively ignore classes in our discussion here.

So what does a compile-time program look like, anyway? That s a good question and there isn t really a specific definition of a compile-time program. Some might consider an entire HLA source file containing compile-time statements to be a single compile-time program. Other programmers might consider a short sequence of compile-time statements in a source file to be a single compile-time program (so a single HLA source file could contain several compile-time program-ettes). HLA doesn t really define what constitutes a single compile-time program, so we ll just work on the assumption that any sequence of compile-time statements used to compute some compile-time value or create a single run-time sequence of statements constitutes a single compile-time program (i.e., there can be multiple compile-time programs in a single HLA source file). Well, enough of the philosophy, let s look at the statements that are available to HLA compile-time language programmers.

2.5.1: Compile-Time Assignment Statements

One of the more important compile-time statements is the compile-time assignment statement. This statement takes one of these forms:

```
? compile_time_variable := compile_time_expression ;
? compile_time_variable : type_id := compile_time_expression ;
? compile_time_variable : type_id;
? compile_time_array_variable[ dimension_list ] := compile_time_expression ;
? @built_in_variable := compile_time_expression ;
```

Here are some examples:

```
? i := 5;
? b :boolean := true;
? a :int32[ 4];  // Default initialization is all zeros.
? a[ 2] := 5;
? @nodisplay := true;
```

Note that the compile-time assignment statement is almost semantically identical to a *val* section declaration. In fact, the internal HLA code is almost identical for handling the *val* section versus the ? statement. The only semantic difference between the two has to do with identifier scope. Consider the following code fragment:

```
val
   i :int32;
   j :uns32;

   procedure nestMe;
   val
      i :real := 3.0; // Creates a new "i" in this procedure's scope.

      i := 2.2;       // Resets the local i's value to 2.2, does not affect value of i
                      //  in the outer scope.
         .
         .
         .
      ?j := 5;        // Stores 5 into the outer scope's "j".
         .
         .
         .
   end nestMe;
```

As you can see in this example, a *val* declaration will always create a new instance of a compile-time variable if that variable does not already exist in the current scope, then HLA will create a new variable in the current scope (though if the compile-time variable does exist in the current scope, then HLA will use that variable). The compile-time assignment operator will always attempt to use a predefined version of the compile-time variable, even if that variable was declared in an outer scope. The compile-time assignment statement will only create a new compile-time variable if it is not visible at all at the point in the program where the assignment occurs.

HLA compile-time variables use a *dynamic typed system*. This means that you can (usually) change the type of a compile-time variable after you ve declared it as some other type. E.g.,

```
? i:int32;  // Declare i as an int32 compile-time object
    .
    .   // compile-time code that uses i as an int32 compile-time variable.
    .
? i :string := "i is now a string"; //From this point forward, i is a string object.
```

The one point we ve not really discussed here is exactly what is legal for an expression in an HLA compile-time assignment statement. The truth is, HLA provides a very sophisticated expression syntax that is very similar to what you would find in a high level language like C/C++ or Pascal. In fact, with all the built-in compile-time functions, one could easily argue that HLAs compile-time language is even more sophisticated that what you ll find in many high level languages. The bottom line is that HLA constant expressions are a bit too complex to do justice here; you ll have to read the HLA Reference Manual to get the full picture. Nevertheless, it s worth listing the arithmetic and relational operators because you ll frequently use them in HLA compile-time programs.

```
! (unary not), - (unary negation)
*, div, mod, /, <<, >>
+, -
=, = =, <>, !=, <=, >=, <, >
&, |, &, in
```

One of HLAs more powerful features is the ability to support structured constants in the compile-time language. Because we ll be using structured constants throughout this book, it s worthwhile to review character set, record, and union constants.

A character set literal constant consists of several comma delimited character set expressions within a pair of braces. The character set expressions can either be individual character values or a pair of character values separated by an ellipse (..). If an individual character expression appears within the character set, then that character is a member of the set; if a pair of character expressions, separated by an ellipse, appears within a character set literal, then all characters between the first such expression and the second expression are members of the set.

Examples:

```
{ 'a','b','c'}              // a, b, and c.
{ 'a'..'c'}                 // a, b, and c.
{ 'A'..'Z','a'..'z'}        //Alphabetic characters.
{ ' ',#$d,#$a,#$9}          //Whitespace (space, return, linefeed, tab).
```

HLA character sets are currently limited to holding characters from the 128-character ASCII character set.

HLA lets you specify an array literal constant by enclosing a set of values within a pair of square brackets. Since array elements must be homogenous, all elements in an array literal constant must be the same type or conformable to the same type. Examples:

```
[ 1, 2, 3, 4, 9, 17 ]
[ 'a', 'A', 'b', 'B' ]
[ "hello", "world" ]
```

Note that each item in the list of values can actually be a constant expression, not a simple literal value.

HLA array constants are always one dimensional. This, however, is not a limitation because if you attempt to use array constants in a constant expression, the only thing that HLA checks is the total number of elements. Therefore, an array constant with eight integers can be assigned to any of the following arrays:

```
const
   a8:       int32[ 8]        := [ 1,2,3,4,5,6,7,8];
   a2x4:     int32[ 2,4]      := [ 1,2,3,4,5,6,7,8];
   a2x2x2:   int32[ 2,2,2]    := [ 1,2,3,4,5,6,7,8];
```

Although HLA doesn t support the notation of a multi-dimensional array constant, HLA does allow you to include an array constant as one of the elements in an array constant. If an array constant appears as a list item within some other array constant, then HLA expands the interior constant in place, lengthening the list of items in the enclosing list. E.g., the following three array constants are equivalent:

```
[ [ 1,2,3,4], [ 5,6,7,8] ]
[ [ [ 1,2], [ 3,4] ], [ [ 5,6], [ 7,8] ] ]
[ 1,2,3,4,5,6,7,8]
```

Although the three array constants are identical, as far as HLA is concerned, you might want to use these three different forms to suggest the shape of the array in an actual declaration, e.g.,

```
const
   a8:       int32[ 8]        := [ 1,2,3,4,5,6,7,8];
   a2x4:     int32[ 2,4]      := [ [ 1,2,3,4], [ 5,6,7,8] ];
   a2x2x2:   int32[ 2,2,2]    := [[[ 1,2], [ 3,4] ], [[ 5,6], [ 7,8]]];
```

Also note that symbol array constants, not just literal array constants, may appear in a literal array constant. For example, the following literal array constant creates a nine-element array holding the values one through nine at indexes zero through eight:

```
const Nine: int32[  9 ]      := [  a8, 9 ];
```

This assumes, of course, that a8 was previously declared as above. Since HLA flattens all array constants, you could have substituted a2x4 or ax2x2x for a8 in the example above and obtained identical results.

You may also create an array constant using the HLA *dup* operator. This operator uses the following syntax:

expression dup [*expression_to_replicate*]

Where *expression* is an integer expression and *expression_to_replicate* is a some expression, possibly an array constant. HLA generates an array constant by repeating the values in the *expression_to_replicate* the number of times specified by the expression. (Note: this does not create an array with *expression* elements unless the *expression_to_replicate* contains only a single value; it creates an array whose element count is expression times the number of items in the *expression_to_replicate*). Examples:

```
10 dup [ 1]    -- equivalent to [ 1,1,1,1,1,1,1,1,1,1]
5 dup [ 1,2]   -- equivalent to [ 1,2,1,2,1,2,1,2,1,2]
```

Please note that HLA does not allow class constants, so class objects may not appear in array constants.

HLA supports record constants using a syntax very similar to array constants. You enclose a comma-separated list of values for each field in a pair of square brackets. To further differentiate array and record constants, the name of the record type and a colon must precede the opening square bracket, e.g.,

```
type
   Planet:
```

```
    record
       x:          int32;
       y:          int32;
       z:          int32;
       density:    real64;
    endrecord;


const
   somePlanet : Planet := Planet:[  1, 12, 34, 1.96 ]
```

HLA associates the items in the list with the fields as they appear in the original record declaration. In this example, the values 1, 12, 34, and 1.96 are associated with fields *x, y, z,* and *density*, respectively. Of course, the types of the individual constants must match (or be conformable to) the types of the individual fields.

Note that you may not create a record constant for a particular record type if that record includes data types that cannot have compile-time constants associated with them. For example, if a field of a record is a class object, you cannot create a record constant for that type since you cannot create class constants.

Union constants allow you to initialize static union data structures in memory as well as initialize union fields of other data structures (including anonymous union fields in records). There are some important differences between HLA compile-time union constants and HLA run-time unions (as well as between the HLA run-time union constants and unions in other languages). Therefore, it s a good idea to begin the discussion of HLAs union constants with a description of these differences.

There are a couple of different reasons people use unions in a program. The original reason was to share a sequence of memory locations between various fields whose access is mutually exclusive. When using a union in this manner, one never reads the data from a field unless they ve previous written data to that field and there are no intervening writes to other fields between that previous write and the current read. The HLA compile-time language fully (and only) supports this use of union objects.

A second reason people use unions (especially in high level languages) is to provide aliases to a given memory location; particularly, aliases whose data types are different. In this mode, a programmer might write a value to one field and then read that data using a different field (in order to access that data s bit representation as a different type). **HLA does not support this type of access to union constants**. The reason is quite simple: internally, HLA uses a special variant data type to represent all possible constant types. Whenever you create a union constant, HLA lets you provide a value for a single data field. From that point forward, HLA effectively treats the union constant as a scalar object whose type is the same as the field you ve initialized; access to the other fields through the union constant is no longer possible. Therefore, you cannot use HLA compile-time constants to do type coercion; nor is there any need to since HLA provides a set of type coercion operators like *@byte, @word, @dword, @int8,* etc. As noted already, the main purpose for providing HLA union constants is to allow you to initialize static union variables; since you can only store one value into a memory location at a time, union constants only need to be able to represent a single field of the union at one time (of course, at run-time you may access any field of the static union object you ve created; but at compile-time you may only access the single field associated with a union constant).

An HLA literal union constant takes the following form:

```
   typename.fieldname:[  constant_expression ]
```

The *typename* component above must be the name of a previously declared HLA union data type (i.e., a *union* type you ve created in the *type* section). The *fieldname* component must be the name of a field within

that union type. The *constant_expression* component must be a constant value (expression) whose type is the same as, or is automatically coercible to, the type of the *fieldname* field. Here is a complete example:

```
type
   u:union
      b:byte;
      w:word;
      d:dword;
      q:qword;
   endunion;

static
   uVar  :u    := u.w:[ $1234];
```

The declaration for *uVar* initializes the first two bytes of this object in memory with the value $1234. Note that *uVar* is actually eight bytes long; HLA automatically zeros any unused bytes when initializing a static memory object with a union constant.

Note that you may place a literal union constant in records, arrays, and other composite data structures. The following is a simple example of a record constant that has a union as one of its fields:

```
type
   r  :record
         b:byte;
         uf:u;
         d:dword;
   endrecord;

static
   sr :r := r:[ 0, u.d:[ $1234_5678], 12345];
```

In this example, HLA initializes the *sr* variable with the byte value zero, followed by a *dword* containing $1234_5678 and a *dword* containing zero (to pad out the remainder of the union field), followed by a *dword* containing 12345.

You can also create records that have anonymous unions in them and then initialize a record object with a literal constant. Consider the following type declaration with an anonymous union:

```
type
   rau :record
         b:byte;
         union
            c:char;
            d:dword;
         endunion;
         w:word;
   endrecord;
```

Since anonymous unions within a record do not have a type associated with them, you cannot use the standard literal union constant syntax to initialize the anonymous union field (that syntax requires a type name). Instead, HLA offers you two choices when creating a literal record constant with an anonymous union field. The first alternative is to use the reserved word union in place of a typename when creating a literal union constant, e.g.,

```
static
```

```
   srau :rau := rau:[ 1, union.d:[ $12345], $5678 ];
```

The second alternative is a shortcut notation. HLA allows you to simply specify a value that is compatible with the first field of the anonymous union and HLA will assign that value to the first field and ignore any other fields in the union, e.g.,

```
static
   srau2 :rau := rau:[ 1, 'c', $5678 ];
```

This is slightly dangerous since HLA relaxes type checking a bit here, but when creating tables of record constants, this is very convenient if you generally provide values for only a single field of the anonymous union; just make sure that the commonly used field appears first and you re in business.

Although HLA allows anonymous records within a union, there was no syntactically acceptable way to differentiate anonymous record fields from other fields in the union; therefore, HLA does not allow you to create union constants if the union type contains an anonymous record. The easy workaround is to create a named record field and specify the name of the record field when creating a union constant, e.g.,

```
type
   r  :record
         c:char;
         d:dword;
   endrecord;

   u  :union
         b:byte;
         x:r;
         w:word;
   endunion;

static
   y :u := u.x:[ r:[ 'a', 5]];
```

You may declare a union constant and then assign data to the specific fields as you would a record constant. The following example provides some samples of this:

```
type
   u_t :union
         b:byte;
         x:r;
         w:word;
   endunion;

val
   u :u_t;
      .
      .
      .
   ?u.b := 0;
      .
      .
      .
   ?u.w := $1234;
```

The two assignments above are roughly equivalent to the following:

```
?u := u_t.b:[0];
```

and

```
?u := u_t.w:[$1234];
```

However, to use the straight assignment (the former example) you must first declare the value *u* as a *u_t* union.

To access a value of a union constant, you use the familiar dot notation from records and other languages, e.g.,

```
?x := u.b;
   .
   .
   .
?y := u.w & $FF00;
```

Note, however, that you may only access the last field of the union into which you ve stored some value. If you store data into one field and attempt to read the data from some other field of the union, HLA will report an error. Remember, you don t use union constants as a sneaky way to coerce one value s type to another (use the coercion functions for that purpose).

HLA allows a very limited form of a pointer constant. If you place an ampersand (&) in front of a static object s name (i.e., the name of a static variable, readonly variable, uninitialized variable, segment variable, procedure, method, or iterator), HLA will compute the run-time offset of that variable. Pointer constants may not be used in arbitrary constant expressions. You may only use pointer constants in expressions used to initialize static or readonly variables or as constant expressions in 80x86 instructions. The following example demonstrates how pointer constants can be used:

```
program pointerConstDemo;

static
   t:int32;
   pt: pointer to int32 := &t;

begin pointerConstDemo;

   mov( &t, eax );

end pointerConstDemo;
```

Also note that HLA allows the use of the reserved word ***NULL*** anywhere a pointer constant is legal. HLA substitutes the value zero for ***NULL***.

You may assign any of these structured literal constants to a compile-time variable or constant using an HLA compile-time assignment statement.

2.5.2: Compile-Time Functions

HLA provides the ability to create your own compile-time functions using macros, and we ll discuss how to do that in a little bit. In this section, however, we ll take a look at a small sample of the built-in compile-time functions that HLA automatically provides for you. HLA actually provides a large number of compile-time functions, too many to present here, so in this section we ll take a look at the ones that Win32 programmers commonly use. Please see the HLA Reference manual for more details on the HLA compile-time function facilities.

Though it is perfectly possible to write an assembly language program without ever calling an HLA compile-time function (or using the compile-time language at all, for that matter), the HLA compile-time functions can make it easier to develop certain macros and they can help make your programs easier to write and maintain. Therefore, it s a good idea to familiarize yourself with the HLA compile-time functions.

Note: don t confuse HLA s compile-time functions with the functions available in the HLA Standard Library. Although there are many similarities, Standard Library functions are run-time functions, compile-time functions execute during compilation.

The first set of compile-time functions to look at are the type-conversion functions. These compile-time function convert a constant expression you pass as an argument to the corresponding data type (or create a compile-time error if the conversion is not possible). These conversion functions are unusual insofar as they are the only set of HLA compile-time functions that don t begin with @ (this functions just use the built-in HLA type names).

```
boolean( const_expr )
```

The expression must be an ordinal or string expression. If *const_expr* is numeric, this function returns false for zero and true for everything else. If *const_expr* is a character, this function returns true for T and false for F . It generates an error for any other character value. If *const_expr* is a string, the string must contain true or false else HLA generates an error.

```
int8( const_expr ), int16( const_expr ), int32( const_expr )
int64( const_expr ), int128( const_expr )
uns8( const_expr ), uns16 const_expr ), uns32( const_expr )
uns64( const_expr ), uns128( const_expr )
byte( const_expr ), word( const_expr ), dword( const_expr )
qword( const_expr ), lword( const_expr )
```

These functions convert their parameter to the specified integer. For real operands, the result is truncated to form a numeric operand. For all other numeric operands, the result is ranged checked. For character operands, the ASCII code of the specified character is returned. For *boolean* objects, zero or one is returned. For *string* operands, the string must be a sequence of decimal characters which are converted to the specified type. Note that *byte, word,* and *dword* types are synonymous with *uns8, uns16,* and *uns32* for the purposes of range checking.

```
real32( const_expr ), real64( const_expr ), real80( const_expr )
```

Similar to the integer functions above, except these functions produce the obvious real results. Only numeric and string parameters are legal.

```
char( const_expr )
```

Const_expr must be a ordinal or string value. This function returns a character whose ASCII code is that ordinal value. For strings, this function returns the first character of the string.

```
string( const_expr )
```

This function produces a reasonable string representation of the parameter. Almost all data types are legal.

```
cset( const_expr )
```

The parameter must be a character, string, or cset. For character parameters, this function returns the singleton set containing only the specified character. For strings, each character in the string is unioned into the set and the function returns the result. If the parameter is a cset, this function makes a copy of that character set.

The type conversion functions just described will automatically convert their operands from the source type to the destination type. Sometimes you might want to change the type of some object without changing its value. For many conversions this is exactly what takes place. For example, when converting and *uns8* object to an *uns16* value using the *uns16(---)* function, HLA does not modify the bit pattern at all. For other conversions, however, HLA may completely change the underlying bit pattern when doing the conversion. For example, when converting the *real32* value 1.0 to a *dword* value, HLA completely changes the underlying bit pattern ($3F80_0000) so that the *dword* value is equal to one. On occasion, however, you might actually want to copy the bits straight across so that the resulting *dword* value is $3F80_0000. The HLA bit-transfer type conversion compile-time functions provide this facility.

The HLA bit-transfer type conversion functions are the following:

```
@int8( const_expr ), @int16( const_expr ), @int32( const_expr )
@int64( const_expr ), @int128( const_expr )
@uns8( const_expr ), @uns16 const_expr ), @uns32( const_expr )
@uns64( const_expr ), @uns128( const_expr )
@byte( const_expr ), @word( const_expr ), @dword( const_expr )
@qword( const_expr ), @lword( const_expr )
@real32( const_expr ), @real64( const_expr ), @real80( const_expr )
@char( const_expr )
@cset( const_expr )
```

The above functions extract eight, 16, 32, 64, or 128 bits from the constant expression for use as the value of the function. Note that supplying a string expression as an argument isn t particularly useful since the functions above will simply return the address of the string data in memory while HLA is compiling the program. The *@byte* function provides an additional syntax with two parameters, we ll describe that in a moment.

```
@string( const_expr )
```

HLA string objects are pointers (in both the language as well as within the compiler). So simply copying the bits to the internal string object would create problems since the bit pattern probably is not a valid pointer to string data during the compilation. With just a few exceptions, what the *@string* function does is takes the bit data of its argument and translates this to a string (up to 16 characters long). Note that the actual string may be between zero and 16 characters long since the HLA compiler (internally) uses zero-terminated strings to represent string constants. Note that the first zero byte found in the argument will end the string.

If you supply a *string* expression as an argument to *@string,* HLA simply returns the value of the string argument as the value for the *@string* function. If you supply a *text* object as an argument to the *@string* function, HLA returns the text data as a string without first expanding the text value (similar to the *@string:identifier* token). If you supply a pointer constant as an argument to the *@string* function, HLA returns the string that HLA will substitute for the static object when it emits the assembly file.

HLA provides a fair set of functions that let you determine attributes of various symbols and expressions during compilation. Here are some of these compile-time functions you ll commonly use:

```
]@name( identifier )
```

This function returns a string of characters that corresponds to the name of the identifier (note: after text/macro expansion). This is useful inside macros when attempting to determine the name of a macro parameter variable (e.g., for error messages, etc.). This function returns the empty string if the parameter is not an identifier.

`@typename( identifier_or_expression )`

This function returns the string name of the type of the identifier or constant expression. Examples include int32 , boolean , and real80 .

`@size( identifier_or_expression )`

This function returns the size, in bytes, of the specified object.

`@elementsize( identifier_or_expression )`

This function returns the size, in bytes, of an element of the specified array. If the parameter is not an array identifier, this function generates an assembly-time error.

`@offset( identifier )`

For VAR, PARM, METHOD, and class ITERATOR objects only, this function returns the integer offset into the activation record (or object record) of the specified symbol.

`@Is External( identifier )`

This function returns true if the specified identifier is an external symbol.

`@arity( identifier_or_expression )`

This function returns zero if the specified identifier is not an array. Otherwise it returns the number of dimension of that array.

`@dim( array_identifier_or_expression )`

This function returns a single array of integers with one element for each dimension of the array passed as a parameter. Each element of the array returned by this function gives the number of elements in the specified dimension. For example, given the following code:

```
val threeD: int32[ 2, 4, 6];
   tdDims:= @dim( threeD );
```

The *tdDims* constant would be an array with the three elements [2, 4, 6];

`@elements( array_identifier_or_expression )`

This function returns the total number of elements in the specified array. For multi-dimensional array constants, this function returns the number of all elements, not just a particular row or column.

`@defined( identifier )`

This function returns true if the specified identifier is has been previously defined in the program and is currently in scope.

`@isconst( expr ), @isreg( expr ), @isreg8( expr ), @isreg16( expr ), @isreg32( expr )`
`@isfreg( expr ),@ismem( expr ), @istype( identifier )`

These functions return true if their argument is a constant, register, memory address, or type ID. These functions are useful in classifying macro parameters. This is not a complete list of the HLA compile-time classification functions, please consult the HLA Reference Manual to see the complete set.

`@linenumber`

This function returns the current line number in the source file.

`@filename`

This function returns the name of the current source file.

HLA provides several general purpose numeric functions that are great for initializing tables and doing other compile-time calculations. Here are some of these functions:

`@abs( numeric_expr )`

Returns the absolute equivalent of the numeric value passed as a parameter.

`@byte( integer_expr, which ), @byte( real32_expr, which ), @byte( real64_expr, which )`
`@byte( real80_expr, which )`

The @byte function extracts a single byte from a multi-byte data type. For integer/ordinal expressions, the *which* parameter is a value in the range 0..15. For *real32* operands, the *which* parameter is a value in the range 0..3. This function extracts the specified byte from the value of the *real32_expression* parameter. For *real64* operands, the *which* parameter is a value in the range 0..7. For real80 operands, the *which* parameter is a value in the range 0..9. These functions extract the specified byte from the value of their second operand.

`@ceil( real_expr ), @floor( real_expr )`

The *@ceil* function returns the smallest integer value larger than or equal to the expression passed as a parameter. The *@floor* function returns the largest integer value less than or equal to the supplied expression. Note that although the result will be an integer, these functions return a *real80* value.

`@cos( real_expr ), @sin( real_expr ), @tan( real_expr )`

The real parameter is an angle in radians. These functions return the usual trigonometric values for the parameter you pass them.

`@date`

This function returns a string of the form YYYY/MM/DD containing the current date.

`@exp( real_expr )`

This function returns a *real80* value that is the result of the computation e^{real_expr}.

`@extract( cset_expr )`

This function returns a character from the specified character set constant. Note that this function doesn t actually remove the character from the set, if you want to do that, then you will need to explicitly remove the character yourself.

`@isalpha( char_expr ), @isalphanum( char_expr ), @isdigit( char_expr )`
`@islower( char_expr ), @isspace( char_expr ), @isupper( char_expr )`
`@isxdigit( char_expr )`

These predicate functions return true or false based on whether there arguments belong to a certain set of characters. Note that isxdigit returns true if its argument is a hexadecimal character.

`@log( real_expr ), @log10( real_expr )`

@log returns the natural (base-e) logarithm of its argument, @log10 returns the base-10 logarithm of the supplied parameter.

`@max( comma_separated_list_of_ordinal_or_real_values ),`
`@min( comma_separated_list_of_ordinal_or_real_values )`

These functions return the maximum or minimum of the values in the specified list.

`@odd( int_expr )`

This function returns true if the integer expression is an odd number.

`@random( int_expr )`

This function returns a random uns32 value.

`@randomize( int_expr )`

This function uses the integer expression passed as a parameter as the new seed value for the random number generator.

`@sqrt( real_expr )`

This function returns the square root of the parameter.

`@time`

This function returns a string of the form HH:MM:SS xM (x= A or P) denoting the time at the point this function was called (according to the system clock).

HLA provides a large number of compile-time string functions. These functions are especially useful for processing macro parameters and other string constants in an HLA compile-time program. Here are some of the compile-time string functions that HLA provides (see the *HLA Reference Manual* for the complete list):

`@delete( str_expr, int_start, int_len )`

This function returns a string consisting of the *str_expr* passed as a parameter with (possibly) some characters removed. This function removes *int_len* characters from the string starting at index *int_start* (note that strings have a starting index of zero).

`@index( str_expr1, int_start, str_expr2 )`

This function searches for *str_expr2* within *str_expr1* starting at character position *int_start* within *str_expr1*. If the string is found, this function returns the index into *str1_expr1* of the first match (starting at *int_start*). This function returns -1 if there is no match.

`@insert( str_expr1, int_start, str_expr2 ), @index( str_expr1, int_start, str_expr2 )`

The insert function inserts *str_expr2* into *str_expr1* just before the character at index *int_start*. The *rindex* function does the same, except *int_start* is from the end of the string rather than the start of the string. These functions return the converted string (note that these functions do not modify the original string).

`@length( str_expr )`

This function returns the length of the specified string.

`@lowercase( str_expr, int_start ), @uppercase( str_expr, int_start )`

These functions return a string of characters from *str_expr* with all uppercase alphabetic characters converted to lower case or vice versa. Only those characters from *int_start* on are copied into the result string.

`@substr( str_expr, int_start, int_len )`

This function returns the substring specified by the starting position and length in *str_expr*.

HLA also provides a large number of string/pattern matching functions. Because of the large number of these functions, we ll not regurgitate their description here. Please see the HLA Reference Manual for details on the compile-time pattern matching functions.

HLA provides a large number of additional compile-time functions and pseudo-variables that you may find useful when creating compile-time programs. Space limitations prevent the complete exposition of these functions here. This chapter just high-lights some of the more common functions in order to give you a taste of what is possible when using the HLA compile-time language. As a Win32 programmer, you might not use all of the functions this chapter describes and you might also use several of HLA s compile-time functions that this chapter does not describe. The set of functions you might use in an application are as varied as the number of applications that have been written.

2.5.3: Generating Code With a Compile-Time Statement

Up to this point, the HLA compile-time statements and functions we ve seen generate constant results. They re useful for embedded within machine instructions or in data declaration statements (e.g., to initialize tables); for example mov(@length(someString), eax); automatically substitutes the length of a compile-time string constant as the source operand of this instruction, the program reflects any changes made to the string by adjusting the value of the source operand each time you recompile the program. However, we haven t looked at how you can actually select which statements in your program to compile or how to iteratively process statements (that is, inject multiple copies of a statement into a program).

Why would you want to do this? Well, consider a data table with 2,000 elements where each entry of the table is initialized with it s index plus the sum of the previous two elements (assuming elements before zero contain the value zero). You could manually create this table in HLA as follows:

```
uns32 0, 1, 3, 7, 14, 26, ...
```

Of course, filling in a table like this one, with 2,000 elements, is laborious. Furthermore, chances are pretty good you d make a mistake somewhere along the line (quick, can you spot any mistakes in this example?). Fortunately, generating a table like this one is very easy to do by writing a short HLA compile-time program. Writing such HLA compile time programs can save you considerable effort, help you write correct code, and make it easier to verify the correctness of your programs. So let s take a look at some of the statements in the HLA compile-time language that allow you to do this.

2.5.4: Conditional Assembly (#if..#elseif..#else..#endif)

HLA s compile-time language provides an if statement that allows you to decide, at compile-time, whether certain statements will be present in the actual object code. Many languages (including various assemblers) provide this facility and they call it conditional assembly or conditional compilation . The conditional compilation statements in HLA use the following syntax:

```
#if( constant_boolean_expression )

   << Statements to compile if the >>
   << expression above is true.    >>

#elseif( constant_boolean_expression )

   << Statements to compile if the >>
   << expression immediately above >>
   << is true and the first expres->>
   << sion above is false.         >>

#else

   << Statements to compile if both    >>
   << the expressions above are false. >>

#endif
```

The *#elseif* and *#ELSE* clauses are optional. Like the HLA run-time *elseif* clause, there may be more than one *#elseif* clause in the same conditional *#if* sequence.

Unlike some other assemblers and high level languages, HLA's conditional compilation directives are legal anywhere whitespace is legal. You could even embed them in the middle of an instruction! While directly embedding these directives in an instruction isn't recommended (because it would make your code very hard to read), it's nice to know that you can place these directives in a macro and then replace an instruction operand with a macro invocation.

The constant expression in the *#if* and *#elseif* clauses must be of type boolean or HLA will emit an error. Any legal compile-time expression that produces a boolean result is legal here. Keep in mind that conditional compilation directives are executed at compile-time, not at run-time. You would not use these directives to (attempt to) make decisions while your program is actually running.

Programmers use conditional assembly for a variety of purposes in HLA programs. A very common use of conditional assembly is to compile different code sequences based on the environment in which the program is going to run. For example, some programmers use conditional assembly to embed non-portable code sequences into their programs and then they select between various code sequences by some boolean expression. For example, HLA programmers who are writing code that is to be portable between Linux and Windows might use conditional compilation sequences like the following:

```
#if( os.win32 ) // os.win32 and os.linux are boolean constants provided by the
                // HLA standard library that specifies the OS you're using.

   << code that is Windows specific >>

#elseif( os.linux )

   << code that is Linux specific >>

#endif
```

Of course, this book isn't dealing with code that could run under Windows or Linux, so this feature is probably of little use to the average reader of this book. However, this same idea is useful for writing code that makes advantage of features available only in certain versions of Windows. For example, if you want to make an API call that

is available only in Windows XP or later, but you still want to be able to create a version of your application that runs under earlier versions of Windows, you can use this same trick, e.g.,

```
#if( WinXPorLater )

   << Code that uses WinXP-specific API functions >>

#else

   << code that deals with the fact that this API is missing >>

#endif
```

In this example, *WinXPorLater* is a compile-time boolean value that you ve defined in your program. A programmer will typically set this variable true or false depending upon whether they want to compile a version of their program for WinXP or an earlier version of Windows.

A traditional trick, inherited from the C/C++ programming community, is to test to see whether a symbol is defined or not defined to determine if code should be assembled one way or another. This technique is easy to use in HLA by employing the *@defined* compile-time variable, e.g.,

```
#if( @defined( WinXPorLater ))

   << Code that uses WinXP-specific API functions >>

#else

   << code that deals with the fact that this API is missing >>

#endif
```

Note that *WinXPorLater's* type and value are irrelevant. In fact, *WinXPorLater* doesn t even have to be a constant or compile-time variable (i.e., you can use any legal HLA identifier here, the *@defined* function returns true if the symbol is defined, false if it is not defined, and it completely ignores the other attributes the symbol may possess). HLA programmers who commonly use *@defined* in this manner generally define such symbols near the beginning of their programs and, although the type and value is irrelevant, they tend to define such compile-time variables as boolean with the value true.

Specifically to support the *@defined* compile-time function, the HLA provides a compile-time option that lets you define symbols. This command option takes the following form:

-d*SymbolToDefine*

This creates an HLA *const* definition for *SymbolToDefine*, setting the type to *boolean* and the value to false.

Here s a suggestion if you re going to use HLA s command-line feature to support compilation options via conditional assembly. Rather than use *@defined* all over your code, just put a short compile-time sequence like the following near the beginning of your source file:

```
#if( !@defined( SymbolToDefine ))
   ?SymbolToDefine := false;
#endif
```

This sequence guarantees that *SymbolToDefine* is defined in your source file. It's value (true or false) determines how conditional assembly will take place later on by simply using the *SymbolToDefine* symbol as a boolean expression in your code. The advantage of using this technique, rather than just using *@defined*, is that you can turn the feature on and off throughout the source code by injecting compile-time assignments like ?SymbolToDefine := true; and SymbolToDefine := false; into your source file.

Of course, conditional assembly has far more uses than simply letting you incorporate different program options in the source file that you can control via simple boolean expressions. Conditional assembly, of course, is what you use to make decisions in an HLA compile-time program. The examples thus far have used conditional assembly to select between one of several sequences of instructions to compile into the final run-time program. In fact, many HLA *#if..#endif* sequences don't contain any code emitting statements at all, instead they just contain sequence of other HLA compile-time language statements. Combined with HLA's compile-time functions, it's quite possible to do some very sophisticated processing. Here's a simple example that might be used to prevent problems when generating a table of log values:

```
#if( r > 0 )
   ?logVal := @log( r );
#else
   #error( "Illegal value passed to @log" )
#endif
```

Here's another example that you might find in a macro that tests a parameter's type to determine how to process the parameter:

```
#if( @typename( macroParm ) = "int32" )
   << do something if the macro parameter is a 32-bit integer >>
#elseif( @typename( macroParm ) = "int16" )
   << do something if the macro parameter is a 16-bit integer >>
#elseif( @typename( macroParm ) = "int88" )
   << do something if the macro parameter is an 8-bit integer >>
#else
   #error( "Expected an int32, int16, or int8 parameter" )
#endif
```

To see some complex examples of HLA's *#if..#elseif..#else..#endif* statement in action, check out the source code to many of the HLA Standard Library routines (especially in several of the HLA Standard Library header files). The HLA Standard Library stdout.put macro is an extreme example that you might want to take a look at.

2.5.5: The #for..#endfor Compile-Time Loop

The *#for..#endfor* loop can take one of the following forms:

```
#for( loop_control_var := Start_expr to end_expr )

   << Statements to execute as long as the loop control variable's >>
   << value is less than or equal to the ending expression.        >>

#endfor

#for( loop_control_var := Start_expr downto end_expr )
```

```
   << Statements to execute as long as the loop control variable's >>
   << value is greater than or equal to the ending expression.     >>

#endfor
```

The HLA compile-time *#for..#endfor* statement is very similar to the for loops found in languages like Pascal and BASIC. This is a definite loop that executes some number of times determine when HLA first encounters the *#for* directive (this can be zero or more times, but HLA computes the number only once when HLA first encounters the *#for*). The loop control variable must be a *val* object or an undefined identifier (in which case, HLA will create a new *val* object with the specified name). Also, the number control variable must be an eight, sixteen, or thirty-two bit integer value (*uns8, uns16, uns32, int8, int16,* or *int32*). Also, the starting and ending expressions must be values that an *int32 val* object can hold.

The *#for* loop with the *to* clause initializes the loop control variable with the starting value and repeats the loop as long as the loop control variable's value is less than or equal to the ending expression's value. The *#for..to..#endfor* loop increments the loop control variable on each iteration of the loop.

The *#for* loop with the *downto* clause initializes the loop control variable with the starting value and repeats the loop as long as the loop control variable's value is greater than or equal to the ending expression's value. The *#for..downto..#endfor* loop decrements the loop control variable on each iteration of the loop.

Note that the *#for..to/downto..#endfor* loop only computes the value of the ending expression once, when HLA first encounters the *#for* statement. If the components of this expression would change as a result of the execution of the *#for* loop's body, this will not affect the number of loop iterations. If you need this capability, you will need to use HLA's compile-time indefinite loop (the *#while* loop, see the next section).

The *#for..#endfor* loop can also take the following form:

```
#for( loop_control_var in composite_expr )

   << Statements to execute for each element present in the expression >>

#endfor
```

The *composite_expr* in this syntactical form may be a string, a character set, an array, or a record constant.

This particular form of the *#for* loop repeats once for each item that is a member of the composite expression. For strings, the loop repeats once for each character in the string and the loop control variable is set to each successive character in the string. For character sets, the loop repeats for each character that is a member of the set; the loop control variable is assigned the value of each character found in the set (you should assume that the extraction of characters from the set is arbitrary, even though the current implementation extracts them in order of their ASCII codes). For arrays, this *#for* loop variant repeats for each element of the array and assigns each successive array element to the loop control variable. For record constants, the *#for* loop extracts each field and assigns the fields, in turn, to the loop control variable.

Examples:

```
   #for( c in "Hello" )
      #print( c )  // Prints the five characters 'H', 'e', ..., 'o'
   #endfor

   // The following prints a..z and 0..9 (not necessarily in that order):

   #for( c in {'a'..'z', '0'..'9'} )
      #print( c )
```

```
   #endfor

   // The following prints 1, 10, 100, 1000

   #for( i in [ 1, 10, 100, 1000] )
      #print( i )
   #endfor

   // The following prints all the fields of the record type r
   // (presumably, r is a record type you've defined elsewhere):

   #for( rv in r:[ 0, 'a', "Hello", 3.14159] )
      #print( rv )
   #endfor
```

The HLA compile-time *#for* loop is really useful for processing variable parameter lists in a macro. HLA creates an array of strings with each array element containing the text for each macro parameter. By using this latter form of the *#for* loop you can process each element of the macro in order. You'll see this feature used when we discuss HLA's macro facilities in a few sections.

One very useful purpose for the *#for..#endfor* loop is to construct data tables at compile time. For example, suppose you want to create a table of sine values for each of the angles in the range 0..359 degrees. You could easily do this in HLA as follows:

```
static
   sineTable: real32[ 360 ]; @nostorage;
      #for( angle := 0 to 359 )

         // Note: the HLA compile-time @sin function requires an angle in radians.
         // Conversion from degrees to radians is via the formula:
         // radians = angle * 2 * pi / 360

         real32 @sin( angle * 3.14159 / 180.0 );

      #endfor
```

HLA's *#for..#endfor* compile-time statement is sufficiently powerful that you can generate almost any type of data table you need at compile-time without having to enter in the data for the entire table yourself (or compute the table entries at run-time, consuming both space for the machine instructions and time to execute those instructions). For the few cases where the *#for* loop isn't entirely appropriate, you'll probably want to consider using the HLA *#while* loop that the next section describes.

2.5.6: The #while..#endwhile Compile-Time Loop

HLA provides an indefinite looping mechanism in the compile-time language via the *#while..#endwhile* loop. The *#while..#endwhile* compile-time loop takes the following form:

```
#while( constant_boolean_expression )

   << Statements to emit repeatedly as long >>
   << as the expression is true.            >>

#endwhile
```

While processing the *#while..#endwhile* loop, HLA evaluates the constant boolean expression. If it is false, HLA immediately skips to the first statement beyond the *#endwhile* directive.

If the expression is true, then HLA proceeds to compile the body of the *#while* loop. Upon encountering the *#endwhile* directive, HLA jumps back up to the *#while* clause in the source code and repeats this process until the expression evaluates false.

Warning: since HLA allows you to create loops in your source code that evaluation during the compilation process, HLA also allows you to create *infinite* loops that will lock up the system during compilation. If HLA seems to have gone off into la-la land during compilation and you re using #while loops in your code, it might not be a bad idea to put some *#print* directives[9] into your loop(s) to see if you ve created an infinite loop.

The HLA *#while..#endwhile* loop is great for repeatedly emitting sections of code whose loop control expression varies while processing the loop (unlike the *#for..#endfor* loop, for which HLA can compute the number of iterations when HLA first encounters the *#for* clause). Though nowhere near as popular in modern HLA code as the *#for..#endfor* loop, the *#while..#endwhile* loop is still quite useful in many situations. The #while..#endwhile loop, for example, is quite useful when you want to increment or decrement a non-integer loop control variable, e.g.,

```
? r :real64 := 10.2;
#while( r > 0 )
   << Statements to process while r is greater than zero >>
   ? r := r - 0.1;
#endwhile
```

If you look at some older HLA code (e.g., in the HLA Standard Library), you find the *#while..#endwhile* loop used in many cases where a *#for..#endfor* loop might be more appropriate. This is because the *#for..#endfor* loop is a fairly recent addition to the HLA language, appearing long after the creation of the HLA Standard library.

2.5.7: Compile-Time I/O and Data Facilities

HLAs compile-time language provides several facilities for printing messages during compilation, reading and writing text file data, executing system commands, and processing blocks of text as string data in the HLA source file. Though some of these features are rather esoteric, they are quite useful in some circumstances. We ll take a look at many of these HLA features in this section.

The *#system* directive requires a single string parameter. It executes this string as an operating system (shell/command interpreter) operation via the C *system* function call. This call is useful, for example, to run a program during compilation that dynamically creates a text file that an HLA program may include immediately after the *#system* invocation.

Example of *#system* usage:

```
#system( "dir" )
```

Note that the *#system* directive is legal anywhere white space is allowable and doesn t require a semicolon at the end of the statement.

9. These will be described a little later in this chapter.

Do keep in mind that HLA does not have any control over the programs you run via the #system compile-time language statement. It is your responsibility to ensure that you don t run some program that interferes with the current assembly in process (e.g., deleting the current source file that HLA is compiling would be an example of interference that might leave the current compilation in an undefined state).

The *#print* directive displays its parameter values during compilation. The basic syntax is the following:

```
#print( comma, separated, list, of, constant, expressions, ... )
```

The *#print* statement is very useful for displaying messages during assembly (e.g., when debugging complex macros or compile-time programs). The items in the *#print* list must evaluate to constant (*const* or *val*) values at compile time. This directive is great for displaying status information during compilation. It s also great for tracking down bugs in a compile-time program (e.g., if HLA hangs up and you suspect that this is due to an infinite *#while* loop in your code, you can use *#print* to track down exactly where HLA is hanging up).

The *#error* directive behaves like *#print* insofar as it prints its parameter to the console device during compilation. However, this instruction also generates an HLA error message and does not allow the creation of an object file after compilation. This statement only allows a single string expression as a parameter. If you need to print multiple values of different types, use string concatenation and the *@string* function to achieve this. Example:

```
#error( "Error, unexpected value.  Value = " + #string( theValue ))
```

Notice that neither the *#print* nor the *#error* statements end with a semicolon.

The *#openwrite, #write,* and *#closewrite* compile-time statements let you do simple file output during compilation. The *#openwrite* statement opens a single file for output, *#write* writes data to that output file, and *#closewrite* closes the file when output is complete. These statements are useful for automatically generating *include* files that the source file will include later on during the compilation. These statements are also useful for storing bulk data for later retrieval or generating a log during assembly.

The *#openwrite* statement uses the following syntax:

#openwrite(*string_expression*)

This call opens a single output file using the filename specified by the string expression. If the system cannot open the file, HLA emits a compilation error. Note that *#openwrite* only allows one output file to be active at a time. HLA will report an error if you execute *#openwrite* and there is already an output file open. If the file already exists, HLA deletes it prior to opening it (so be careful!). If the file does not already exist, HLA creates a new one with the specified name.

The *#write* statement uses the same syntax as the *#print* directive. Note, however, that *#write* doesn t automatically emit a newline after writing all its operands to the file; if you want a newline output you must explicitly supply it as the last parameter to *#write*.

The *#closewrite* statement closes the file opened via *#openwrite*. HLA automatically closes this file at the end of assembly if you leave it open. However, you must explicitly close this file before attempting to use the data (via *#include* or *#openread*) in your program. Also, since HLA allows only one open output file at a time, you must use *#closewrite* to close the file before you can open another with *#openwrite*.

Here is an example of the *#openwrite, #write,* and *#closewrite* statements in action. This short compile-time program creates an array of integer values for inclusion elsewhere in the source file:

```
#openwrite( "myDataInclude.hhf" )
#write( "[ " )
```

```
   #for( i := 0 to 2047 )

      #write( i )
      #if( i <> 2047 )
         #write( ", " )
      #elseif( ( i mod 16) == 15 )
         #write( nl )
      #endif
   #endfor
   #write( "];" )
   #closewrite
      .
      .
      .
const
   constArray :dword[ 2048] :=
      #include( "myDataInclude.hhf" )
```

Of course, initializing an array in this manner is rather silly, because we could have (more easily) initialized the array directly with an HLA compile-time program (e.g., the #for loop above with a few minor modifications would have been sufficient). However, it s easy enough to imagine building up the include file at various points throughout the code, thus making the use of *#openwrite/#write/#closewrite* more applicable.

The *#openread, @read,* and *#closeread* compile-time statements and function let you do simple file input during compilation. The *#openread* statement opens a single file for input, *@read* is a compile-time function that reads a line of text from the file, and *#closeread* closes the file when input is complete. These statements are useful for reading files produced by *#openwrite/#write/#closewrite* during compilation, or any other text file for that matter.

The *#openread* statement uses the following syntax:

#openread(*filename*)

The *filename* parameter must be a string expression or HLA reports an error. HLA attempts to open the specified file for reading; HLA prints an error message if it cannot open the file.

The *@read* function uses the following call syntax:

@read(*val_object*)

The *val_object* parameter must either be a symbol you ve defined in a *val* section (or via ?) or it must be an undefined symbol (in which case *@read* defines it as a *val* object). *@read* is an HLA compile-time function (hence the *@* prefix rather than *#* ; HLA uses *#* for compile-time statements). It returns either true or false, true if the read was successful, false if the read operation encountered the end of file. Note that if any other read error occurs, HLA will print an error message and return false as the function result. If the read operation is successful, then HLA stores the string it read (up to 4095 characters) into the *val* object specified by the parameter. Unlike *#openread* and *#closeread*, the *@read* function may not appear at an arbitrary point in your source file. It must appear within a constant expression since it returns a boolean result (and it is your responsibility to check for EOF).

The *#closeread* statement closes the input file. Since you may only have one open input file at a time, you must close an open input file with *#closeread* prior to opening a second file. Syntax:

#closeread

Example of using compile-time file I/O:

```
#openwrite( "hw.txt" )
#write( "Hello World", nl )
#closewrite
#openread( "hw.txt" )
?goodread := @read( s );
#closeread
#print( "data read from file = ", s )
```

The *#text* and *#endtext* directives surround a block of text in an HLA program from which HLA will create an array of string constants. The syntax for these directives is:

```
#text( identifier )

   << arbitrary lines of text >>

#endtext
```

The *identifier* must either be an undefined symbol or an object declared in the *val* section.

This directive converts each line of text between the *#text* and *#endtext* directives into a string and then builds an array of strings from all this text. After building the array of strings, HLA assigns this array to the identifier symbol. This is a *val* constant array of strings. The *#text..#endtext* directives may appear anywhere in the program where white space is allowed.

These directives provide an easy way to initialize a constant array of strings and they provide a convenient alternative to using *#openread/@read/#closeread* when you simply want to inject a fair amount of textual data into your HLA source file for further processing by the compile-time language. Here is an example that uses the *#text..#endtext* to initialize an array of strings in a program.

```
#text( constStringArray )
This goes into the first string.
This line goes into the second string.
The third string will contain this line.
And so on...
#endtext

static
   strArray :string[] := constStringArray;
```

By using *#text..#endtext* or the *#openread..@read..#closeread* compile-time statements, it is actually possible to create your own languages using the HLA compile-time language. Processing the text appearing in a #text..#endtext block, or the text read from a file via the *#openread..@read..#closeread* statements is one place where HLAs compile-time string and pattern matching functions come in real handy.

2.5.8: Macros (Compile-Time Procedures and Functions)

HLA provides one of the most sophisticated macro processing facilities of any assembler, indeed, any language, available. Macros are a great tool for Win32 programmers because a considerable amount of Win32 programming is the repetitive application of some common programming template. Learning how to properly use macros can help you reduce the drudgery often associated with assembly language programming. Unfortunately, HLAs macro sophistication comes with a price tag: if you want to learn the macros inside and out, there is a steep learning curve associated with them. Fortunately, you don t have to learn everything there is to know about

HLA macros in order to effectively employ them in your source files. This section will concentrate on those features of the HLA macro subsystem that a Win32 programmer will commonly use.

You can declare macros almost anywhere HLA allows whitespace in a program using the following syntax:

```
#macro identifier ( optional_parameter_list ) ;
   statements
#endmacro
```

Note that a semicolon does not follow the *#endmacro* clause.

The optional parameter list must be a list of one or more identifiers separated by commas. Unlike procedure declarations, you do not associate a type with macro parameters. HLA automatically associates the type text with all macro parameters (except for one special case noted below). Example:

```
#macro MacroWParms( a, b, c );
   ?a = b + c;
#endmacro
```

Optionally, the last (or only) name in the identifier list may take the form *identifier[]* . This syntax tells the macro that it may allow a variable number of parameters and HLA will create an array of string objects to hold all the parameters (HLA uses a string array rather than a text array because text arrays are illegal).

Example:

```
#macro MacroWVarParms( a, b, c[] );
   ?a = b + text(c[0]) + text(c[1]);
#endmacro
```

If the macro does not allow any parameters, then you follow the identifier with a semicolon (i.e., no parentheses or parameter identifiers). See the first example in this section for a macro without any parameters.

Occasionally you may need to define some symbols that are local to a particular macro invocation (that is, each invocation of the macro generates a unique symbol for a given identifier). The local identifier list allows you to do this. To declare a list of local identifiers, simply following the parameter list (after the parenthesis but before the semicolon) with a colon (:) and a comma separated list of identifiers, e.g.,

```
#macro ThisMacro(parm1):id1,id2;
...
```

HLA automatically renames each symbol appearing in the local identifier list so that the new name is unique throughout the program. HLA creates unique symbols of the form *_XXXX_* where *XXXX* is some hexadecimal numeric value. To guarantee that HLA can generate unique symbols, you should avoid defining symbols of this form in your own programs (in general, symbols that begin and end with an underscore are reserved for use by the compiler and the HLA standard library). Example:

```
#macro LocalSym : i,j;

j: cmp(ax, 0)
   jne( i )
   dec( ax )
   jmp( j )
i:
#endmacro
```

Without the local identifier list, multiple expansions of this macro within the same procedure would yield multiple statement definitions for *i* and *j*. With the local statement present, however, HLA substitutes symbols similar to *_0001_* and *_0002_* for *i* and *j* for the first invocation and symbols like *_0003_* and *_0004_* for *i* and *j* on the second invocation, etc. This avoids duplicate symbol errors if you do not use (poorly chosen) identifiers like *_0001_* and *_0004_* in your code.

The statements section of the macro may contain any legal HLA statements (including definitions of other macros). However, the legality of such statements is controlled by where you expand the macro.

To invoke a macro, you simply supply its name and an appropriate set of parameters. Unless you specify a variable number of parameters (using the array syntax) then the number of actual parameters must exactly match the number of formal parameters. If you specify a variable number of parameters, then the number of actual parameters must be greater than or equal to the number of formal parameters (not counting the array parameter).

During macro expansion, HLA automatically substitutes the text associated with an actual parameter for the formal parameter in the macro s body. The array parameter, however, is a string array rather than a text array so you will have force the expansion yourself using the *@text* function:

```
#macro example( variableParms[] );
   ?@text(variableParms[0]) := @text(variableParms[1]);
#endmacro
```

Actual macro parameters consist of a string of characters up to, but not including a separate comma or the closing parentheses, e.g.,

```
example( v1, x+2*y )
```

v1 is the text for parameter #1, *x+2*y* is the text for parameter #2. Note that HLA strips all leading whitespace and control characters before and after the actual parameter when expanding the code in-line. The example immediately above would expand do the following:

```
?v1 := x+2*y;
```

If (balanced) parentheses appear in some macro s actual parameter list, HLA does not count the closing parenthesis as the end of the macro parameter. That is, the following is perfectly legal:

```
example(  v1, ((x+2)*y) )
```

This expands to:

```
?v1 := ((x+2)*y);
```

If you need to embed commas or unmatched parentheses in the text of an actual parameter, use the HLA literal quotes *#(* and *)#* to surround the text. Everything (except surrounding whitespace) inside the literal quotes will be included as part of the macro parameter s text. Example:

```
example( v1, #( array[0,1,i] )# )
```

The above expands to:

```
   ?v1 := array[ 0,1,i];
```

Without the literal quote operator, HLA would have expanded the code to

```
   ?V1 := array[ 0;
```

and then generated an error because (1) there were too many actual macro parameters (four instead of two) and (2) the expansion produces a syntax error.

Of course, HLAs macro parameter parser does not consider commas appearing inside string or character constants as parameter separators. The following is perfectly legal, as you would expect:

```
   example( charVar, ',' )
```

As you may have noticed in these examples, a macro invocation does not require a terminating semicolon. Macro expansion occurs upon encountering the closing parenthesis of the macro invocation. HLA uses this syntax to allow a macro expansion *anywhere* in an HLA source file. Consider the following:

```
#macro funny( dest )
   , dest );
#endmacro

   mov( 0 funny( ax )
```

This code expands to `mov( 0, ax );` and produces a legal machine instruction. Of course, the this is a truly horrible example of macro use (the style is really bad), but it demonstrates the power of HLA macros in your program. This expand anywhere philosophy is the primary reason macro invocations do not end with a semicolon.

HLA macros provide some very powerful facilities not found in other macro assemblers. One of the really unique features that HLA macros provides is support for multi-part (or context-free) macro invocations. This feature is accessed via the `#terminator` and `#keyword` reserved words. Consider the following macro declaration:

```
program demoTerminator;

#include( "stdio.hhf" );

#macro InfLoop:TopOfLoop, LoopExit;
   TopOfLoop:
#terminator endInfLoop;
   jmp TopOfLoop;
   LoopExit:
#endmacro;

static
   i:int32;

begin demoTerminator;

   mov( 0, i );
   InfLoop

      stdout.put( "i=", i, nl );
      inc( i );
```

```
   endInfLoop;

end demoTerminator;
```

The *#terminator* keyword, if it appears within a macro, defines a second macro that is available for a one-time use after invoking the main macro. In the example above, the *endInfLoop* macro is available only after the invocation of the *InfLoop* macro. Once you invoke the *EndInfLoop* macro, it is no longer available (though the macro calls can be nested, more on that later). During the invocation of the *#terminator* macro, all local symbols declared in the main macro (*InfLoop* above) are available (note that these symbols are not available outside the macro body. In particular, you could not refer to either *TopOfLoop* nor *LoopExit* in the statements appearing between the *InfLoop* and endInfLoop invocations above). The code above, by the way, emits code similar to the following:

```
_0000_:
      stdout.put( "i=", i, nl );
      inc( i );
      jmp _0000_;
_0001_:
```

The macro expansion code appears in italics. This program, therefore, generates an infinite loop that prints successive integer values.

These macros are called multi-part macros for the obvious reason: they come in multiple pieces (note, though, that HLA only allows a single *#terminator* macro). They are also referred to as *Context-Free macros* because of their syntactical nature. Earlier, this document claimed that you could refer to the *#terminator* macro only once after invoking the main macro. Technically, this should have said you can invoke the terminator once for each outstanding invocation of the main macro. In other words, you can nest these macro calls, e.g.,

```
   InfLoop

      mov( 0, j );
      InfLoop

         inc( i );
         inc( j );
         stdout.put( "i=", i, " j=", j, nl );

      endInfLoop;

   endInfLoop;
```

The term *Context-Free* comes from automata theory; it describes this nestable feature of these macros.

As should be painfully obvious from this *InfLoop* macro example, it would be really nice if one could define more than one macro within this context-free group. Furthermore, the need often arises to define limited-scope scope macros that can be invoked more than once (limited-scope means between the main macro call and the terminator macro invocation). The *#keyword* definition allows you to create such macros.

In the *InfLoop* example above, it would be really nice if you could exit the loop using a statement like *brkLoop* (note that *break* is an HLA reserved word and cannot be used for this purpose). The *#keyword* section of a macro allows you to do exactly this. Consider the following macro definition:

```
#macro InfLoop:TopOfLoop, LoopExit;
   TopOfLoop:
#keyword brkLoop;
   jmp LoopExit;
#terminator endInfLoop;
   jmp TopOfLoop;
   LoopExit:
#endmacro;
```

As with the *#terminator* section, the *#keyword* section defines a macro that is active after the main macro invocation until the terminator macro invocation. However, *#keyword* macro invocations to not terminate the multi-part invocation. Furthermore, *#keyword* invocations may occur more that once. Consider the following code that might appear in the main program:

```
   mov( 0, i );
   InfLoop

      mov( 0, j );
      InfLoop

         inc( j );
         stdout.put( "i=", i, " j=", j, nl );
         if( j >= 10 ) then

            brkLoop;

         endif

      endInfLoop;
      inc( i );
      if( i >= 10 ) then

         brkLoop;

      endif;

   endInfLoop;
```

The *brkLoop* invocation inside the *if(j >= 10)* statement will break out of the inner-most loop, as expected (another feature of the context-free behavior of HLAs macros). The *brkLoop* invocation associated with the *if(i >= 10)* statement breaks out of the outer-most loop. Of course, the HLA (run-time) language provides the *forever..endfor* loop and the *break* and *breakif* statements, so there is no need for this *InfLoop* macro, nevertheless, this example is useful because it is easy to understand. If you are looking for a challenge, try creating a statement similar to the C/C++ *switch/case* statement; it is perfectly possible to implement such a statement with HLAs macro facilities, see the HLA Standard Library for an example of the *switch* statement implemented as a macro.

The discussion above introduced the *#keyword* and *#terminator* macro sections in an informal way. There are a few details omitted from that discussion. First, the full syntax for HLA macro declarations is actually:

```
#macro identifier ( optional_parameter_list ) :optional_local_symbols;
   statements
```

```
#keyword identifier ( optional_parameter_list ) :optional_local_symbols;
   statements

note: additional #keyword declarations may appear here

#terminator identifier ( optional_parameter_list ) :optional_local_symbols;
   statements
#endmacro
```

There are three things that should immediately stand out here: (1) You may define more than one *#keyword* within a macro. (2) *#keywords* and *#terminators* allow optional parameters. (3) *#keyword* and *#terminator* allow their own local symbols.

The scope of the parameters and local symbols isn t particularly intuitive (although it turns out that the scope rules are exactly what you would want). The parameters and local symbols declared in the main macro declaration are available to all statements in the macro (including the statements in the *#keyword* and *#terminator* sections). The *InfLoop* macro used this feature because the *jmp* instructions in the *brkLoop* and *endInfLoop* sections referred to the local symbols declared in the main macro. The *InfLoop* macro did not declare any parameters, but had they been present, the *brkLoop* and *endInfLoop* sections could have used those symbols as well.

Parameters and local symbols declared in a *#keyword* or *#terminator* section are local to that particular section. In particular, parameters and/or local symbols declared in a *#keyword* section are not visible in other *#keyword* sections or in the *#terminator* section.

One important issue is that local symbols in a multipart macro are visible in the main code between the start of the multipart macro and the terminating macro. That is, if you have some sequence like the following:

```
InfLoop

   jmp LoopExit;

endInfLoop;
```

Then HLA substitutes the appropriate internal symbol (e.g., "*_xxxx_*") for the *LoopExit* symbol. This is somewhat unintuitive and might be considered a flaw in HLAs design. Future versions of HLA may deal with this issue; in the meantime, however, some code takes advantage of this feature (to mask global symbols) so it s not easy to change without breaking a lot of code. Be forewarned before taking advantage of this feature , however, that it will probably change in HLA v2.x. An important aspect of this behavior is that macro parameter names are also visible in the code section between the initial macro and the terminator macro. Therefore, you must take care to choose macro parameter names that will not conflict with other identifiers in your program. E.g., the following will probably lead to some problems:

```
static
   i:int32;

#macro parmi(i);
   mov( i, eax );
#terminator endParmi;
   mov( eax, i );
#endmacro
   .
   .
```

```
   .
   parmi( xyz );
   mov( i, ebx ); // actually moves xyz into ebx, since the parameter i
                  // overrides the global variable i here.
   endParmi;
```

As mentioned earlier, HLA treats all non-array macro parameters as text constants that are assigned a string corresponding to the actual parameter(s) passed to the macro. I.e., consider the following:

```
#macro SetI( v );
   ?i := v;
#endmacro

SetI( 2 );
```

The above macro and invocation is roughly equivalent to the following:

```
const
   v : text := "2";
   ?i := v;
```

When utilizing variable parameter lists in a macro, HLA treats the parameter object as a string array rather than a text array (because HLA v1.x does not currently support text arrays). For example, consider the following macro and invocation:

```
#macro SetI2( v[] );
   ?i := v[0];
#endmacro

SetI2( 2 );
```

Although this looks quite similar to the previous example, there is a subtle difference between the two. The former example will initialize the constant (value) *i* with the *int32* value two. The second example will initialize *i* with the string value 2 .

If you need to treat a macro array parameter as text rather than as a string object, use the HLA *@text* function that expands a string parameter as text. E.g., the former example could be rewritten as:

```
#macro SetI2( v[] );
   ?i := @text( v[0]);
#endmacro

SetI2( 2 );
```

In this example, the *@text* function tells HLA to expand the string value *v[0]* (which is 2) directly as text, so the "*SetI2(2)*" invocation expands as

```
?i := 2;
rather than as
?i := "2";
```

On occasion, you may need to do the converse of this operation. That is, you may want to treat a standard (non-array) macro parameter as a string object rather than as a text object. Unfortunately, text objects are expanded by the lexer in-line upon initial processing; the compiler never sees the text variable name (or parameter name, in this particular case). To overcome this problem, the lexer has a special feature to avoid expanding text constants if they appear inside an *@string* compile-time function. The following example demonstrates one possible use of this feature:

```
program demoString;

#macro seti3( v );
   #print( "i is being set to " + @string( v ))
   ?i := v;
#endmacro


begin demoString;

   seti3( 4 )
   #print( "i = " + string( i ) )
   seti3( 2 )
   #print( "i = " + string( i ) )

end demoString;
```

If an identifier is a *text* constant (e.g., a macro parameter or a const/value identifier of type *text*), special care must be taken to modify the string associated with that text object. A simple *val* expression like the following won t work:

?textVar:text := "SomeNewText";

The reason this doesn t work is subtle: if *textVar* is already a text object, HLA immediately replaces *textVar* with its corresponding string; this includes the occurrence of the identifier immediately after the ? in the example above. So were you to execute the following two statements:

```
?textVar:text := "x";
?textVar:text := "1";
```

the second statement would not change *textVar's* value from *x* to *1*. Instead, the second statement above would be converted to:

```
?x:text := "1";
```

and *textVar's* value would remain *x*. To overcome this problem, HLA provides a special syntactical entity that converts a text object to a string and then returns the text object ID. The syntax for this special form is "*@tostring:identifier*". The example above could be rewritten as:

```
?textVar:text := "x";
?@tostring:textVar:text := "1";
```

In this example, *textVar* would be a text object that expands to the string 1.

As described earlier, HLA processes as parameters all text between a set of matching parentheses after the macro s name in a macro invocation. HLA macro parameters are delimited by the surrounding parentheses and

commas. That is, the first parameter consists of all text beyond the left parenthesis up to the first comma (or up to the right parenthesis if there is only one parameter). The second parameter consists of all text just beyond the first comma up to the second comma (or right parenthesis if there are only two parameters). Etc. The last parameter consists of all text from the last comma to the closing right parenthesis.

Note that HLA will strip away any white space at the beginning and end of the parameter s text (though it does not remove any white space from the interior of the parameter s text).

If a single parameter must contain commas or parentheses, you must surround the parameter with the literal text macro quotes `#(` and `)#`. HLA considers everything but leading and trailing space between these macro quote symbols as a single parameter. Note that this applies to macro invocations appearing within a parameter list. Consider the following (erroneous) code:

```
CallToAMacro( 5, "a", CallToAnotherMacro( 6,7 ), true );
```

Presumably, the `( 6,7 )` text is the parameter list for the `CallToAnotherMacro` invocation. When HLA encounters a macro invocation in a parameter list, it defers the expansion of the macro. That is, the third parameter of `CallToAMacro` should expand to `CallToAnotherMacro( 6,7 )`, not the text that `CallToAnotherMacro` would expand to. Unfortunately, this example will not compile correctly because the macro processor treats the comma between the 6 and the 7 as the end of the third parameter to `CallToAMacro` (in other words, the third parameter is actually `CallToAnotherMacro( 6` and the fourth parameter is `7 )`. If you really need to pass a macro invocation as a parameter, use the `#(` and `)#` macro quotes to surround the interior invocation:

```
CallToAMacro( 5,  a , #( CallToAnotherMacro( 6,7 ) )#, true );
```

In this example, HLA passes all the text between the `#(` and `)#` markers as a single parameter (the third parameter) to the `CallToAMacro` macro.

This example demonstrates another feature of HLAs macro processing system - HLA uses *deferred macro parameter expansion*. That is, the text of a macro parameter is expanded when HLA encounters the formal parameter within the macro s body, *not* while HLA is processing the actual parameters in the macro invocation (which would be *eager* evaluation).

There are three exceptions to the rule of deferred parameter evaluation: (1) text constants are always expanded in an eager fashion (that is, the value of the text constant, not the text constant s name, is passed as the macro parameter). (2) The `@text` function, if it appears in a parameter list, expands the string parameter in an eager fashion. (3) The `@eval` function immediately evaluates its parameter; the discussion of `@eval` appears a little later.

In general, there is very little difference between eager and deferred evaluation of macro parameters. In some rare cases there is a semantic difference between the two. For example, consider the following two programs:

```
program demoDeferred;
#macro two( x, y ):z;
   ?z:text:="1";
   x+y
#endmacro

const
   z:string := "2";

begin demoDeferred;
```

```
   ?i := two( z, 2 );
   #print( "i=" + string( i ))

end demoDeferred;
```

In the example above, the code passes the actual parameter *z* as the value for the formal parameter *x*. Therefore, whenever HLA expands *x* it gets the value *z* which is a local symbol inside the *two* macro that expands to the value *1*. Therefore, this code prints *3* (*1* plus *y's* value which is *2*) during assembly. Now consider the following code:

```
program demoEager;
#macro two( x, y ):z;
   ?z:text:="1";
   x+y
#endmacro

const
   z:string := "2";

begin demoEager;

   ?i := two( @text( z ), 2 );
   #print( "i=" + string( i ))

end demoEager;
```

The only differences between these two programs are their names and the fact that *demoEager* invocation of *two* uses the *@text* function to eagerly expand *z's* text. As a result, the formal parameter *x* is given the value of *z's* expansion (*2*) and HLA ignores the local value for *z* in macro *two*. This code prints the value *4* during assembly. Note that changing *z* in the main program to a text constant (rather than a string constant) has the same effect:

```
program demoEager;
#macro two( x, y ):z;
   ?z:text:="1";
   x+y
#endmacro

const
   z:text := "2";

begin demoEager;

   ?i := two( z, 2 );
   #print( "i=" + string( i ))

end demoEager;
```

This program also prints *4* during assembly.

One place where deferred vs. eager evaluation can get you into trouble is with some of the HLA built-in functions. Consider the following HLA macro:

```
   #macro DemoProblem( Parm );
```

```
    #print( string( Parm ) )

#endmacro
    .
    .
    .
DemoProblem( @linenumber );
```

(The *@linenumber* function returns, as an *uns32* constant, the current line number in the file).

When this program fragment compiles, HLA will use deferred evaluation and pass the text *@linenumber* as the parameter *Parm*. Upon compilation of this fragment, the macro will expand to *#print(string(@linenumber))* with the intent, apparently, being to print the line number of the statement containing the *DemoProblem* invocation. In reality, that is not what this code will do. Instead, it will print the line number, in the macro, of the *#print(string (Parm));* statement. By delaying the substitution of the current line number for the *@linenumber* function call until inside the macro, deferred execution produces the wrong result. What is really needed here is eager evaluation so that the *@linenumber* function expands to the line number string before being passed as a parameter to the *DemoProblem* macro. The *@eval* built-in function provides this capability. The following coding of the *DemoProblem* macro invocation will solve the problem:

```
    DemoProblem( @eval( @linenumber ) );
```

Now the compiler will execute the *@linenumber* function and pass that number as the macro parameter text rather than the string *@linenumber*. Therefore, the *#print* statement inside the macro will print the actual line number of the *DemoProblem* statement rather than the line number of the *#print* statement.

Of course, always having to type *@eval(@linenumber)* whenever you want to pass *@linenumber* as a macro parameter can get rather burdensome. Fortunately, you can create a text constant to ease this problem for you:

```
const
    evalLnNum :text := "@eval( @linenumber )";
    .
    .
    .
    DemoProblem( evalLnNum );
```

Because text constants expand in place (before any evaluation takes place), this code properly substitutes the current line number (of the *DemoProblem* statement) for the macro parameter.

2.5.9: Performance of the HLA Compile-Time Language

The HLA compile-time language was written to save development time by giving the assembly language programmer the ability to automate the creation of long or complex sequences of instructions. Because of HLA's design and the languages used to implement HLA (Flex, Bison, and C), HLA uses a pure interpreter implementation for the compile-time language. This means that HLA processes the text in your source file directly when executing a compile-time program. Unfortunately, pure interpretation is one of the slowest forms of program execution in common use. For the most part, the fact that HLA interprets compile-time language relatively slowly is a moot issue. Modern machines are sufficiently fast that even a pure interpretation scheme won't normally introduce a noticeable delay in the compilation time of your programs. However, the overuse of HLA's

compile-time facilities can lead to slower development times, but consuming several seconds (or even minutes in some extreme cases) to interpret all the compile-time language statements in a source file.

Before discussing this subject any further, it s important for you to realize that we are talking about compilation times here, not the run-time of your actual application. The speed of HLA s compile-time language has nothing to do with the execution time of an application you write in HLA. So don t confuse the two and be concerned that HLA s compile-time language is somehow slowing down your applications.

As noted, modern machines are sufficiently fast that even HLA s pure interpretation execution model for compile-time programs won t have a tremendously significant impact on the compilation times of your programs. However, don t forget one thing - whenever HLA expands a macro the system isn t processing a single line of text in the source file, it s actually processing every line of text in that macro *for each invocation of the macro*. Likewise, whenever HLA processes a #while loop or a #for loop it isn t simply processing the number of lines in the loop s body, it s processing that number of lines *times the number of loop iterations*. It is very easy to create a very short HLA source file that takes a tremendous amount of time to compile. For example, on a 2GHz Pentium IV machine, the following trivial HLA program requires about a minute to compile:

```
program t;
   #for( i:= 0 to 100_000_000 )
   #endfor
begin t;
end t;
```

The fact that HLA is actually processing 1.7 million lines per second while compiling this program (a phenomenal number) is of little relief to the programmer waiting around a minute for this program to compile. The average programmer sees a short source file with only five statements and questions why the assembler would take nearly a minute to compile this file. They don t see the fact that as far as HLA is concerned, this source file is actually over one hundred million lines long. Compiling 100,000,000 lines of source code in a minute is very impressive. It s just not obvious to most people looking at this source code that this is what is going on.

Though examples like this one are rather rare, it is possible to create HLA macros that consume a fair amount of time, particularly if you invoke those macros many times within a source file. For example, the HLA stdout.put macro is between 400 and 500 source lines long (not counting all the loops present in the macro). If you stick a thousand invocations of stdout.put (each with multiple parameters) into a source file, you will measure compilation time in *minutes*, not seconds, on typical PCs. Counting loops and other features in the stdout.put macro, an invocation of this macro typically requires the interpretation of 500 or so HLA compile-time language statements per macro argument. A source file with 1,000 stdout.put macro invocations can easily expand to between 1,000,000 and 2,000,000 compile-time statements that HLA must interpret (which is slow, much slower than processing the empty `#for` loop in the previous example).

The moral of this story is that if you intend to write (or use) some extremely complex macros, and you intend to invoke that macro many, many, times in your source file, be prepared for slower than usual compilation times. If this loss of performance hinders your development, you might consider using separate compilation and splitting up the macro invocations across as many source files as possible so you don t have to recompile the entire program every time you make a minor change.

2.5.10: A Complex Macro Example: stdout.put

Before concluding this chapter, it would be a good idea to provide an example of a relatively complex HLA compile-time program. Probably the most common example of just such a compile-time program is the HLA Standard Library s `stdout.put` macro. The stdout.put macro is interesting because it parses the list of argu-

ments you supply, determines the type of the argument, extracts optional formatting information (if present), and calls an appropriate HLA Standard Library routine to actually print the value to the standard output device. Using stdout.put is far more convenient than calling all the individual routines in the HLA Standard Library.

Note: although we won t use the stdout.put macro much in this book, it s mainly useful for writing console applications, not GUI apps, we will make use of the *fileio.put* and *str.put* macros which work in a similar fashion to *stdout.put*. Even if these other macros weren t useful to Win32 programmers, the concepts that stdout.put employs are quite useful for Win32-based macros.

To begin with, you should note that the HLA *stdout.put* macro is really the *put* macro that just happens to be defined in the *stdout* name space. We ll continue to call it *stdout.put* in this discussion to differentiate it from other *put* macros that appear in the HLA Standard Library. This macro s definition appears in the *stdout.hhf* header file; Those who would like to see the original source code can find *stdout.hhf* in the HLA include subdirectory.

The *stdout.put* macro allows zero or more arguments. To handle this, the put macro declaration defines a variable parameter list (using HLAs variable parameter list syntax -- an open ended array). In addition to this varying parameter list, the *put* macro uses several local symbols within the macro, they are all part of the *stdout.put* macro declaration:

```
#macro put( _parameters_[] ):
    _curparm_, _pType_, _arg_, _width_,
    _decpts_, _parmArray_, _id_, _tempid_, _fieldCnt_;
```

In order to process each parameter the caller supplies, the *stdout.put* macro uses a compile-time variable and a *#while* loop to step through each of the elements of the *_parameters_* array. This could actually be done more conveniently with a *#for* loop, but the *stdout.put* macro was written long before the *#for* loop was added to the HLA language, hence the use of the *#while* loop to simulate a *#for* loop:

```
    ?_curparm_:uns32 := 0;


    // The following loop repeats once for each PUT parameter
    // we process.

    #while( _curparm_ < @elements( _parameters_ ))
```

The *stdout.put* macro allows operands to take the following form: *operand, operand:n,* or *operand:n:m.* The *n* and *m* items provide a minimum field width (*n*) and positions after the decimal point (*m*, for real values). In order to properly process each argument, HLA needs to split up each parameter into one, two, or three separate strings, depending on the presence of the field width and decimal point options. HLA maintains this information in an array of strings and uses the local symbol *_parmArray_* to hold these strings. On each iteration of the loop, HLA redefines *_parmArray_* as an *uns32* object in order to free the string storage used by the previous iteration of the loop. After doing this, the macro calls the @tokenize compile-time function to break up the current parameter string into various parts (see the *HLA Reference Manual* for a complete description of the *@tokenize* function; for our purposes, just assume that it puts the actual operand name into the first element of the *_parmArray_* array of strings, the field width into the second element, and the decimal point position into the third element).

```
        ?_parmArray_:uns32 := 0;

        ?_parmArray_ := @tokenize
```

```
                    (
                        _parameters_[ _curparm_ ],
                        0,
                        { ':'},
                        {
                            '"',
                            ''',
                            '[ ',
                            '] ',
                            '(',
                            ')',
                            '{ ',
                            '} '
                        }
                    );
```

Next, the stdout.put macro does some processing on the first portion of the current parameter to determine if we ve got an identifier or an expression. First, this code strips away any leading and trailing spaces and then checks to see if the string begins with an identifier. If so, the code invokes the stdio._GetID_ macro that extracts the identifier from the start of the string.

```
?_arg_ := @trim( _parmArray_[ 0 ], 0 );
#if( char( _arg_ ) in stdio._idchars_ )

    // If this parameter begins with an id character,
    // then strip away any non-ID symbols from the
    // end of the string.  Then determine if we've
    // got a constant or some other class (e.g.,
    // variable or procedure).  If not a constant,
    // keep only the name.  If a constant, we need
    // to keep all trailing characters as well.

    ?_id_ := stdio._GetID_( _arg_ );
    #if
    (
            @class( _id_ ) = hla.cConstant
        |   @class( _id_ ) = hla.cValue
    )

        ?_id_ := _arg_;

    #endif

#else

    // If it's not an ID, we need to keep everything.

    ?_id_ := _arg_;

#endif
```

After extracting the first operand, the next step is to process the optional field width and fractional width components. The *stdout.put* macro determines if these fields are present by checking the number of elements in the *_parmArray_* object. If the number of elements is two or greater, then *stdout.put* assumes that the sec-

ond array element holds the minimum field width. If the number of elements is three or greater, then *stdout.put* assumes that the third element contains the fractional width information.

```
// Okay, determine if the caller supplied a field width
// value with this parameter.

?_fieldCnt_ := @elements( _parmArray_ );
#if( _fieldCnt_ > 1 )

    ?_width_ := @trim( _parmArray_[ 1 ], 0 );

#else

    ?_width_ := "-1";       // Default width value.

#endif

// Determine if the user supplied a fractional width
// value with this parameter.

#if( _fieldCnt_ > 2 )

    ?_decpts_ := @trim( _parmArray_[ 2 ], 0 );

#else

    ?_decpts_ := "-1";  // Default fractional value.

#endif

// Quick check to see if the user supplied too many
// width fields with this parameter.

#if( _fieldCnt_ > 3 )

    #error
    (
        "<<" + _parameters_[ _curparm_ ] + ">>" +
        " has too many width fields"
    );

#endif
```

After extracting the components of the current parameter, the stdout.put macro now goes about its business of determining the type of the current parameter so it can determine which HLA Standard Library function to call to actually print the value. This code sequence uses the HLA *@pType* and *@typename* compile-time functions to determine the symbol s type (see the *HLA Reference Manual* for more details). Note that this code also handles arrays of these objects by determining the base address of the array object (this is what the *#while* loop is doing in the following code sequence).

```
// Determine the type of this parameter so we can
// call the appropriate routine to print it.

?_pType_ := @pType( @text( _id_ ));
?_tempid_ := _id_;
```

```
#while( _pType_ = hla.ptArray )

    ?_tempid_ := @typename( @text( _tempid_ ));
    ?_pType_ := @pType( @text( _tempid_ ));

#endwhile
```

Once stdout.put has a handle on the parameter s type, it invokes the stdout._put_ macro (which we ll describe in a moment) to actually emit the code. Note that the *@pType* compile-time function returns an integer value that specifies a primitive HLA type (hence the name *pType*) and the corresponding type values have been given meaningful names in the *hla.hhf* header file (which this code is using). Another interesting aspect to this code is that if the argument is a class object, then it will automatically call a put method for that class, if such a method exists. This is how *stdout.put* extends its ability to print user-defined data types.

```
// Based on the type, call the appropriate library
// routine to print this value.

#if( _pType_ = hla.ptBoolean )
    stdout._put_( stdout.putbool, boolean )

#elseif( _pType_ = hla.ptUns8 )
    stdout._put_( stdout.putu8, uns8 )

#elseif( _pType_ = hla.ptUns16 )
    stdout._put_( stdout.putu16, uns16 )

#elseif( _pType_ = hla.ptUns32 )
    stdout._put_( stdout.putu32, uns32 )

#elseif( _pType_ = hla.ptUns64 )
    stdout._put_( stdout.putu64, uns64 )

#elseif( _pType_ = hla.ptUns128 )
    stdout._put_( stdout.putu128, uns128 )

#elseif( _pType_ = hla.ptByte )
    stdout._put_( stdout.putb, byte )

#elseif( _pType_ = hla.ptWord )
    stdout._put_( stdout.putw, word )

#elseif( _pType_ = hla.ptDWord )
    stdout._put_( stdout.putd, dword )

#elseif( _pType_ = hla.ptQWord )
    stdout._put_( stdout.putq, qword )

#elseif( _pType_ = hla.ptLWord )
    stdout._put_( stdout.putl, lword )

#elseif( _pType_ = hla.ptInt8 )
    stdout._put_( stdout.puti8, int8 )

#elseif( _pType_ = hla.ptInt16 )
    stdout._put_( stdout.puti16, int16 )
```

```
#elseif( _pType_ = hla.ptInt32 )
    stdout._put_( stdout.puti32, int32 )

#elseif( _pType_ = hla.ptInt64 )
    stdout._put_( stdout.puti64, int64 )

#elseif( _pType_ = hla.ptInt128 )
    stdout._put_( stdout.puti128, int128 )

#elseif( _pType_ = hla.ptChar )
    stdout._put_( stdout.putc, char )

#elseif( _pType_ = hla.ptCset )
    stdout._put_( stdout.putcset, cset )

#elseif( _pType_ = hla.ptReal32 )
    stdout._put_( stdout.putr32, real32 )

#elseif( _pType_ = hla.ptReal64 )
    stdout._put_( stdout.putr64, real64 )

#elseif( _pType_ = hla.ptReal80 )
    stdout._put_( stdout.putr80, real80 )

#elseif( _pType_ = hla.ptString )
    stdout._put_( stdout.puts, string )

#elseif( @isclass( @text( _parameters_[ _curparm_ ] )))

    #if
    (
        @defined
        (
            @text( _parameters_[ _curparm_ ] + ".toString" )
        )
    )

        push( eax );
        push( esi );
        push( edi );
        @text
        (
            _parameters_[ _curparm_ ] +
            ".toString()"
        );
        puts( eax );
        strfree( eax );
        pop( edi );
        pop( esi );
        pop( eax );

    #else

        #error
        (
            "stdout.put: Class does not provide a toString " +
```

```
                    "method or procedure"
                );

            #endif

        #else

            #error
            (
                "stdout.put: Unknown data type (" +
                _parameters_[ _curparm_ ] +
                ":" +
                @typename( @text( _id_ )) +
                ")"
            );

        #endif
        ?_curparm_ := _curparm_ + 1;

    #endwhile

    // The following is stuck here just to require
    // that the user end the stdout.put(--) invocation
    // with a semicolon.

    static
        ;
    endstatic


#endmacro;
```

The *stdout.put* macro actually invokes a couple of macros. One of these is the *stdio._GetID_* macro (which we ll not cover here, see the source code in the *stdio.hhf* header file found in the HLA include subdirectory for details). The second macro that *stdout.put* invokes is the *stdout._put_* macro that expands into a call to a specific HLA Standard Library function depending on the type of the parameter that *stdout.put* is processing. This macro calculates the size information and fills in the parameter to the actual HLA Standard Library call.

```
#macro _put_( _routine_, _typename_ ):
                        _func_, sizeParms, _realsize_, _typ_;

    ?_func_:string := @string(_routine_);
    ?sizeParms:string := "";
    ?_typ_:string := @string(_typename_)

    // Real values allow two size parameters (width & decpts).

    #if( @substr( _typ_, 0, 4 ) = "real" )

        // Note: on entry, typename = real32, real64, or real80 and
        // routine = putr32, putr64, putr80, fputr32, fputr64, or
        // fputr80.

        ?_realsize_:string := @substr( _typ_, 4, 2 );
```

```
    #if( _width_ <> "-1" )


        // If decpts is <> -1, print in dec notation,
        // else print in sci notation.

        #if( _decpts_ <> "-1" )

            ?sizeParms:string := "," + _width_ + "," + _decpts_;

        #else

            ?_func_:string := "stdout.pute" + _realsize_;
            ?sizeParms:string := "," + _width_;

        #endif

    #else

        // If the user did not specify a format size,
        // then use the puteXX routines with default
        // sizes of: real32=15, real64=22, real80=28.

        ?_func_:string := "stdout.pute" + _realsize_;
        #if( _realsize_ = "32" )

            ?sizeParms:string := ",15";

        #elseif( _realsize_ = "64" )

            ?sizeParms:string := ",20";

        #else

            ?sizeParms:string := ",23";

        #endif

    #endif

#else //It's not a real type.

    #if( _decpts_ <> "-1" )

        #error
        (
            "Fractional width specification is not supported here"
        )

    #elseif( _width_ <> "-1" )

        // Width specifications are only supported for
        // certain types.  Check for that here.

        #if
        (
                _typ_ <> "uns8"
```

```
            &   _typ_ <> "uns16"
            &   _typ_ <> "uns32"
            &   _typ_ <> "uns64"
            &   _typ_ <> "uns128"
            &   _typ_ <> "int8"
            &   _typ_ <> "int16"
            &   _typ_ <> "int32"
            &   _typ_ <> "int64"
            &   _typ_ <> "int128"
            &   _typ_ <> "char"
            &   _typ_ <> "string"
            &   _typ_ <> "byte"
            &   _typ_ <> "word"
            &   _typ_ <> "dword"
            &   _typ_ <> "qword"
            &   _typ_ <> "lword"
        )

            #error
            (
                "Type " +
                _typ_ +
                " does not support width format option"
            )

        #else

            ?_func_:string := _func_ + "Size";
            ?sizeParms:string := "," + _width_ + ", ' '";

        #endif

    #endif

#endif

// Here's the code that calls the appropriate function based on the parameter's
// type and format information:

#if
(
        @isconst( @text( _arg_ ))
    &   _typ_ = "string"
    &   _arg_ = "#13 #10"
)
    stdout.newln();

#elseif( @isconst( @text( _arg_ )) )

    @text( _func_ )( @text( _arg_ ) @text( sizeParms ));

#else

    @text( _func_ )
        ( (type _typename_ @text( _arg_ )) @text( sizeParms ));

#endif
```

```
#endmacro;
```

2.6: Even More Advanced HLA Programming...

This chapter could go on and on forever. HLA is a very sophisticated assembly language and provides tons of features that this chapter doesn t even touch upon. However, we do need to get on with the real purpose of this book, learning Win32 assembly language programming. We ve covered enough advanced HLA programming to deal with just about everything you will encounter the need for when writing Win32 assembly applications. Of course, if you want to learn more about HLA, here are two suggestions: (1) Read the HLA Reference manual and (2) write lots of HLA code and experiment.

Chapter 3: The C - Assembly Connection

3.1: Why are We Reading About C?

You probably purchased this book to learn assembly language programming under Windows (after all, that's what the title promises). This chapter is going to spend considerable time talking about the C programming language. Now assembly language programmers fall into two camps with respect to the C programming language: those who already know it and don't really need to learn a whole lot more about C, and those who don't know C and probably don't want to learn it, either. Unfortunately, as the last chapter points out, the vast majority of Windows programming documentation assumes that the reader is fluent in C. This book cannot begin to provide all the information you may need to write effective Win32 applications; therefore, this chapter does the next best thing - it describes how you can translate that C documentation for use in your assembly language programs.

This chapter contains two main sections. The first section provides a basic description of the C programming language for those readers who are not familiar with the C/C++ programming language. It describes various statements in C/C++ and provides their HLA equivalents. Though far from a complete course on the C programming language, this section will provide sufficient information to read some common Win32 programming examples in C and translate them into assembly language. Experienced C/C++ programmers can elect to skip this section (though if you're not comfortable with HLA, you may want to skim over this section because it will help you learn HLA from a C perspective). The second portion of this chapter deals with the Win32 interface and how C passes parameter data to and from Windows. Unless you're well-versed in compiler construction, mixed language calling sequences, and you've examined a lot of compiler code, you'll probably want to take a look at this material.

3.2: Basic C Programming From an Assembly Perspective

The C programming language is a member of the group of programming languages known as the *imperative* or *procedural* programming languages. Languages in this family include FORTRAN, BASIC, Pascal (Delphi/Kylix), Ada, Modula-2, and, of course, C. Generally, if you've learned to write programs in one of these languages, it's relatively easy to learn one of the other languages in the same category. When you attempt to learn a new language from a different class of languages (i.e., you switch *programming paradigms*), it's almost like you're learning to program all over again; learning a new language that is dissimilar to the one(s) you already know is a difficult task. A recent trend in programming language design has been the *hybrid language*. A hybrid language bridges the gap between two different programming paradigms. For example, the C++ language is a hybrid language that shares attributes common to both procedural/imperative languages and object-oriented languages. Although hybrid languages often present some compromises on one side or the other of the gulf they span, the advantage of a hybrid language is that it is easy to learn a new programming paradigm if you're already familiar with one of the programming methodologies that the language presents. For example, programmers who already know find it much easier to learn object-oriented programming via C++ rather than learning the object-oriented programming paradigm from scratch, say by learning Smalltalk (or some other "pure" object-oriented language). So hybrid languages are good in the sense that they help you learn a new way of programming by leveraging your existing knowledge.

The High Level Assembler, HLA, is a good example of a hybrid programming language. While a true assembly language, allowing you to do just about anything that is possible with a traditional (or *low-level*) assembler, HLA also inherits some syntax and many other features from various high-level imperative programming languages. In particular, HLA borrows several control and data structures from the C, Pascal, Ada, and Modula-2 programming languages. The original intent for this design choice was to make it easier to learn assembly lan-

guage if you already knew an high level language like Pascal or C/C++. By borrowing heavily from the syntax of these high-level programming languages, a new assembly language programmer could learn assembly programming much more rapidly by leveraging their C/C++/Pascal knowledge during the early phase of their assembly education.

Note, however, that the reverse is also true. Someone who knows HLA well but doesn't know C can use their HLA knowledge to help them learn the C programming language. HLA's high level control structures are strongly based on languages like C and Modula-2 (or Ada); therefore, if you're familiar with HLA's high level control structures, then learning C's control structures will be a breeze. The sections that immediately follow use this concept to teach some basic C syntax. For those programmers who are not comfortable or familiar with HLA's high level control structures, the following subsections will also describe how to convert between "pure" assembly language and various C control structures. The ultimate goal here is to show you how to convert C code to HLA assembly code; after all, when reading some Win32 programming documentation, you're going to need to convert the examples you're reading in C into assembly language. Although it is always possible (and very easy) to convert any C control structure directly into assembly language, the reverse is not true. That is, it is possible to devise some control flow scheme in assembly language that does not translate directly into a high level language like C. Fortunately, for our purposes, you generally won't need to go in that direction. So even though you're learning about C from an assembly perspective (that is, you're being taught how to read C code by studying the comparable assembly code), this is not a treatise on converting assembly into C (which can be a very difficult task if the assembly code is not well structured).

3.2.1: C Scalar Data Types

The C programming language provides three basic scalar data types[1]: integers, and a couple floating point types. Other data types you'd find in a traditional imperative programming language (e.g., character or boolean values) are generally implemented with integer types in C. Although C only provides three basic scalar types, it does provide several variations of the integer and floating point types. Fortunately, every C data type maps directly to an HLA structured data type, so conversion from C to HLA data types is a trivial process.

3.2.1.1: C and Assembler Integer Data Types

The C programming language specifies (up to) four different integer types: `char` (which, despite its name, is a special case of an integer value), `short, int,` and `long`. A few compilers support a fifth size, "long long". In general, the C programming language does not specify the size of the integer values; that decision is left to whomever implements a specific compiler. However, when working under Windows (Win32), you can make the following assumptions about integer sizes:

- char - one byte
- short - two bytes
- int, long - four bytes

1. For our purposes, a scalar data type is a primitive or atomic data type; one that the language treats as a single unit, that isn't composed of smaller items (like, say, elements of an array or fields of a structure).

The C programming language also specifies two types of integers: signed and unsigned. By default, all integer values are signed. You can explicitly specify unsigned by prefacing one of these types with the keyword *unsigned*. Therefore, C's integral types map to HLA's types as shown in Table 3-1.

Table 3-1: Integer Type Correspondence Between HLA and C

C Type	Corresponding HLA Types
char	char, byte, int8[a]
short	word, int16
int	dword, int32
long	dword, int32
long long	qword, int64
unsigned char	char, byte, uns8
unsigned short	word, uns16
unsigned	dword, uns32
unsigned int	dword, uns32
unsigned long	dword, uns32
unsigned long long	qword, uns64

a.Some compilers have an option that lets you specify the use of unsigned char as the default. In this case, the corresponding HLA type is uns8.

Generic integer literal constants in C take several forms. C uses standard decimal representation for base 10 integer constants, just like most programming languages (including HLA). For example, the sequence of digits:

```
128
```

represents the literal integer constant 128.

If a literal integer constant begins with a zero (followed by one or more octal digits in the range 0..7), then C treats the literal constant as a base-8 (octal) value. HLA doesn't support octal constants, so you will have to manually convert such constants to decimal or hexadecimal prior to using them in an assembly language program. Fortunately, you rarely see octal constants in modern C programs (especially in Win32 programs).

C integer literal constants that begin with "0x" are hexadecimal (base-16) constants. You will replace the "0x" prefix with a "$" prefix when converting the value from C to HLA. For example, the C literal constant "0x1234ABCD" becomes the HLA literal constant "$1234ABCD".

C also allows the use of an "L" suffix on a literal integer constant to tell the compiler that this should be a long integer value. HLA automatically adjusts all literal constants to the appropriate size, so there is no need to tell HLA to extend a smaller constant to a long (32-bit) value. If you encounter an "L" suffix in a C literal constant, just drop the suffix when translating the value to assembly.

3.2.1.2: C and Assembly Character Types

As the previous section notes, C treats character variables and constants as really small (one-byte) integer values. There are some non-intuitive aspects to using C character variables that can trip you up; hence the presence of this section.

The first place to start is with a discussion of C and HLA literal character constants. The two literal forms are quite similar, but there are just enough differences to trip you up if you're not careful. The first thing to note is that both HLA and C treat a character constant differently than a string containing one character. We'll cover character strings a little later in this chapter, but keep in mind that character objects are not a special case of a string object.

A character literal constant in C and HLA usually consists of a single character surrounded by apostrophe characters. E.g., 'a' is a character constant in both of these languages. However, HLA and C differ when dealing with non-printable (i.e., control) and a couple of other characters. C uses an *escape character sequence* to represent the apostrophe character, the backslash character, and the control characters. For example, to represent the apostrophe character itself, you'd use the C literal constant '\''. The backslash tells C to treat the following value specially; in this particular case, the backslash tells C to treat the following apostrophe character as a regular character rather than using it to terminate the character constant. Likewise, you use '\\' to tell C that you want a single backslash character constant. C also uses a backslash followed by a single lowercase alphabetic character to denote common control characters. Table 3-2 lists the escape character sequences that C defines.

Table 3-2: C Escape Character Sequences

C Escape Sequence	Control Character
'\n'	New line (carriage return/line feed under Windows, though C encodes this as a single line feed)
'\r'	Carriage return
'\b'	Backspace
'\a'	Alert (bell character, control-G)
'\f'	Form Feed (control-L)
'\t'	Tab character (control-I)
'\v'	Vertical tab character (control-k)

C also allows the specification of the character's numeric code by following a backslash with an octal or hexadecimal constant in the range 0..0xff, e.g., '\0x1b'.

HLA does not support escape character sequences using the backslash character. Instead, HLA uses a pound sign ('#') followed immediately by a numeric constant to specify the ASCII character code. Table 3-3 shows how to translate various C escape sequences to their corresponding HLA literal character constants.

Table 3-3: Converting C Escape Sequences to HLA Character Constants

C Escape Sequence	HLA Character Constant	Description
'\n'	#$a #$d	Note that the end of line sequence under Windows is not a character, but rather a string consisting of two characters. If you need to represent newline with a single character, using a linefeed (as see does) whose ASCII code is $A; linefeed is also defined in the HLA Standard Library as stdio.lf. Note that the "nl" symbol an HLA user typically uses for newline is a two-character string containing line feed followed by carriage return.
'\r'	#$d	Carriage return character. This is defined in the HLA Standard Library as stdio.cr.
'\b'	#8	Backspace character. This is defined in the HLA Standard Library as stdio.bs.
'\a'	#7	Alert (bell) character. This is defined in the HLA Standard Library as stdio.bell.
'\f'	#$c	Form feed character.
'\t'	#9	Tab character. This is defined in the HLA Standard Library as stdio.tab.
'\v'	#$b	Vertical tab character.

Because C treats character values as single-byte integer values, there is another interesting aspect to character values in C - they can be negative. One might wonder what "minus 'z'" means, but the truth is, there really is no such thing as a negative character; C simply uses signed characters to represent small integer values in the range -128..+127 (versus unsigned characters that represent values in the range 0..255). For the standard seven-bit ASCII characters, the values are always positive, regardless of whether you're using a signed character or an unsigned character object. Note, however, that many C functions return a signed character value to specify certain error conditions or other states that you cannot normally represent within the ASCII character set. For example, many functions return the character value minus one to indicate end of file.

3.2.1.3: C and Assembly Floating Point (Real) Data Types

The C programming language defines three different floating point sizes: *float, double,* and *long double*[2]. Like the integer data type sizes, the C language makes no guarantees about the size or precision of floating point values other than to claim that *double* is at least as large as *float* and *long double* is at least as large as *double*. However, while nearly every C/C++ compiler that generates code for Windows uses the same sizes for integers (8, 16, and 32 bits for *char, short,* and *int/long*), there are differences in the sizes of floating objects among compilers. In particular, some compilers use a 10-byte extended precision format for *long double* (e.g., Borland) while others use an eight-byte double precision format (e.g., Microsoft). Fortunately, all (reasonable) compilers running under Windows use the IEEE 32-bit single-precision format for *float* and the IEEE 64-bit double-precision format for *double*. If you encounter a *long double* object in C code, you will have to check with the com-

2. Only recent versions of the C programming language support the "long double" floating point type.

piler's vendor to determine the size of the object in order to convert it to assembly language. Of course, for many applications it won't really matter if you go ahead and use a 10-byte real value whenever you encounter a *long double* object. After all, the original programmer is probably expecting something bigger than a *double* object anyway. Do keep in mind, however, that some algorithms involving real arithmetic may not be stable when run with a different floating point size other than the sized used when they were developed.

T shows the correspondence between C/C++ floating point data types and HLA's floating point data types.

Table 3-4: Real Type Correspondence Between C and Assembly

C Real Type	Corresponding HLA Type	Comment
float	real32	32-bit IEEE format floating point value.
double	real64	64-bit IEEE format floating point value.
long double	real64	64-bit IEEE format floating point value on certain compilers (e.g., Microsoft).
	real80	80-bit IEEE format floating point value on certain compilers (e.g., Borland)

C and HLA floating point literal constants are quite similar. They may begin with an optional sign, followed by one or more decimal digits. Then you can have an optional decimal point followed by another optional sequence of decimal digits; following that, you can have an optional exponent specified as an 'e' or 'E', an optional sign, and one or more decimal digits. The final result must not look like an integer constant (i.e., the decimal point or an exponent must be present).

C allows an optional "F" suffix on a floating point constant to specify single precision, e.g., 1.2f. Similarly, you can attach an "L" suffix to a float value to indicate *long double*, e.g., 1.2L. By default, C literal floating point constants are always double precision values. HLA always maintains all constants as 80-bit floating point values internally and converts them to 64 or 32 bits as necessary, so there is no need for such a suffix. So by dropping the "F" or "L" suffix, if it is present, you can use any C floating point literal constant as-is in the HLA code.

3.2.2: C and Assembly Composite Data Types

C provides four major composite (or aggregate) data types: arrays, structures, unions, and pointers. C++ adds the class types well. A composite type is a type that is built up from smaller types (e.g., an array is a collection of smaller objects, each element having the same type as all other elements in the array). In the following sub-sections, we'll explore these composite data types in C and provide the corresponding type in HLA.

As is the case throughout this chapter, this section assumes that you already know assembly language and may not know C. This section provides the correspondence between C and HLA, but doesn't attempt to teach you how to do things like access an array in assembly language; the previous chapter already covered that material.

3.2.2.1: C and Assembly Array Types

Since HLA's syntax was based on the C and Pascal programming languages, it should come as no surprise that array declarations in HLA are very similar to those in C. This makes the translation from C to HLA very easy.

The syntax for an array declaration in C is the following:

```
elementType  arrayName[ elements ] <<additional dimension infor-
mation>>;
```

elementType is the type of an individual element of the array. *arrayName* is the name of the array object. *elements* is an integer constant value the specifies the number of array elements. Here are some sample C array declarations:

```
int intArray[4];
char grid[3][3];  // 3x3 two-dimensional array
double flts[16];
userType userData[2][2][2]
```

In HLA, multiple dimension arrays use a comma-delimited list to separate each of the maximum bounds (rather than using separate sets of brackets). Here are the corresponding declarations in HLA:

```
intArray :int32[ 4 ];
grid     :char[3,3];
flts     :real64[16];
userData :userType[2,2,2];
```

Both C and HLA index arrays from zero to *n*-1, where *n* is the value specified as the array bound in the declaration.

C stores arrays in memory using row-major ordering. Therefore, when accessing elements of a multi-dimensional array, always use the row-major ordering algorithm (see *The Art of Assembly Language* for details if you're unfamiliar with accessing elements of a multi-dimensional array in assembly language).

In C, it is possible to provide an initial value for an array when declaring an array. The following C example demonstrates the syntax for this operation:

```
int iArray[4] = {1,2,3,4};
```

HLA also allows initialization of array variables, but only for static objects. Here's the HLA version of this C code:

```
static
   iArray :int32[4] := [1,2,3,4];
```

C allows the same syntax for an array initialization to automatic variables. However, you cannot initialize an automatic variable at compile time (this is true for C and HLA); therefore, the C compiler automatically emits code to copy the data from some static memory somewhere into the automatic array, e.g., a C declaration like the following appearing in a function (i.e., as an automatic local variable):

```
int autoArray[4] = {1,2,3,4};
```

gets translated into machine code that looks something like the following:

```
readonly
   staticInitializeData :dword := [1,2,3,4];
      .
      .
      .
var
   autoArray: int32[4];
      .
      .
      .
   mov( &staticInitializerData, esi );
   lea( edi, autoArray );
   mov( 4, ecx );
   rep.movsd();
```

(yes, compilers really do generate code like this). This code is pretty disgusting. If you see an automatic array variable with an initializer, it's probably a better idea to try and figure out if you really need a new copy of the initial data every time you enter the function.

3.2.2.2: C and Assembly Record/Structure Types

C implements the equivalent of HLA'S records using the `struct` keyword. Structure declarations in C look just like standard variable declarations sandwiched inside a "struct {...}" block. Conversion to HLA is relatively simple: just stick the HLA equivalent of those field declarations in a `record..endrecord` block. The only point of confusion is C's syntax for declaring tags, types, and variables of some `struct` type.

C allows you to declare structure variables and types several different ways. First, consider the following structure variable declaration:

```
   struct
   {
      int fieldA;
      float fieldB;
      char fieldC;
   }
      structVar;
```

Assuming this is a global variable in C (i.e., not within a function) then this creates a static variable `structVar` that has three fields: `structVar.fieldA`, `structVar.fieldB`, and `structVar.fieldC`. The corresponding HLA declaration is the following (again, assuming a static variable):

```
static
   structVar:
      record
         fieldA  :int32;
         fieldB  :real32;
         fieldC  :char;
      endrecord;
```

In an HLA program, you'd access the fields of the structVar variable using the exact same syntax as C, specifically: *structVar.fieldA*, *structVar.fieldB*, and *structVar.fieldC*.

There are two ways to declare structure types in C: using *typedef* and using *tag fields*. Here's the version using C's *typedef* facility (along with a variable declaration of the structure type):

```
typedef struct
   {
      int fieldA;
      float fieldB;
      char fieldC;
   }
      structType;

   structType structVar;
```

Once again, you access the fields of *structVar* as *structVar.fieldA*, *structVar.fieldB*, and *structVar.fieldC*.

The *typedef* keyword was added to the C language well after it's original design. In the original C language, you'd declare a structure type using a structure tag as follows:

```
struct structType
   {
      int fieldA;
      float fieldB;
      char fieldC;
   } /* Note: you could actually declare structType variables here */ ;

   struct structType strucVar;
```

HLA provides a single mechanism for declaring a record type - by declaring the type in HLA's type section. The syntax for an HLA record in the type section takes this form:

```
type
   structType:
      record
         fieldA  :int32;
         fieldB  :real32;
         fieldC  :char;
      endrecord;

static
   structVar  :structType;
```

C also allows the initialization of structure variables by using initializers in a variable declaration. The syntax is similar to that for an array (i.e., a list of comma separated values within braces). C associates the values in the list with each of the fields by position. Here is an example of the *structVar* declaration given earlier with an initializer:

```
   struct structType structVar = { 1234, 5.678, '9' };
```

Like the array initializers you read about in the previous section, the C compiler will initialize these fields at compile time if the variable is a static or global variable. however, if it's an automatic variable, then the C compiler emits code to copy the data into the structure upon each entry into the function defining the initialized structure (i.e., this is an expensive operation).

Like arrays, HLA allows the initialization of static objects you define in a *static, storage,* or *readonly* section. Here is the previous C example translated to HLA:

```
static
   structVar  :structType := structType:[1234, 5.678, '9' ];
```

One major difference between HLA and C with respect to structures is the alignment of the fields within the structure. By default, HLA packs all the fields in a record so that there are no extra bytes within a structure. The *structType* structure, for example, would consume exactly nine bytes in HLA (four bytes each for the *int32* and *real32* fields and one byte for the *char* field). C structures, on the other hand, generally adhere to certain alignment and padding rules. Although the rules vary by compiler implementation, most Win32 C/C++ compilers align each field of a struct on an offset boundary that is an even multiple of that field's size (up to four bytes). In the current *structType* example, *fieldA* would fall at offset zero, *fieldB* would fall on offset four within the structure, and *FieldC* would appear at offset eight in the structure. Since each field has an offset that is an even multiple of the object's size, C doesn't manipulate the field offsets for the record. Suppose, however, that we have the following C declaration in our code:

```
typedef struct
   {
      char fieldC;
      int fieldA;
      short fieldD;
      float fieldB;
   }
      structType2;
```

If C, like HLA, didn't align the fields by default, you'd find *fieldC* appearing at offset zero, *fieldA* at offset 1, *fieldD* at offset five, and *fieldB* at offset seven. Unfortunately, these fields would appear at less than optimal addresses in memory. C, however, tends to insert padding between fields so that each field is aligned at an offset within the structure that is an even multiple of the object's size (up to four bytes, which is the maximum field alignment that compilers support under Win32). This padding is generally transparent to the C programmer; however, when converting C structures to assembly language records, the assembly language programmer must take this padding into consideration in order to correctly communicate information between an assembly language program and Windows. The traditional way to handle this (in assembly language) has been to explicitly add padding fields to the assembly language structure, e.g.,

```
type
   structType2:
      record
         fieldC  :char;
         pad0    :byte[3]; // Three bytes of padding.
         fieldA  :int32;
         fieldD  :int16;
         pad1    :byte[2]; // Two bytes of padding.
         fieldB  :real32;
      endrecord;
```

HLA provides a set of features that automatically and semi-automatically align record fields. First of all, you can use the HLA *align* directive to align a field to a given offset within the record; this mechanism is great for programs that need absolute control over the alignment of each field. However, this kind of control is not necessary when implementing C records in assembly language, so a better approach is to use HLA's automatic record field alignment facilities[3]. The rules for C/C++ (under Windows, at least) are pretty standard among compilers; the rule is this: each object (up to four bytes) is aligned at a starting offset that is an even multiple of that object's size. If the object is larger than four bytes, then it gets aligned to an offset that is an even multiple of four bytes. In HLA, you can easily do this using a declaration like the following:

```
type
   structType2:
      record[4:1];
         fieldC  :char;
         pad0    :byte[3]; // Three bytes of padding.
         fieldA  :int32;
         fieldD  :int16;
         pad1    :byte[2]; // Two bytes of padding.
         fieldB  :real32;
      endrecord;
```

The "[4:1];" appendage to the record keyword tells HLA to align objects between one and four bytes on their natural boundary and larger objects on a four-byte boundary, just like C. This is the only record alignment option you should need for C/C++ code (and, in fact, you should always use this alignment option when converting C/C++ structs to HLA records). If you would like more information on this alignment option, please see the HLA reference manual.

Specifying the field alignment via the "[4:1];" option only ensures that each field in the record starts at an appropriate offset. It does not guarantee that the record's length is an even multiple of four bytes (and most C/C++ compilers will add padding to the end of a *struct* to ensure the length is an even multiple of four bytes). However, you can easily tell HLA to ensure that the record's length is an even multiple of four bytes by adding an align directive to the end of the field list in the record, e.g.,

```
type
   structType2:
      record[4:1];
         fieldC  :char;
         pad0    :byte[3]; // Three bytes of padding.
         fieldA  :int32;
         fieldD  :int16;
         pad1    :byte[2]; // Two bytes of padding.
         fieldB  :real32;
         align(4);
      endrecord;
```

Another important thing to remember is that the fields of a record or structure are only properly aligned in memory if the starting address of the record or structure is aligned at an appropriate address in memory. Specifying the alignment of the fields in an HLA record does not guarantee this. Instead, you will have to use HLA's *align* directive when declaring variables in your HLA programs. The best thing to do is ensure that an instance

3. For those who are interested in using the align directive and other advanced record field alignment facilities, please consult the HLA reference manual.

of a record (i.e., a record variable) begins at an address that is an even multiple of four bytes[4]. You can do this by declaring your record variables as follows:

```
static
   align(4);  // Align following record variable on a four-byte address in memory.
   recVar   :structType2;
```

3.2.2.3: C and Assembly Union Types

A C union is a special type of structure where all the fields have the same offset (in C, the offset is zero, in HLA you can actually select the offset though the default is zero). That is, all the fields of an instance of a union overlay one another in memory. Because the fields of a union all have the same starting address, there are no issues regarding field offset alignment between C and HLA. Therefore, all you need really do is directly convert the C syntax to the HLA syntax for a union declaration.

In C, union type declarations can take one of two forms:

```
union unionTag
{
   <<fields that look like variable declarations>>
} <<optional union variable declarations>>;

typedef union
{
   <<fields that look like variable declarations>>
}
   unionTag2; /* The type declaration */
```

You may also declare *union* variables directly using syntax like the following:

```
union
{
   << fields that look like variable declarations >>
}
   unionVariable1, unionVariable2, unionVariable3;

   union unionTag unionVar3, unionVar4;
   unionTag2 unionVar5;
```

In HLA, you declare a union type in HLA's *type* section using the *union/endunion* reserved words as follows:

```
type
   unionType:
      union
         << fields that look like HLA variable declarations >>
      endunion;
```

4. Technically, you should align a record object on a boundary that is an even multiple of the largest field's size, up to four bytes. Aligning record variables on a four-byte boundary, however, will also work.

You can declare actual HLA union variables using declarations like the following:

```
storage
   unionVar1:
      union
         << fields that look like HLA variable declarations >>
      endunion;

   unionvar2: unionType;
```

The size of a union object in HLA is the size of the largest field in the *union* declaration. You can force the size of the union object to some fixed size by placing an align directive at the end of the union declaration. For example, the following HLA type declaration defines a union type that consumes an even multiple of four bytes:

```
type
   union4_t:
      union
         b    :boolean;
         c    :char[3];
         w    :word;
         align(4);
      endunion;
```

Without the "align(4);" field in this union declaration, HLA would only allocate three bytes for a object of type *union4_t* because the single largest field is *c*, which consumes three bytes. The presence of the "align(4);" directive, however, tells HLA to align the end of the union on a four-byte boundary (that is, make the union's size an even multiple of four bytes). You'll need to check with your compiler to see what it will do with unions, but most compilers probably extend the union so that it's size is an even multiple of the largest scalar (e.g., non-array) object in the field list (in the example above, a typical C compiler would probably ensure that and instance of the *union4_t* type is an even multiple of two bytes long).

As for records, the fact that a union type is an even multiple of some size does not guarantee that a variable of that type (i.e., an instance) is aligned on that particular boundary in memory. As with records, if you want to ensure that a union variable begins at a desired address boundary, you need to stick an align directive before the declaration, e.g.,

```
var
   align(4);
   u  :union4_t;
```

3.2.2.4: C and Assembly Character String Types

The C/C++ programming language does not support a true string type. Instead, C/C++ uses an array of characters with a zero terminating byte to represent a character string. C/C++ uses a pointer to the first character of the character sequence as a "string object". HLA defines a true character string type, though it's internal representation is a little different than C/C++'s. Fortunately, HLA's string format is upwards compatible with the zero-terminated string format that C/C++ uses, so it's very easy to convert HLA strings into the C/C++ format (indeed, the conversion is trivial). There are a few problems going in the other direction (at least, at run time). So a special discussion of HLA versus C/C++ strings is in order.

Character string declarations are somewhat confused in C/C++ because C/C++ often treats pointers to an object and arrays of that same object type equivalently. For example, in an arithmetic expression, C/C++ does not differentiate between the use of a *char** object (a character pointer) and an array of characters. Because of this syntactical confusion in C/C++, you'll often see strings in this language declared one of two different ways: as an array of characters or as a pointer to a character. For example, the following are both typical string variable declarations in C/C++:

```
char stringA[ 256 ]; // Holds up to 255 characters plus a zero terminating byte.
char *stringB;       // Must allocate storage for this string dynamically.
```

The big difference between these two declarations is that the *stringA* declaration actually reserves the storage for the character string while the *stringB* declaration only reserves storage for a pointer. Later, when the program is running, the programmer must allocate storage for the string associated with *stringB* (or assign the address of some previously allocated string to *stringB*). Interestingly enough, once you have two declarations like *stringA* and *stringB* in this example, you can access characters in either string variable using either pointer or array syntax. That is, all of the following are perfectly legal given these declarations for *stringA* and *stringB* (and they all do basically the same thing):

```
char ch;

ch = stringA[i];       // Access the char at position i in stringA.
ch = *(stringB + i);   // Access the char at position i in stringB.
ch = stringB[i];       // Access the char at position i in stringB.
ch = *(stringA + i);   // Access the char at position i in stringA.
```

String literal constants in C/C++ are interesting. Syntactically, a C/C++ string literal looks very similar to an HLA string literal: it consists of a sequence of zero or more characters surrounded by quotes. The big difference between HLA string literals and C/C++ string literals is that C/C++ uses escape sequences to represent control characters and non-graphic characters within a string (as well as the backslash and quote characters). HLA does not support the escape character sequences (see the section on character constants for more details on C/C++ escape character sequences). To convert a C/C++ string literal that contains escape sequences into an HLA character string, there are four rules you need follow:

- Replace the escape sequence \" with "". HLA uses doubled-up quotes to represent a single quote within a string literal constant.
- Replace the escape sequence \\ with a single backslash. Since HLA doesn't use the backslash as a special character within a string literal, you need only one instance of it in the string to represent a single backslash character
- Convert special control-character escape sequences, e.g., \n, \r, \a, \b, \t, \f, and \v, to their corresponding ASCII codes (see Table 3-3) and splice that character into the string using HLA's #nn character literal, e.g., the C/C++ string "hello\nworld\n" becomes the following:

```
"hello" #$d #$a "world" #$d #$a    //HLA automatically splices all
this together.
```

- Whenever a C/C++ numeric escape sequence appears in a string (e.g., \0nn or \0Xnn) then simply convert the octal constant to a hexadecimal constant (or just use the hexadecimal constant as-is) along with the HLA #$nn literal constant specification and splice the object into the string as before. For example, the C/C++ string "hello\0xaworld0xa" becomes:

```
"hello" #$a "world" #$a
```

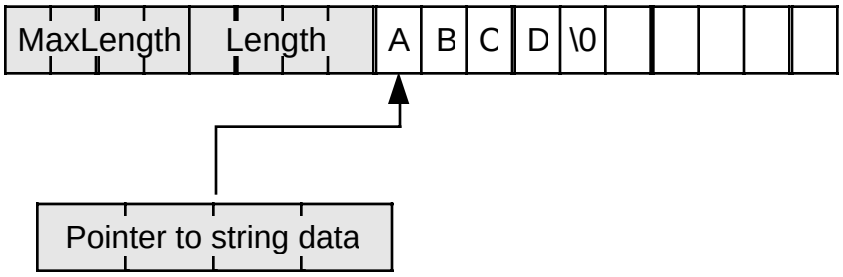
Whenever a C/C++ compiler sees a string literal constant in a source file, the compiler will allocate storage for each character in that constant plus one extra byte to hold the zero terminating byte[5] and, of course, the compiler will initialize each byte of this storage with the successive characters in the string literal constant. C/C++ compilers will typically (though not always) place this string they've created in a read-only section of memory. Once the compiler does this, it replaces the string literal constant in the program with the address of the first character it has created in this read-only memory segment. To see this in action, consider the following C/C++ code fragment:

```
char* str;

str = "Some Literal String Constant";
```

This does not copy the character sequence "Some Literal String Constant" into *str*. Instead, the compiler creates a sequence of bytes somewhere in memory, initializes them with the characters in this string (plus a zero terminating byte), and then stores the address of the first character of this sequence into the *str* character pointer variable. This is very efficient, as C/C++ needs to only move a four-byte pointer around at run-time rather than moving a 29 byte sequence.

The HLA language provides an explicit *string* data type. Internally, however, HLA represents strings using a four-byte pointer variable, just as C/C++ does. There is an important difference, however, between the type of data that an HLA string variable references and a C/C++ character pointer references: HLA includes some additional information with its string data: specifically, a length field and a maximum length field. Like C/C++ string objects, the address held in an HLA *string* variable points at the first character of the character string (and HLA strings also end with a zero terminating byte). Unlike C/C++ however, the four bytes immediately preceding the first character of the string contain the current length of the string (as an *uns32* value). The four bytes preceding the length field contain the maximum possible length of the string (that is, how much memory is reserved for the string variable, which can be larger than the current number of characters actually found in the string). Figure 3-1 shows the HLA string format in memory.

Figure 3-1: HLA String Format in Memory



The interesting thing to note about HLA strings is that they are downwards compatible with C/C++ strings. That is, if you've got a function, procedure, or some other piece of code that operates on a C/C++ string, you can generally pass it a pointer to an HLA string and the operation will work as expected[6]. This was done by design

5. Some compilers may actually allocate a few extra bytes of padding at the end of the string to ensure that the literal string constant's length is an even multiple of four bytes. However, this is not universal among compilers so don't count on it.
6. The only 'gotcha' is that you cannot expect the string to remain in a consistent HLA format if your code that operates on C/C++ strings makes any modifications to the HLA string data.

and is one of the reasons HLA interfaces so well with the Win32 API. The Win32 API generally expects C/C++ zero terminated strings when you pass it string data. Usually, you can pass an HLA string directly to a Win32 API function and everything will work as expected (few Win32 API function calls actually change the string data).

Going the other direction, converting a C/C++ string to an HLA string, is not quite as trivial (though still easy to accomplish using appropriate code in the HLA Standard Library). Before describing how to convert C/C++ strings to the HLA format, however, it *is* important to point out that HLA is assembly language and assembly language is capable of working with any string format you can dream up (including C/C++ strings). If you've got some code that produces a zero-terminated C/C++ string, you *don't* have to convert that string to an HLA string in order to manipulate it within your HLA programs. Although the HLA string format is generally more efficient (faster) because the algorithms that process HLA strings can be written more efficiently, you can write your own zero-terminated string functions and, in fact, there are several common zero-terminated string functions found in the HLA Standard Library. However, as just noted, HLA strings are generally more efficient, so if you're going to be doing a bit of processing on a zero terminated string, it's probably wise to convert it to an HLA string first. On the other hand, if you're only going to do one or two trivial operations on a zero-terminated string that the Win32 API returns, you're better off manipulating it directly as a zero-terminated string and not bothering with the conversion (of course, if convenience is your goal rather than efficiency, it's probably always better to convert the string to the HLA string format upon return from a Win32 API call just so you don't have to work with two different string formats in your assembly code). Of course, the other option open to you is to work exclusively with zero-terminated strings in your assembly code (though the HLA Standard Library doesn't provide anywhere near the number of zero-terminated string functions).

The purpose of this chapter is to explain how to convert C/C++ based Windows documentation to a form suitable for assembly language programmers and describe how assembly language programmers can call the C-based functions that make up the Win32 API set. Therefore, we won't get into the details of converting string function calls in C/C++ to their equivalent (or similar) HLA Standard Library calls. Rest assured, the HLA Standard Library provides a much richer set of string functions than does the C Standard Library; so anything you find someone doing in C/C++ can be done just as easily (and usually more efficiently) using an HLA Standard Library function. Please consult the HLA Standard Library documentation for more details. In this section, we'll simply cover those routines that are necessary for converting between the two different string formats and copying string data from one location to another.

In C/C++ code, it is very common to see the program declare a string object (character pointer) and initialize the pointer with the address of a literal string constant all in one operation, e.g.,

```
char *str = "Literal String Constant";
```

Remember, this doesn't actually copy any character data; it simply loads the character pointer variable `str` with the address of the 'L' character in this string literal constant. The compiler places the literal string data somewhere in memory and either initializes `str` with the address of the start of that character data (if `str` is a global or static variable) or it generates a short (generally one instruction) machine code sequence to initialize a run-time variable with the address of this string constant. Note, *and this is very important*, many compilers will allocate the actual string data in read-only memory. Therefore, as long as `str` continues to point at this literal string constant's data in memory, any attempt to store data into the string will likely produce a memory protection fault. Some very old C programs may have made the assumption that literal string data appears in read/write memory and would overwrite the characters in the string literal constant at run-time. This is generally not allowed by modern C/C++ compilers. HLA, by default, also places string literal data in write-protected memory. So if you encounter some older C code that overwrites the characters in a string literal constant, be aware of the fact that you will not (generally) be able to get away with this in HLA.

If *str* (in the previous example) is a static or global variable, converting this declaration to assembly language is very easy, you can use a declaration like the following:

```
static
   HLAstr  :string := "Literal String Constant";
```

Remember, like C/C++, HLA implements string objects using pointers. So *HLAstr* is actually a four-byte pointer variable and this code initializes that pointer with the address of the first character in the literal string constant.

Because HLA string variables are pointers that (when properly initialized) point at a sequence of characters that end with a zero byte (among other things), you can use an HLA string variable just about anywhere C/C++ expects a string object. So, for example, if you want to make some Win32 API call that requires a string parameter, you can pass in *HLAstr* as that parameter, e.g.,

```
Win32APIcall( HLAstr );
```

Of course, you don't have to assign the address of a string literal constant to an HLA string variable to make this function call, HLA allows you to pass the literal string constant directly:

```
Win32APIcall( "Literal String Constant" );
```

Note that this call does not pass the string data directly to the function; like C/C++, HLA continues to allocate and initialize the literal string constant in write-protected memory somewhere else and simply passes the address of the literal string constant as the function's parameter.

For actual string variable objects (that is, strings whose character values you can change at run-time), you'll typically find two different ways of variable declaration/allocation in a C/C++ program: the programmer will either create a character array with sufficient storage to hold the characters in the string, or the programmer will simply declare a character pointer variable and allocate storage for the string at run-time using a function like *malloc* (C) or new (C++). We'll look at both of these mechanisms in HLA in the following paragraphs.

One way to allocate storage for a C string variable is to simply declare a character array with sufficient room to hold the longest possible string value you will assign plus an extra byte for the zero terminator. To do this, a C/C++ programmer simply declares a character array with n+1 elements, where n is the maximum number of characters they expect to put into the string. The following is a typical "string" declaration in C/C++ that creates a string capable of holding up to 255 characters:

```
char myStr[256];  // 255 chars plus a zero terminating byte
```

Although HLA most certainly allows you to create arrays of characters, such arrays are not directly compatible with HLA strings (because they don't reserve extra storage prior to the string object for the length and maximum length fields that are present in the HLA string format). Therefore, you cannot allocate storage for an HLA string variable by simply doing the following:

```
static
      myStr :char[256];  // Does NOT create an HLA string!
```

You *can* use this technique to pre-allocate storage for a zero-terminated string in HLA, but this does not create the proper storage for an HLA string variable.

In general, HLA expects you to allocate storage for string objects dynamically (that is, at run-time). However, for static strings (i.e., non-automatic string variables) the HLA Standard Library does provide a macro,

str.strvar, that lets you declare string variable and allocate storage for the string variable at compile-time. Here is an example of the use of this macro:

```
static  //Note: str.strvar only makes sense in the STATIC section
   preAllocated :str.strvar( 255 ); // Allows strings up to 255 characters long.
```

There are a couple of important things to note about this declaration. First of all, note that you surround the size of the string with parentheses, not square brackets. This is because *str.strvar* is actually a macro, not a type and the 255 that appears in this example is a macro argument. The second thing to note is that the use of the *str.strvar* macro only makes sense in the static declaration section. It is illegal in a *var* or *storage* section because this macro initializes the variable you're declaring (something you can't do in a *var* or *storage* declaration section). While the use of the *str.strvar* macro is technically legal in HLA's *readonly* declaration section, it doesn't make sense to use it there since doing so would allocate the string data storage in write-protected memory, so you'd never be able to store any character data into the string. Also note that the *str.strvar* macro doesn't provide any mechanism for initializing the character data in the string within the declaration (not that it would be that hard to create a new macro that does this). Finally, note that you specify the maximum number of characters you want to allow in the string as the *str.strvar* argument. You do not have to account for any zero-terminating bytes, the current string length field, or the maximum length field.

Although the *str.strvar* macro is very convenient to use, it does have a couple of serious limitations. Specifically, it only makes sense to use it in the *static* section of an HLA program and you must know the maximum length of the string before you can use the *str.strvar* macro to declare a string object. Both of these issues are easy to resolve by using dynamically allocated string objects. C/C++ programmers can also create dynamically allocated strings by declaring their string variables as pointers to characters, e.g.,

```
   char *strAsPtr;
      .
      .
      .
   // Allocate storage for a 255 character string at run-time

   strAsPtr = malloc( 256 );  // 255 chars + zero terminating byte

   // Or (in C++):

   strAsPtr = new char[256];
```

In HLA, the declaration and allocation is very similar:

```
var  // Could also be static or storage section
   strAsPtr :string;
      .
      .
      .
   strmalloc( 255 );
   mov( eax, strAsPtr );
```

Do note a couple of things about the HLA usage. First, you should note that the specify the actual number of characters you wish to allow in the string; you don't have to worry about the zero terminating byte. Second, you should call the HLA Standard Library *stralloc* function (rather than *malloc*, which HLA also provides) in order to allocate storage for a string at run time; in addition to allocating storage for the character data in the string, *stralloc* also allocates (additional) storage for the zero terminating byte, the maximum length value, and

the current length value. The *stralloc* function also initializes the string to the empty string (by setting the current length to zero and storing a zero in the first character position of the string data area). Finally, note that stralloc returns the address of the first character of the string in EAX so you must store that address into the *string* variable. HLA does support a feature known as instruction composition, so you could actually write these two instructions as follows:

```
mov( stralloc( 255 ), strAsPtr );
```

The drawback to this form, however, is that it hides the fact that this code sequence wipes out the EAX register. So be careful if you decide to use this form.

Once you've initialized an HLA string variable so that it points at valid string data, you can generally pass that string variable to a C/C++ function (i.e., a Win32 API function) that expects a string value. The only restriction is that if the C/C++ function modifies the string data in such a way that it changes the length of the string, HLA will not recognize the change if you continue to treat the string as an HLA type string upon return from the C/C++ function. Likewise, if a C/C++ function returns a pointer to a string that the function has created itself, your HLA functions cannot treat that string as an HLA string because it's probably just a zero-terminated string. Fortunately, there are only a small number of Win32 API functions that return a string or modify an existing string, so you don't have to worry about this very often. However, that's a problem in and of itself; you may have to deal with this problem so infrequently that it slips your mind whenever you call a function that returns such a string. As a concrete example, consider the following (HLA) prototype for the Win32 *GetFullPathName* function:

```
static
   GetFullPathName: procedure
   (
          lpFileName    : string;
          nBufferLength : dword;
      var lpBuffer      : var;
      var lpFilePart    : var
   );
   @stdcall; @returns( "eax" ); @external( "__imp__GetFullPathNameA@16" );
```

This function, given the address of a zero-terminated string containing a filename (*lpFileName*), the length length of a buffer (*nBufferLength*), the address of a buffer (*lpBuffer*), and the address of a pointer variable (*lpFilePart*) will convert the filename you pass in to a full drive letter/path/filename string. Specifically, this function returns a string in the buffer whose address you specify in the *lpbuffer* parameter. However, this function does not create an HLA-compatible string; it simply creates a zero-terminated string.

Fortunately, most functions in the Win32 API that return string data do two things for you that will help make your life easier. First of all, Win32 API functions that store string data into your variables generally require that you pass a maximum length for the string (e.g., the *nBufferLength* parameter in the prototype above). The function will not write any characters to the buffer beyond this maximum length (doing so could corrupt other data in memory). The other important thing this function does (which is true for most Win32 API functions that return string data) is that it returns the length of the string in the EAX register; the function returns zero if there was some sort of error. Because of the way this function works, converting the return result to an HLA string is nearly trivial. Consider the following call to *GetFullPathName*:

```
static
   fullName :string;
   namePtr  :pointer to char;
      .
      .
```

```
    .
    stralloc( 256 );       // Allocate sufficient storage to hold the string data.
    mov( eax, fullName );
    .
    .
    .
    mov( fullName, edx );
    GetFullPathName
    (
       "myfile.exe",                          // File to get the full path for.
       (type str.strRec [edx]).MaxStrLen,     // Maximum string size
       [edx],                                 // Pointer to buffer
       namePtr                                // Address of base name gets stored here
    );
    mov( fullName, edx );                     // Note: Win32 calls don't preserve EDX
    mov( eax, (type str.strRec [edx]).length  // Set the actual string length
```

The nice thing about most Win32 API calls that return string data is that they guarantee that they won't overwrite a buffer you pass in (you also pass in the maximum string length, which is available in the *MaxStrLen* field of the string object, that is, at offset -8 from the string pointer). These string functions also return the string length as the function's result, so you can shove this into the HLA string's *length* field (at offset -4 from the string's base address) immediately upon return. This is a very efficient way to convert C/C++ strings to HLA format strings upon return from a function.

Of course, converting a C/C++ string to an HLA string is only easy if the C/C++ function you're calling returns the length of the string it has processed. It also helps if the function guarantees that it won't overstep the bounds of the string variable you've passed it (i.e., it accepts a *MaxStrLen* parameter and won't write any data beyond the maximum buffer size you've specified). Although most Win32 API functions that return string data operate this way (respect a maximum buffer size and return the length of the actual string), there are many C/C++ functions you may need to call that won't do this. In such a case, you've got to compute the length of the string yourself (and guarantee that your character buffer is large enough to hold the maximum possible string the function will produce). Fortunately, there is a function in the HLA Standard Library, str.zlen, that will compute the length of a zero terminated string so you can easily update the length field of an HLA string object that a C/C++ function has changed (without respect to the HLA string's length field). For example, suppose you have a C/C++ function named *fstr* that expects the address of a character buffer where it can store (or modify) a zero-terminated string. Since HLA strings are zero-terminated, you can pass an HLA string to this function. However, if *fstr* changes the length of the string, the function will not update the HLA string's *length* field and the result will be inconsistent. You can easily correct this by computing the length of the string yourself and storing the length into the HLA string's *length* field, e.g.,

```
static
    someStr  :str.strvar( 255 );  // Allow strings up to 255 characters in length.
    .
    .
    .
    fstr( someStr );     // Assume this changes the length of someStr.

    mov( someStr, edx ); // Get the pointer to the string data structure.
    str.zlen( edx );     // Compute the new length of someStr.
    mov( eax, (type str.strRec [edx]).length );  // Update the length field.
```

One thing to keep in mind is that *str.zlen* has to search past every character in the string to find the zero-terminating byte in order to compute the string's length. This is not a particularly efficient operation, particularly if the

string is long. Although *str.zlen* uses a fairly decent algorithm to search for the terminating zero byte, the amount of time it takes to execute is still proportional to the length of the string. Therefore, you want to avoid calling *str.zlen* if at all possible. Fortunately, many C/C++ string functions (that modify string data) return the length as the function result, so calling *str.zlen* isn't always necessary.

Although not explicitly shown in the example above, do keep in mind that many string functions (especially Win32 API functions) assign double-duty to the function return result; they'll return a positive value if the function successfully produces a string and they'll return a negative result (or sometimes a zero result) if there was an error. For example, the *GetFullPathName* function in the earlier example returns zero if it there was a problem producing the string. You code should check for errors on return from these functions to prevent problems. While shoving a zero into a string length field isn't cause for concern (indeed, that's perfectly reasonable), a negative number will create all kinds of problems (since the HLA string functions treat the length field as an *uns32* value, those functions will interpret a negative number as a really large positive value).

3.2.2.5: C++ and Assembly Class Types

C++ and HLA both support classes. Since HLA is an assembly language, it should be obvious that anything you can do with a C++ class you can do with in HLA (since C++ compilers convert C++ source code into x86 machine code, it should be obvious that you can achieve anything possible in C++ using assembly language). However, directly translating C++ classes into HLA classes is not a trivial matter. Many constructs transfer across directly. Some constructs in C++, however, do not have a direct correspondence in HLA. Quite honestly, the conversion of C++ classes to HLA classes is beyond the scope of this book; fortunately, we're dealing with the Win32 API in this book so we won't see much Windows source code that uses classes in typical documentation we're interested it. There is quite a bit of C++-based object-oriented code out there for Windows but most of it uses the Microsoft Foundation Classes (MFC) class library, and that's not applicable to assembly language programming (at least, not today; someday there might be an "Assembly Language Foundation Class" library but until that day arrives we don't have to worry about MFC). Since this book covers Win32 API programming, there really is no need to worry about converting C++ classes into assembly - the Win32 API documentation doesn't use classes. This book may very well use HLA classes and objects in certain programming examples, but that will be pure assembly language, not the result of translating C++ code into HLA code.

3.2.3: C and Assembly Pointer Types[7]

When running under 32-bit versions of the Windows operating system, C pointers are always 32-bit values and map directly to 32-bit flat/linear machine addresses. This is true regardless of what the pointer references (including both data and functions in C). At one level, the conversion of a pointer from C to HLA/assembly is fairly trivial, just create a *dword* variable in you HLA program and keep the pointer there. However, HLA (like C) supports typed pointers and procedure pointers; generally, it's a good idea to map C pointers to their HLA typed-pointer equivalent.

In C, you can apply the address-of operator, '&' to an object (e.g., a static variable or a function name) to take the address of that object. In HLA, you may use this same operator to take the address of a *static* object (a *static/storage/readonly* variable, statement label, or procedure name). The result is a 32-bit value that you may load into a register or 32-bit memory variable. For example, if in C you have a statement like the following (*pi* is a pointer to an integer and *i* is a static integer variable):

7. Many programmers (and languages) consider pointers to be a scalar type because the value is not composed of other types. This book treats pointers as a separate classification simply because it has to discuss both scalar and composite data types before being able to discuss pointers.

```
pi = &i;
```

You convert this to HLA syntax as follows:

```
mov( &i, pi );
```

If the object you're taking the address of with the '&' operator is not a function name or a static (non-indexed) variable, then you must compute the address of the object at run-time using the *lea* instruction. For example, automatic variables (non-static local variables) fall into this class. Consider the following C function fragment:

```
int f( void )
{
   int *pi;   // Declares a variable that is a pointer to an integer.
   int i;     // Declares a local integer variable using automatic allocation.

   pi = &i;
      .
      .
      .
}
```

Because *i* is an automatic variable (the default storage class for local variables in a function) HLA cannot static compute this address for you at compile time. Instead, you would use the *lea* instruction as follows:

```
procedure f;
var
   pi: pointer to int32;  // Declare HLA pointers this way.
   i:  int32;             // Declares an automatic variable in HLA (in the VAR section)
      .
      .
      .
   lea( eax, i );    // Compute the run-time address of i
   mov( eax, pi );   // Save address in pi.
      .
      .
      .
```

You should also note that you need to use the *lea* instruction when accessing an indexed object, even if the base address is a static variable. For example, consider the following C/C++ code that takes the address of an array element:

```
int *pi;
static int array[ 16 ];
   .
   .
   .
pi = &array[i];
```

In order to access an array element in assembly language, you will need to use a 32-bit register as an index register and compute the actual element address at run-time rather than at compile-time (assuming that *i* in this example is a variable rather than a constant). Therefore, you cannot use the '&' operator to statically compute the address of this array element at compile-time, instead you should use the *lea* instruction as follows:

```
mov( i, ebx );               // Move index into a 32-bit register.
lea( eax, array[ ebx*4 ] );  // int objects are four bytes under Win32
```

```
mov( eax, pi );
```

As you've seen in these simple examples, C uses the unary '*' operator to declare pointer objects. Anytime you see a declaration like the following in C:

```
typename *x;
```

you can convert it to a comparable HLA declaration as follows:

```
x:pointer to hla_typename;  // hla_typename is the HLA equivalent of C's typename
```

Of course, since all pointers are simply 32-bit objects, you can also convert to assembly language using a statement like the following:

```
x:dword;
```

Both forms are equivalent in assembly language, though the former version is preferable since it's a little more descriptive of what's going on here.

Function pointers in C is one area where the C syntax can get absolutely weird. A simple C function pointer declaration takes the following form:

```
returnType (*functionPtrName) ( parameters );
```

For example,

```
int (*ptrToFuncReturnsInt) ( int i, int j );
```

This example declares a pointer to a function that takes two integer parameters and returns an integer value. Note that the following is *not* equivalent to this example:

```
int *ptrToFuncReturnsInt( int i, int j );  //Not a function pointer declaration!
```

This example is a prototype for a function that returns a pointer to an integer, not a function pointer. C's syntax is a little messed up (i.e., it wasn't thought out properly during the early stages of the design), so you can get some *real* interesting function pointer declarations; some are nearly indecipherable.

Of course, at the assembly language level all pointers are just `dword` variables. So all you really need to implement the C function pointer in HLA is a statement like the following:

```
ptrToFuncReturnsInt: dword;
```

You can all this function in HLA using the `call` instruction as follows:

```
call( ptrToFuncReturnsInt );
```

As for data pointers, however, HLA provides a better solution: procedure variables. A procedure variable is a pointer object that (presumably) contains the address of some HLA procedure. The advantage of a procedure variable over a `dword` variable is that you can use HLA's high-level syntax calling convention with procedure

variables; something you cannot do with a *dword* variable. Here's an example of a procedure variable declaration in HLA:

```
ptrToFuncReturnsInt: procedure( i:int32; j:int32 );
```

HLA allows you to call the code whose address this variable contains using the standard HLA procedure call syntax, e.g.,

```
ptrToFuncReturnsInt( 5, intVar );
```

To make this same call using a *dword* variable[8], you'd have to manually pass the parameters yourself as follows (assuming you pass the parameters on the stack):

```
push( 5 );       // Pass first parameter (i)
push( intVar );  // Pass second parameter.
call( ptrToFuncReturnsInt );
```

In C, any time a function name appears without the call operator attached to it (the "call operator" is a set of parenthesis that may contain optional parameters), C substitutes the address of the function in place of the name. You do not have to supply the address-of operator ('&') to extract the function's address (though it is legal to go ahead and do so). So if you see something like the following in a C program:

```
ptrToFuncReturnsInt = funcReturnsInt;
```

where *funcReturnsInt* is the name of a function that is compatible with *ptrToFuncReturnsInt's* declaration (e.g., it returns an integer result and has two integer parameters in our current examples), this code is simply taking the address of the function and shoving it into *ptrToFuncReturnsInt* exactly as though you'd stuck the '&' operator in front of the whole thing. In HLA, you can use the '&' operator to take the address of a function (they are always static objects as far as the compiler is concerned) and move it into a 32-bit register or variable (e.g., a procedure variable). Here's the code above rewritten in HLA:

```
mov( &funcReturnsInt, ptrToFuncReturnsInt );
```

Both C and HLA allow you to initialize static variables (including pointer variables) at compile time. In C, you could do this as follows:

```
static int (*ptrToFuncReturnsInt) ( int i, int j) = funcRe-
turnsInt;
```

The comparable statement in HLA looks like this:

```
procedure funcReturnsInt( i:int32; j:int32 );
begin funcReturnsInt;
      .
      .
      .
end funcReturnsInt;
      .
      .
      .
static
```

8. Actually, this calling scheme works for HLA procedure variables, too.

```
   ptrToFuncReturnsInt:procedure( i:int32; j:int32 ) := &funcReturnsInt;
```

Especially note that in HLA, unlike C, you still have to use the '&' operator when taking the address of a function name.

Note that you cannot initialize HLA automatic variables (*var* variables) using a statement like the one in this example. Instead, you must move the address of the function into the pointer variable using the *mov* instruction given a little earlier.

Arrays are another object that C treats specially with respect to pointers. Like functions, C will automatically supply the address of an array if you specify the name of an array variable without the corresponding index operator (the square brackets). HLA requires that you explicitly take the address of the array variable. If the array is a static object (static/readonly/storage) then you may use the (static) address-of operator, '&'; however, if the variable is an automatic (*var*) object, then you have to take the address of the object at run-time using the *lea* instruction:

```
static
   staticArray  :byte[10];
var
   autoArray    :byte[10];
      .
      .
      .
   mov( &staticArray, eax );   // Can use '&' on static objects
   lea( ebx, autoArray );      // Must use lea on automatic (VAR) objects.
```

C doesn't automatically substitute the address of a structure or union whenever it encounters a struct or union variable. You have to explicitly take the address of the object using the '&' operator. In HLA, taking the address of a structure or union operator is very easy - if it's a static object you can use the '&' operator, if it's an automatic (*var*) object, you have to use the lea instruction to compute the address of the object at run-time, just like other variables in HLA.

There are a couple of different ways that C/C++ allows you to dereference a pointer variable[9]. First, as you've already seen, to dereference a function pointer you simply "call" the function pointer the same way you would directly call a function: you append the call operator (parenthesis and possible parameters) to the pointer name. As you've already seen, you can do the same thing in HLA as well. Technically, you could also dereference a function pointer in C/C++ as follows:

```
(*ptrToFuncReturnsInt)( 5, intVar );
```

However, you'll rarely see this syntax in an actual C/C++ source file. You may convert an indirect function call in C/C++ to either HLA's high-level or low-level syntax, e.g., the following two calls are equivalent:

```
      push( 5 );
      push( intVar );
      call( ptrToFuncReturnsInt );

// -or-

      ptrToFuncReturnsInt( 5, intVar );
```

9. Dereferencing means to access the data pointed at by a pointer variable.

Dereferencing a pointer to a data object is a bit more exciting in C/C++. There are several ways to dereference pointers depending upon the underlying data type that the pointer references. If you've got a pointer that refers to a scalar data object in memory, C/C++ uses the unary '*' operator. For example, if *pi* is a pointer that contains the address of some integer in memory, you can reference the integer value using the following C/C++ syntax:

```
*pi = i+2;  // Store the sum i+2 into the integer that pi points at.
j = *pi;    // Grab a copy of the integer's value and store it into j.
```

Converting an indirect reference to assembly language is fairly simple. The only gotcha, of course, is that you must first move the pointer value into an 80x86 register before dereferencing the pointer. The following HLA examples demonstrate how to convert the two C/C++ statements in this example to their equivalent HLA code:

```
// *pi = i + 2;

mov( pi, ebx );    // Move pointer into a 32-bit register first!
mov( i, eax );     // Compute i + 2 and leave sum in EAX
add( 2, eax );
mov( eax, [ebx] ); // Store i+2's sum into the location pointed at by EBX.

// j = *pi;

mov( pi, ebx );    // Only necessary if EBX no longer contains pi's value!
mov( [ebx], eax ); // Only necessary if EAX no longer contains *pi's value!
mov( eax, j );     // Store the value of *pi into j.
```

If you've got a pointer that holds the address of a sequence of data values (e.g., an array), then there are two completely different (but equivalent) ways you can indirectly access those values. One way is to use C/C++'s *pointer arithmetic syntax*, the other is to use *array syntax*. Assuming *pa* is a pointer to an array of integers, the following example demonstrates these two different syntactical forms in action:

```
*(pa+i) = j;   // Stores j into the ith object beyond the address held in pa.
pa[i] = j;     // Stores j into the ith element of the array pointed at by pa.
```

The important thing to note here is that both forms are absolutely equivalent and almost every C/C++ compiler on the planet generates exactly the same code for these two statements. Since compilers generally produce exactly the same code for these two statements, it should come as no surprise that you would manually convert these two statements to the same assembly code sequence. Conversion to assembly language is slightly complicated by the fact that you must remember to multiply the index into an array (or sequence of objects) by the size of each array element. You might be tempted to convert the statements above to something like the following:

```
mov( j, XXX );          // XXX represents some register that will hold j's value.
mov( pa, ebx );         // Get base address of array/sequence in memory
mov( i, ecx );          // Grab index
mov( XXX, [ebx][ecx] ); // XXX as above
```

The problem with this sequence is that it only works properly if each element of the array is exactly one byte in size. For larger objects, you must multiply the index by the size of an array element (in bytes). For example, if each element of the array is six bytes long, you'd probably use code like the following to implement these two C/C++ statements:

```
mov( j, eax );            // Get the L.O. four bytes of j.
mov( j[4], dx );          // Get the H.O. two bytes of j.
mov( pa, ebx );           // Get the base address of the array into EBX
mov( i, ecx );            // Grab the index
intmul( 6, ecx );         // Multiply the index by the element size (six bytes).
mov( eax, [ebx][ecx] );   // Store away L.O. four bytes
mov( dx, [ebx][ecx][4] ); // Store away H.O. two bytes.
```

Of course, if the size of your array elements is one of the four magic sizes of one, two, four, or eight bytes, then you don't need to do an explicit multiplication. You can get by using the 80x86 scaled indexed addressing mode as the following HLA example demonstrates:

```
mov( j, eax );
mov( pa, ebx );           // Get base address of array/sequence in memory
mov( i, ecx );            // Grab index
mov( eax, [ebx][ecx*4] ); // Store j's value into pa[i].
```

C uses yet another syntax when accessing fields of a structure indirectly (that is, you have a pointer to some structure in memory and you want to access a field of that structure via the pointer). The problem is that C's unary '*' (dereference) operator has a lower precedence than C's '.' (field access) operator. In order to access a field of some structure to which you have a pointer, you'd have to write an expression like the following when using the '*' operator:

```
(*ptrToStruct).field
```

Avoid the temptation to write this as follows:

```
*ptrToStruct.field
```

The problem with this latter expression is that '.' has a higher precedence than '*', so this expression tells the C compiler to dereference the thing that `ptrToStruct.field` points at. That is, `ptrToStruct` must be an actual struct object and it must have a field, `field`, that is a pointer to some object. This syntax indirectly references the value whose address `field` contains. An expression of the form "*(ptrToStruct).field" tells the compiler to first dereference the pointer `ptrToStruct` and then access `field` at the given offset from that indirect address.

Because accessing fields of a structure object indirectly is a common operation, using the syntax "(*ptrToStruct).field" tends to clutter up a program and make it less readable. In order to reduce the clutter the C/C++ programming language defines a second dereferencing operator that you use specifically to access fields of a structure (or union) via a pointer: the "->" operator. The "->" operator has the same precedence as the field selection operator (".") and they are both left associative. This allows you to write the following expression rather than the ungainly one given earlier:

```
ptrToStruct->field
```

Regardless of which syntax you find in the C/C++ code, in assembly language you wind up using the same code sequence to access a field of a structure indirectly. The first step is to always load the pointer into a 32-bit register and then access the field at some offset from that indirect address. In HLA, the best way to do this is to coerce an indirect expression like "[ebx]" to the structure type (using the "(type *XXX* [ebx])" syntax) and then use the "." field reference operator. For example,

```
type
   Struct:
      record
         field  :int32;
```

```
            .
            .
            .
      endrecord;
         .
         .
         .
static
   ptrToStruct :pointer to Struct;
      .
      .
      .
   // Access field "field" indirectly via "ptrToStruct"

   mov( ptrToStruct, ebx );
   mov( (type Struct [ebx]).field, eax );
```

For more details on the HLA type coercion operator, please consult the HLA language reference manual. Note that you use this same technique to indirectly access fields of a *union* or a *class* in HLA.

You may combine structure, array, and pointer types in C/C++ to form *recursive* and *nested* types. That is, you can have an array of structs, a struct may contain a field that is an array (or a struct), or you could even have a structure that has a field that is an array of structs whose fields are arrays of pointers to structs whose fields... In general, a C/C++ programmer can create an arbitrarily complex data structure by nesting array, struct, union, class, and pointer data types. Translating such objects into assembly language is equally complex, often taking a half dozen or more instructions to access the final object. Although such constructs rarely appear in real-world C/C++ programs, they do appear every now an then, so you'll need to know how to translate them into assembly language.

Although a complete description of every possible data structure access conversion would require too much space here, an example that demonstrates the process you would go through is probably worthwhile. For our purposes, consider the following complex C/C++ expression:

```
per->field[i].p->x[j].y
```

This expression uses three separate operators: "->", "[ ]", and ".". These three operators all have the same precedence and are left associative, so we process this expression strictly on a left-to-right basis[10]. The first object in this expression, *per*, is a pointer to some structure. So the first step in the conversion to HLA is to get this pointer value into a 32-bit register so the code can refer to the indirect object:

```
mov( per, ebx );
```

The next step is to access a field of the structure that *per* references. In this example, *field* is an array of structures and the code accesses element *i* of this array. To access this particular object, we need to compute the index into the array (by multiplying the index by the size of an array element). Assuming *field* is an array of *fieldpiece* objects, you'd use code like the following to reference the i^{th} object of *field*:

```
mov( i, ecx );                      // Get the index into the field array.
intmul( @size( fieldpiece ), ecx );  // Multiply index by the size of an element
```

10. A later section in this chapter will discuss C/C++ operator precedence and associativity. Please see that section for more details concerning operator precedence and associativity.

The next step in the conversion of this expression is to access the *p* field of the i^{th} array element of *field*. The following code does this:

```
mov( (type ptrsType[ebx]).p[ecx], edx );
```

The interesting thing to note here is that the index into the field array is tacked on to the end of the HLA address expression we've created, i.e., we write "(type ptrsType[ebx]).p[ecx]" rather than "(type ptrsType[ebx])[ecx].p". This is done simply because HLA doesn't allow this latter syntax. Because the "." and "[ ]" operators both involve addition and addition is commutative, it doesn't matter which syntax we use. Note that HLA would allow an address expression of the form "(type ptrsType [ebx][ecx]).p" but this tends to (incorrectly) imply that EBX points at an array of pointers, so we'll not use this form here[11].

The array element that "(type ptrsType [ebx]).p[ecx]" access is a pointer object. Therefore, we have to move this pointer into a 32-bit register in order to dereference the pointer. That's why the previous HLA statement moved this value into the EDX register. Now this pointer points at an array of array objects (the x field) and the code then accesses the j^{th} element of this array (of structures). To do this, we can use the following HLA statements:

```
mov( j, esi );                  // Get the index into the array.
intmul( @size( xType ), esi );  // Multiply by the size of an array element.

// Now [edx][esi] references the jth element of the x field
```

The last step in our code sequence is to reference the *y* field of the j^{th} element of the *x* array. Assuming that *y* is a double-word object (just to make things easy for this example), here's the code to achieve this:

```
mov( (type xType [edx]).y[esi], eax );
```

Here's the complete code sequence:

```
// ptr->field[i].p->x[j].y

    mov( ptr, ebx );
    mov( i, ecx );                      // Get the index into the field array.
    intmul( @size( fieldType ), ecx );  // Multiply index by the size of an element
    mov( (type ptrsType [ebx]).p[ecx], edx );
    mov( j, esi );                  // Get the index into the array.
    intmul( @size( xType ), esi );  // Multiply by the size of an array element.
    mov( (type xType [edx]).y[esi], eax );
```

3.2.4: C and Assembly Language Constants

Although we've already looked at literal constants in C/C++ and assembly language, we must still consider symbolic constants in C/C++ and their equivalents in assembly language. A *symbolic constant* is one that we refer to by name (an identifier) rather than the constant's value (i.e., a literal constant). Syntactically, a symbolic constant is an identifier and you use it as you would a variable identifier; semantically of course, there are certain limitations on symbolic constants (such as you cannot assign the value of an expression to a symbol constant). There are two types of symbolic constants you'll find in the C and C++ languages: *manifest constants* and *storage constants*.

11.The implication is only visual. This form is completely equivalent to the previous form since addition is *still* commutative.

The first type of constant to consider is a *manifest constant*. A manifest constant is a symbolic identifier to which you've bound (assigned) some value. During compilation, the compiler simply substitutes the actual value of the constant everywhere it encounters the manifest constant's identifier. From the standpoint of the compiler's code generation algorithms, there is no difference between a manifest constant and a literal constant. By the time the code generator sees the constants, it's a literal value. In C/C++, you can use the `#define` preprocessor directive to create manifest constants. Note that you can use a manifest constant anywhere a literal constant is legal.

There are two ways to convert a C/C++ manifest constant into assembly language. The first way, of course, is to manually do the translation from symbolic to literal constant. That is, whenever you encounter a manifest constant in C/C++, you simply translate it to the corresponding literal constant in your assembly language code. E.g.,

```
#define someConst 55
      .
      .
      .
   i = someConst;
```

in assembly language becomes:

```
   mov( 55, i );
```

Obviously, manually expanding manifest constants when converting C/C++ code into assembly language is not a good idea. The reasons for using symbolic constants in C/C++ apply equally well to assembly language programs. Therefore, the best solution is to keep the constants symbolic when translating the C/C++ code to assembly language. In HLA, the way to create manifest constants is by using the `const` declaration section. The exact form of the translation depends how the C/C++ code uses the `#define` preprocessor directive. Technically, the `#define` preprocessor directive in C/C++ doesn't define a constant, it defines a macro. There are two basic forms of the `#define` directive:

```
#define someID  some text...

#define someID(parameter list) some text...
```

We'll not consider the second form here, since that's a true macro declaration. We'll return to macros and how to convert this second example to HLA a little later in this chapter.

The first `#define` directive in this example defines a *textual substitution macro*. The C/C++ preprocessor will substitute the text following `someID` for each occurrence of `someID` appearing after this declaration in the source file. Note that the text following the declaration can be *anything*, it isn't limited to being a literal constant. For the moment, however, let's just consider the case where the text following the `#define` directive and the identifier is a single literal constant. This begin the case, you can create an equivalent HLA manifest constant declaration using code like the following:

```
// #define fiftyFive 55

const
   fiftyFive := 55;
```

Like C/C++, HLA will substitute the literal value 55 for the identifier `fiftyFive` everywhere it appears in the HLA source file.

There is a subtle difference between HLA manifest constants you define in a *const* section and manifest constants you define with C/C++'s *#define* directive: HLA's constants are typed while C/C++ *#define* constants are untyped textual substitutions. Generally, however, you will not notice a difference between the two. However, one special case does deserve additional comment: the case where a program specifies a constant expression rather than a single literal constant (which both languages allow). Consider the following statements in C/C++ and in HLA:

```
// C/C++ constant expression:

#define constExpr i*2+j

// HLA constant expression:

const
   constExpr := i*2+j;
```

The difference between these two is that the C/C++ preprocessor simply saves up the text "i*2+j" and emits that string whenever it encounters *constExpr* in the source file. C/C++ does not require that *i* and *j* be defined prior to the *#define* statement. As long as these identifiers have a valid definition prior to the first appearance of *constExpr* in the source file, the C/C++ code will compile correctly. HLA, on the other hand, evaluates the constant expression at the point of the declaration. So *i* and *j* must be defined at the point of the constant declaration (another difference is that HLA requires *i* and *j* to both be constants whereas C/C++ doesn't require this; though if *i* and *j* are not constant objects then this isn't really a manifest constant declaration as we're defining it here, so we won't worry about that).

Beyond the fact that C/C++ relaxes the requirement that i and j be defined before the manifest constant declaration, there is another subtle difference between C/C++ constant declarations and HLA constant declarations: *late binding*. HLA evaluates the value of the expression at the point you declare the constant in your source file (which is why *i* and *j* have to be defined at that point in the previous example). C/C++, on the other hand, only evaluates the constant expression when it actually expands the symbolic identifier in your source file. Consider the following C/C++ source fragment:

```
#define constExpr i*2 + j
#define i 2
#define j 3

   printf( "1:%d\n", constExpr );

#define i 4  //The compiler may issue a warning about this

   printf( "2:%d\n", constExpr );
```

The first *printf* statement in this example will display the value seven (2*2+3) whereas the second example will display 11 (4*2+3). Were you to do the equivalent in HLA (using *val* constants and the "?" statement in HLA, see the HLA reference manual for more details), you would get a different result, e.g.,

```
program t;
#include( "stdlib.hhf" )

? i := 2; // Defines i as a redefinable constant
? j := 3;
const
```

```
   constExpr := i*2 + j;

begin t;

   stdout.put( "1:", constExpr, nl );
   ? i := 4;
   stdout.put( "2:", constExpr, nl );

end t;
```

The HLA code prints the strings "1:7" and "2:7" since HLA only computes the expression "i*2+j" once, when you define the `constExpr` manifest constant.

HLA does allow the definition of *textual substitution constants* using the `text` data type in the const section. For example, consider the following HLA `const` declaration:

```
const
   constExpr :text := "i*2+j";
```

This declaration is totally equivalent to the C/C++ `#define` declaration. However, as you cannot drop in arithmetic expressions into assembly code at arbitrary points in your source file, this textual substitution isn't always legal in an assembly file as it might be in a C/C++ source file. So best not to attempt to use textual substitution constants like this. For completeness' sake, however, the following HLA example demonstrates how to embed textual substitution constant expressions in an HLA source file (and have the compiler calculate the expression at the point of expansion):

```
program t;
#include( "stdlib.hhf" )

? i := 2; // Defines i as a redefinable constant
? j := 3;

const

   // Note: "@eval" tells HLA to evaluate the constant expression inside
   //       the parentheses at the point of expansion.  This avoids some
   //       syntax problems with the stdout.put statements below.

   constExpr :text := "@eval(i*2 + j)";

begin t;

   stdout.put( "1:", constExpr, nl );
   ? i := 4;
   stdout.put( "2:", constExpr, nl );

end t;
```

This HLA example prints "1:7" and "2:11" just like the C/C++ example. Again, however, if the C/C++ manifest constant expansion depends upon late binding (that is, computing the value of the expression at the point of use rather than the point of declaration in the source file) then you should probably expand the text manually at each point of use to avoid problems.

The other way to define constant objects in C/C++ is to use the *const* keyword. By prefacing what looks like an initialized C/C++ variable declaration with the const keyword, you can create constant (immutable at run-time) values in your program, e.g.,

```
const int cConst = 4;
```

Although C/C++ lets you declare constant objects using the *const* keyword, such constants possess different semantics than manifest and literal constants. For example, in C/C++ you may declare an array as follows:

```
#define maxArray 16

int array[ maxArray ];
int anotherArray[ maxArray ];
```

However, the following is generally not legal in C/C++:

```
const int maxBounds = 8;
int iArray[ maxBounds ];
```

The difference between manifest constants and *const* objects in C/C++ has to do with how the program treats the constant object at run-time. Semantically, C++ treats *const* objects as read-only variables. If the CPU and operating system support write-protected memory, the compiler may very well place the const object's value in write-protected memory to enforce the read-only semantics at run-time. Other than the compiler doesn't allow you to store the result of some expression into a *const* object, there is little difference between *const* and *static* variable declarations in C++. This is why a declaration like the one for *iArray* earlier is illegal. C/C++ does not allow you to specify an array bounds using a variable and (with the exception of the read-only attribute) *const* objects are semantically equivalent to variables. To understand why C/C++ *const* objects are not manifest constants and why such declarations are even necessary in C/C++ (given the presence of the *#define* preprocessor directive), we need to look at how CPUs encode constants at the machine code level.

The 80x86 provides special instructions that can encode certain constants directly in a machine instruction. Consider the following two 80x86 assembly language instructions:

```
mov( maxBound, eax );  // Copy maxBound's value into eax
mov( 8, eax );         // Copy the value eight into eax
```

In both cases the machine code the CPU executes consists of three components: an opcode that tells the processor that it needs to move data from one location to another; an addressing mode specification that specifies the register, whether the register is a destination or source register, and the format the other operand takes (it could be a register, a memory location, or a constant); and the third component is the encoding of the memory address or the actual constant. The instruction that copies *maxBound's* value into EAX encodes the address of the variable as part of the instruction whereas the instruction that copies the value eight into EAX encodes the 32-bit value for eight into the instruction. At the machine level, there is a fundamental difference between the execution of these two instructions – the CPU requires an extra step to fetch *maxBound's* value from memory (and this fact remains true even if you initialize *maxBound* to eight and place *maxBound* in write-protected memory). Therefore, the CPU treats certain types of literal and manifest constants differently than it does other types of constants.

Note that a constant (literal, manifest, or otherwise) object in a high level language does not imply that the language encodes that constant as part of a machine instruction. Most CPUs only allow you to encode integer constants (and in some cases, *small* integer constants) directly in an instruction's opcode. The 80x86, for example, does not allow you to encode a floating point constant within a machine instruction. Even if the CPU were capable of encoding floating point and all supported integer values as immediate constants within an opcode,

high level languages like C/C++ support the declaration of large data objects as constant data. For example, you could create a constant array in C/C++ as follows:

```
const int constArray[4] = {0,1,2,3};
```

Few processors, if any, support the ability to encode an arbitrary array constant as immediate data within a machine instruction. Similarly, you'll rarely find structure/record constants or string constants encoded directly within an instruction. Support for large structured constants is the main reason C/C++ adds a another class of constants to the language.

A high level language compiler may encode a literal constant or a manifest constant as an instruction's immediate operand. There is no guarantee, however, that the compiler might actually do this; the CPU must support immediate constants of the specified type and the compiler write must choose to emit the appropriate immediate addressing mode along with the instruction. On the other hand, constants that are not manifest constants (e.g., `const` objects in C/C++) are almost always encoded as memory references rather than as immediate data to an instruction.

So why would you care whether the compiler emits a machine instruction that encodes a constant as part of the opcode versus accessing that constant value appearing in memory somewhere? After all, since machine instructions appear in memory, an immediate constant encoded as part of an instruction also appears in memory. So what's the difference? Well, the principle difference is that accessing a constant value appearing elsewhere in memory (i.e., not as immediate data attached to the instruction) requires twice as much memory to encode. First, you need the constant itself, consuming memory somewhere; second, you need the address that constant encoded as part of the instruction. Since the address of a constant value typically consumes 32-bits, it typically takes twice as much memory to encode the access to the constant. Of course, if you reference the same constant value throughout your code, the compiler should only store one copy of the constant in memory and every instruction that references that constant would reference the same memory location. However, even if you amortize the size of the constant access over any number of instructions, the bottom line is that encoding constants as memory location still takes more room than encoding immediate constants.

Another difference between manifest/literal constants and read-only objects is that decent compilers will compute the result of constant expressions at compile-time, something that it may not be able to do with read-only objects. Consider the following C++ code:

```
#define one 1
#define two 2
const int three = 3;
const int four = 4;
int i;
int j;

   i = one + two;
   j = three + four;
```

Most decent C/C++ compilers will replace the first assignment statement above with the following:

```
i = 3;  // Compiler compute 1+2 at compile-time
```

On the 80x86 processor this statement takes a single machine instruction to encode (this is generally true for most processors). Some compilers, however, may not precompute the value of the expression "three+four" and will, instead, emit machine instructions to fetch these values from their memory locations and add them at run-time.

HLA provides a mechanism whereby you can create immutable "variables" in your code if you need the storage semantics of a variable that the program must not change at run-time. You can use HLA's *readonly* declaration section to declare such objects, e.g.,

```
readonly
   constValue :int32 := -2;
```

For all intents and purposes, HLA treats the *readonly* declaration like a *static* declaration. The two major differences are than HLA requires an initializer associated with all *readonly* objects and HLA attempts to place such objects in read-only memory at run time. Note that HLA doesn't prevent you from attempting to store a value into a *readonly* object. That is, the following is perfect legal and HLA will compile the program without complaint:

```
mov( 56, constValue );
```

Of course, if you attempt to execute the program containing this statement, the program will probably abort with an *illegal access* violation when the program attempts to execute this statement. This is because HLA will place this object in write-protected memory and the operating system will probably raise an exception when you attempt to execute this statement.

A compiler may not be able to efficiently process a constant simply because it is a literal constant or a manifest constant. For example most CPUs are not capable of encoding a string constant in an instruction. Using a manifest string constant may actually make your program less efficient. Consider the following C code:

```
#define strConst "A string constant"
      .
      .
      .
   printf( "string: %s\n", strConst );
      .
      .
      .
   sptr = strConst;
      .
      .
      .
   result = strcmp( s, strConst );
      .
      .
      .
```

Because the compiler (actually, the C preprocessor) expands the macro *strConst* to the string literal "A string constant" everywhere the identifier *strConst* appears in the source file, the above code is actually equivalent to:

```
      .
      .
      .
   printf( "string: %s\n", "A string constant" );
      .
      .
      .
   sptr = "A string constant";
      .
```

```
    .
    .
result = strcmp( s, "A string constant" );
    .
    .
    .
```

The problem with this code is that the same string constant appears at different places throughout the program. In C/C++, the compiler places the string constant off in memory somewhere and substitutes a pointer to that string for the string literal constant. A naive compiler would wind up making three separate copies of the string in memory, thus wasting space since the data is exactly the same in all three cases. Compiler writers figured this out a couple of decades ago and modified their compilers to keep track of all the strings the compiler had already emitted; when the program used the same literal string constant again, the compiler would not allocate storage for a second copy of the string, it would simply return the address of the earlier string appearing in memory. Such an optimization could reduce the size of the code the compiler produced by a fair amount if the same string appears through some program. Unfortunately, this optimization probably lasted about a week before the compiler vendors figured out that there were problems with this approach. One major problem with this approach is that a lot of C programs would assign a string literal constant to a character pointer variable and then proceed to change the characters in that literal string, e.g.,

```
sptr = "A String Constant";
    .
    .
    .
*(sptr+2) = 's';
    .
    .
    .
printf( "string: '%s'\n", sptr ); /* displays "string: 'A string Constant'" */
    .
    .
    .
printf( "A String Constant" );    /* Prints "A string Constant"! */
```

Compilers that used the same string constant in memory for multiple occurrences of the same string literal appearing in the program quickly discovered that this trick wouldn't work if the user stored data into the string object, as the code above demonstrates. Although this is a bad programming practice, it did occur frequently enough that the compiler vendors could not use the same storage for multiple copies of the same string literal. Even if the compiler vendor were to place the string literal constant into write-protected memory to prevent this problem, there are other semantic issues that this optimization raise. Consider the following C/C++ code:

```
sptr1 = "A String Constant";
sptr2 = "A String Constant";
s1EQs2 = sptr1 == sptr2;
```

Will *s1EQs2* contain true (1) or false (0) after executing this instruction sequence? In programs written before C compilers had strong optimizers available, this sequence of statements would leave false in *s1EQs2* because the compiler created two different copies of the same string data and placed those strings at different addresses in memory (so the addresses the program assigns to *sptr1* and *sptr2* would be different). In a later compiler, that kept only a single copy of the string data in memory, this code sequence would leave true sitting in *s1EQs2* since

both *sptr1* and *sptr2* would be pointing at the same address in memory; this difference exists regardless of whether the string data appears in write-protected memory.

Of course, when converting the C/C++ code to assembly language, it is your responsibility to determine whether you can merge strings and use a common copy of the data or whether you will have to use a separate copy of the string data for each instance of the symbolic constant throughout your assembly code.

C/C++ supports other composite constant types as well (e.g., arrays and structures/records). This discussion of string constants in a program applies equally well to these other data types. Large data structures that the CPU cannot represent as a primitive data type (i.e., hold in a general purpose register) almost always wind up stored in memory and the program access the constant data exactly as it would access a variable of that type. On many modern systems, the compiler may place the constant data in write-protected memory to prevent the program from accidentally overwriting the constant data, but otherwise the "constant" is structurally equivalent to a variable in every sense except the ability to change its value. You can place such "constant" declarations in an HLA *readonly* declaration section to achieve the HLA equivalent to the C/C++ code.

HLA also allows the declaration of composite constants in the *const* section. For example, the following is perfect legal in HLA:

```
const
   constArray :int32[4] := [1,2,3,4];
```

However, you should note that HLA maintains this array strictly at compile-time within the compiler. You cannot, for example, write HLA code like the following once you have the above declaration:

```
   for( mov( 0, ebx ); ebx < 4; inc( ebx )) do

      mov( constArray[ ebx*4 ], eax );
      stdout.puti32( eax );

   endfor;
```

The problem with this code is that constArray is not a memory location so you cannot refer to it using an 80x86 addressing mode. In order for this array constant to be accessible (as an array) at run time, you have to make a copy of it in memory. You can do this with an HLA declaration like the following:

```
readonly
   rtConstArray :int32[4] := constArray; //Assuming the declaration given earlier.
```

Please consult the HLA documentation for more details on structured (composite) constants. Although these are quite useful for HLA programmers, they aren't generally necessary when converting C/C++ code to HLA. As such, they're a bit beyond the scope of this chapter so we won't deal any farther with this issue here.

3.2.5: Arithmetic Expressions in C and Assembly Language

One of the major advances that high level languages provided over low level languages was the use of algebraic-like expressions. High level language arithmetic expressions are an order of magnitude more readable than the sequence of machine instructions the compiler converts them into. However, this conversion process (from arithmetic expressions into machine code) is also one of the more difficult transformation to do efficiently and a fair percentage of a typical compiler's optimization phase is dedicated to handling this transformation.

Computer architects have made extensive studies of typical source files and one thing they've discovered is that a large percentage of assignment statements in such programs take one of the following forms:

```
var = var2;
var = constant;
var = op var2;
var = var op var2;
var = var2 op var3;
```

Although other assignments do exist, the set of statements in a program that takes one of these form is generally larger than any other group of assignment statements. Therefore, computer architects have generally optimized their CPUs to efficiently handle one of these forms.

The 80x86 architecture is what is known as a *two-address machine*. In a two-address machine, one of the source operands is also the destination operand. Consider the following 80x86/HLA `add` instruction:

```
add( ebx, eax );  ; computes eax := eax + ebx;
```

Two-address machines, like the 80x86, can handle the first four forms of the assignment statement given earlier with a single instruction. The last form, however, requires two or more instructions and a temporary register. For example, to compute "var1 = var2 + var3;" you would need to use the following code (assuming `var2` and `var3` are memory variables and the compiler is keeping `var1` in the EAX register):

```
mov( var2, eax );
add( var3, eax );  //Result (var1) is in EAX.
```

Once your expressions get more complex than the five forms given earlier, the compiler will have to generate a sequence of two or more instructions to evaluate the expression. When compiling the code, most compilers will internally translate complex expressions into a sequence of "three address statements" that are semantically equivalent to the more complex expression. The following is an example of a more complex expression and a sequence of three-address instructions that are representative of what a typical compiler might produce:

```
// complex = ( a + b ) * ( c - d ) - e/f;

temp1 = a + b;
temp2 = c - d;
temp1 = temp1 * temp2;
temp2 = e / f;
complex = temp1 - temp2;
```

If you study the five statements above, you should be able to convince yourself that they are semantically equivalent to the complex expression appearing in the comment. The major difference in the computation is the introduction of two temporary values (`temp1` and `temp2`). Most compilers will attempt to use machine registers to maintain these temporary values (assuming there are free registers available for the compiler to use).

Table 3-5 lists most of the arithmetic operators used by C/C++ programs as well as their associativity.

Table 3-5: C/C++ Operators, Precedence, and Associativity

Precedence	Operator Classification	Associativity	C/C++ Operators
1 (highest)	Primary Scope Resolution (C++)	left to right	::
2	Primary	left to right	( ) [ ] . ->
3	Unary (monadic[a])	right to left	++ -- + - ! ~ & * (*type*) sizeof new delete
4	Multiplicative (dyadic[b])	left to right	* / %
5	Additive (dyadic)	left to right	+ -
6	Bitwise Shift (dyadic)	left to right	<< >>
7	Relational (dyadic)	left to right	< > <= >=
8	Equality (dyadic)	left to right	== !=
9	Bitwise AND (dyadic)	left to right	&
10	Bitwise Exclusive OR (dyadic)	left to right	^
11	Bitwise Inclusive OR (dyadic)	left to right	|
12	Logical AND (dyadic)	left to right	&&
13	Logical OR (dyadic)	left to right	||
14	Conditional (triadic[c])	right to left	? :
15	Assignment	right to left	= += -= *= /= <<= >>= %= &= ^= |=
16 (lowest)	Comma	left to right	,

a.Monadic means single operand .

b.Dyadic means two operand .
c.Triadic means three operands .

3.2.5.1: Converting Simple Expressions Into Assembly Language

In this section we will consider the case where we need to convert a simple C/C++ expressions into assembly language. We've already discussed the conversion of the primary operators (in Table 3-5) into assembly language, so we won't bother repeating that discussion here. Likewise, we've already discussed the address-of operator ("&") and the dereferencing operator ("*") so we'll skip their discussions here as well.

Although the conversion of the remaining operators into assembly language is generally obvious, there are a few peculiarities. So it's worthwhile to quickly discuss how to convert a simple expression of the form @X, X@, or X@Y (where '@' represents one of the operators found in Table 3-5) into assembly language. Note that the discussion that follows deals with integer (signed or unsigned) only. The conversion of floating point expressions into assembly language is actually easier than converting integer expressions. This book will not deal with converting floating point expressions into assembly language. There are two reasons for this: (1) once you see how to convert integer expressions to assembly, you'll discover than floating point expression conversion is very similar; (2) the Win32 API uses very few floating point values. The whole reason for this chapter is to describe the C/C++ to assembly conversion process so you can read and understand existing C/C++ documentation when writing Windows assembly code. Since you won't find much Windows programming documentation that involves the use of floating point arithmetic, there is little need to present that information here. If you're interested in the subject, be sure to check out the discussion of this process in *The Art of Assembly Language Programming*.

Each of the examples appearing in this section will assume that you're operating on 32-bit integers producing a 32-bit result (except in the case of boolean results, where this book will assume an 8-bit result is probably sufficient). If you need to operate on eight or sixteen bit values, so sweat, just substitute the 8-bit or 16-bit registers in place of the 32-bit registers you'll find in these examples. If you need to deal with larger values (e.g., long long ints), well, that's beyond the scope of this book; please see the section on extended precision arithmetic in *The Art of Assembly Language* for details on those operations.

Translating the ++ and -- (increment and decrement) operators from C/C++ to assembly language looks, at first, like a trivial operation. You simply substitute an `inc` or `dec` instruction for these operators. However, there are two details that complicate this conversion by a slight amount: pre- and post- increment/decrement operations and pointer increment/decrement operations.

Normally, when you apply the ++ (increment) operator to an integer variable, the ++ operator increments the value of that variable by one. Similarly, when you apply the -- (decrement) operator to an integer variable, the -- operator decrements that variable by one. However, C/C++ also allows you to apply the ++ and -- operators to pointer variables as well as integers (pointers and integer variables are the only legal objects to which you may apply these operators, though). The semantics of a pointer increment are different than the semantics of an integer increment; applying the ++ operator to a pointer increments that pointer variable *by the size of the object at which the pointer refers*. For example, if `pi` is a pointer that points at a 32-bit integer value somewhere in memory, then `++pi` adds four to `pi` (rather than one); this cause `pi` to point at the next sequential memory location that can hold a 32-bit integer (without overlapping the current integer in memory). Similarly, the -- (decrement) operator subtracts the size of the object at which a pointer refers from the pointer's value. So `--pi` would subtract

four from *pi* if *pi* points at a 32-bit integer. So the basic conversion of the ++ and -- operator to assembly language is as Table 3-6 describes.

Table 3-6: Converting C/C++ Increment/Decrement Operators to Assembly

C/C++	HLA[a]
int i; ++i; i++;	inc(i);
int *pi; ++pi; pi++;	add(@size(int32), i);
int i; --i; i--;	dec(i);
int *pi; --pi; pi--;	sub(@size(int32), i);

a.In the pointer examples, substitute the appropriate type identifier when incrementing a pointer to some type other than int32.

The increment and decrement operators may appear before or after a variable. If a C/C++ statement consists of a single variable with one of these operators, then whether you use the *pre-increment/decrement form* (sticking the ++ or -- before the variable) or the *post-increment/decrement* form (placing the ++ or -- operator after the variable) is irrelevant. In either case the end result is that the program will increment or decrement the variable accordingly:

```
   c++;  // is equivalent to
   ++c;

// and

   --c;  // is equivalent to
   c--;
```

If the C/C++ increment and decrement operators are attached to a variable within a larger expression, then the issue of pre-increment/decrement versus post-increment/decrement makes a big difference in the final result. Consider the statements “a = ++c;” and “a = c++;”. In both cases the program will add one to variable *c* (assuming c is an integer rather than a pointer to some object). However, these two statements are quite different with respect to the value they assign to variable *a*. The first example here first increments the value in *c* and then assigns the value in *c* to *a* (hence the term *pre-increment* since the program first increments *c* and then uses its value in the expression). The second statement here first grabs the value in *c*, assigns that to *a*, and then increments *c* (hence the term *post-increment* since this expression increments *c* after using its value). Here’s some sample HLA code that implements these two statements:

```
// a = ++c;

   inc( c );         // pre-increment the value in c.
   mov( c, eax );
   mov( eax, a  );

// a = c++;

   mov( c, eax );
   mov( eax, a );
   inc( c );         // post-increment the value in c.
```

The C/C++ compiler effectively ignores the unary "+" operator. If you attach this operator to an operand, it does not affect that value of that operand in any way. It's presence in the language is mainly for notational purposes. It lets you specify positive numeric constants like +123.456 in the source file. Sometimes explicitly place the "+" in front of such a constant can make the program more readable. However, since this operator rarely appears in real-world C/C++ programs, you're unlikely to see it.

The unary "-" operator negates the expression/variable that immediately follows it. The important thing to note is that this operator negates the value of the operand immediately following the "-" for use in the expression containing the operator. In particular, if a simple variable immediately follows the unary "-" this operator doesn't negate that operator directly. Therefore, you cannot use the 80x86 neg instruction on the variable except for the very special case where you have a statement like the following:

```
i = -i;
```

Instead, you must move the value of the variable into a register, negate the value of that register, and then use that register's value within the expression. For example, consider the following:

```
//  j = -i;

   mov( i, eax );
   neg( eax );
   mov( eax, j );
```

The unary "!" operator is the logical not operator. If the sub-expression (i.e., variable) appearing immediately to the left of this operator is zero, the "!" operator returns one. If the value of that sub-expression is non-zero, this operator returns zero. To convert this to assembly language, what you would do is test the operand for zero and set the result to one if it is zero, zero if the operand is not zero. You can use the *cmp* (or *test*) instruction along with the 80x86 *setne* instruction to achieve this:

```
   // Convert !i to assembly, assume i is an int variable, leave result in AL(or EAX)

   cmp( i, 0 );
   setne( al );
   // movsx( al, eax );  // Do this if you need a 32-bit boolean result.
```

A very common construct you'll see in many C/C++ programs is a sub-expression like "!!i" (that is, apply the logical not operator twice to the same value. This *double logical negation* converts the value in *i* to zero if it was previously zero, to one if it was previously non-zero. Rather than execute the previous code fragment twice, you can easily achieve this effect as follows:

```
   // Convert !!i to assembly, assume i is an int variable, leave result in AL(or EAX)
```

```
cmp( i, 0 );
sete( al );
// movsx( al, eax );  // Do this if you need a 32-bit boolean result.
```

The C/C++ unary "~" operator does a bitwise logical not on its operand (that is, it inverts all the bits of the operand). This is easily achieved using the 80x86 *not* instruction as follows:

```
// j = ~i

    mov( i, eax );
    not( eax );
    mov( i, j );
```

For the special case of "i = ~i;" you can use the 80x86 *not* instruction to negate i directly, i.e., "not(i);".

A simple C/C++ expression like "x = y * z;" is easily converted to assembly language using a code sequence like the following:

```
// x = y * z;

mov( y, eax );
intmul( z, eax );
mov( eax, x );

// Note: if y is a constant, can do the following:
// (because multiplication is commutative, this also works if z is a constant,
//  just swap z and y in this code if that is the case):

intmul( y, z, eax );
mov( eax, x );
```

Technically, the *intmul* instruction expects signed integer operands so you would normally use it only with signed integer variables. However, if you're not checking for overflow (and C/C++ doesn't so you probably won't need to either), then a two's complement signed integer multiply produces exactly the same result as an unsigned multiply. See *The Art of Assembly Language* if you need to do a true unsigned multiply or an extended precision multiply. Also note that the intmul instruction only allows 16-bit and 32-bit operands. If you need to multiple two 8-bit operands, you can either zero extend them to 16 (or 32) bits or you can use the 80x86 *imul* or *mul* instructions (see *The Art of Assembly Language* for more details).

The C/C++ division and modulo operators ("/" and "%", respectively) almost translate into the same code sequence. This is because the 80x86 *div* and *idiv* instructions calculate both the quotient and remainder of a division at the same time.

Unlike integer multiplication, division of signed versus unsigned operands does not produce the same result. Therefore, when dividing values that could potentially be negative, you must use the *idiv* instruction. Only use the *div* instruction when dividing unsigned operands.

Another complication with the division operation is that the 80x86 does a 64/32-bit division (that is, it divides a 64-bit number by a 32-bit number). Since both C/C++ operands are 32-bits you will need to sign extend (for signed integer operands) or zero extend (for unsigned integer operands) the numerator to 64 bits. Also remember that the *div* and *idiv* instructions expect the numerator in the EDX:EAX register pair (or DX:AX for 32/16 divisions, or AH:AL for 16/8 divisions, see *The Art of Assembly Language* for more details). The last thing to note is

that these instructions return the quotient in EAX (AX/AL) and they return the remainder in EDX (DX/AH). Here's the code to translate an expression of the form "x=y/z;" into 80x86 assembly code:

```
// x = y / z;   -- assume all operands are unsigned.

mov( y, eax );
xor( edx, edx );  // zero extend EAX to 64 bits in EDX:EAX
div( z );
mov( eax, x );    // Quotient winds up in EAX


// x = y % z;   -- assume all operands are unsigned.

mov( y, eax );
xor( edx, edx );  // zero extend EAX to 64 bits in EDX:EAX
div( z );
mov( edx, x );    // Remainder winds up in EDX


// x = y / z;   -- assume all operands are signed.

mov( y, eax );
cdq();            // sign extend EAX to 64 bits in EDX:EAX
idiv( z );
mov( eax, x );    // Quotient winds up in EAX


// x = y % z;   -- assume all operands are signed.

mov( y, eax );
cdq();            // sign extend EAX to 64 bits in EDX:EAX
idiv( z );
mov( edx, x );    // Remainder winds up in EDX
```

Converting C/C++ expressions involving the "+", "-", "&", "|", and "&" operators into assembly language is quite easy. A simple C/C++ expression of the form "a = b @ c;" (where '@' represents one of these operators) translates into the following assembly code:

```
//  a = b @ c;

mov( b, eax );
instr( c, eax );  //instr = add, sub, and, or, xor, as appropriate
mov( eax, a );
```

The C/C++ programming language provides a shift left operator ("<<"). This dyadic operator returns the result of its left operand shifted to the left the number of bits specified by its right operand. An expression of the form "a=b<<c;" is easily converted to one of two different HLA instruction sequences (chosen by whether *c* is a constant or a variable expression) as follows:

```
// a = b << c;   -- c is a constant value.

mov( b, eax );
shl( c, eax );
mov( eax, a );
```

```
// a = b << c;   -- c is a variable value in the range 0..31.

mov( b, eax );
mov( (type byte c), cl );  //assume H.O. bytes of c are all zero.
shl( cl, eax );
mov( eax, a );
```

C/C++ also provides a shift right operator, “>>”. This translates to a sequence that is very similar to the conversion of the “<<“ operator with one caveat: the 80x86 supports two different shift right instructions: *shr* (shift logical right) and *sar* (shift arithmetic right). The C/C++ language doesn’t specify which shift you should use. Some compilers always use a logical shift right operation, some use a logical shift right for unsigned operands and they use an arithmetic shift right for signed operands. If you don’t know what you’re supposed to use when converting code, using a logical (unsigned) shift right is probably the best choice because this is what most programmers will expect. That being the case, the shift right operator (“>>”) appearing in an expression like “a=b>>c;” translates into 80x86 code thusly:

```
// a = b >> c;   -- c is a constant value.

mov( b, eax );
shr( c, eax );
mov( eax, a );

// a = b >> c;   -- c is a variable value in the range 0..31.

mov( b, eax );
mov( (type byte c), cl );  //assume H.O. bytes of c are all zero.
shr( cl, eax );
mov( eax, a );
```

If you decide you need to use an arithmetic (signed) shift right operation, simply substitute *sar* for *shr* in this code.

The logical OR and logical AND operators (“||” and “&&”) return the values zero or one based on the values of their two operands. The logical OR operator (“||”) returns one if either or both operands are non-zero; it returns zero if both operands are zero. The logical AND operator (“&&”) returns zero if either operand is zero, it returns one only if both operands are non-zero. There is, however, one additional issue to consider: these operators employ *short-circuit boolean evaluation*. When computing “X && Y” the logical AND operator will not evaluate Y if it turns out that X is false (there is no need because if X is false, the full expression is always false). Likewise, when computing “X || Y” the logical OR operator will not evaluate Y if it turns out that X is true (again, there will be no need to evaluate Y for if X is true the result is true regardless of Y’s value). Probably for the majority of expressions it doesn’t really matter whether the program evaluates the expression using short-circuit evaluation or *complete boolean evaluation*; the result is always the same. However, because C/C++ promises short-circuit boolean evaluation semantics, many programs are written to depend on these semantics and will fail if you recode the expression using complete boolean evaluation. Consider the following two examples that demonstrate two such situations:

```
if( pi != NULL && *pi == 5 )
{
   // do something if *pi == 5...
}
   .
   .
   .
```

```
if( --x == 0 || ++y < 10 )
{
   // do something if x == 0 or y < 10
}
```

The first *if* statement in this example only works properly in all cases when using short-circuit evaluation. The left-hand operand of the "&&" operator evaluates false if *pi* is NULL. In that case, the code will not evaluate the right operand and this is a good thing for if it did it would dereference a NULL pointer (which will raise an exception under Windows). In the second example above, the result is not as drastic were the system to use short-circuit evaluation rather than complete boolean evaluation, but the program would produce a different result in *y* when using complete boolean evaluation versus short-circuit boolean evaluation. The reason for this difference is that the right-hand side of the expression increments *y*, something that doesn't happen if the left operand evaluates true.

Handling short-circuit boolean evaluation almost always means using conditional jumps to skip around an expression. For example, given the expression "Z = X && Y" the way you would encode this using pure short-circuit evaluation is as follows:

```
   xor( eax, eax );   // Assume the result is false.
   cmp( eax, X );     // See if X is false.
   je isFalse;
   cmp( eax, Y );     // See if Y is false.
   je isFalse;

      inc( eax );     // Set EAX to 1 (true);

isFalse:
   mov( eax, Z );     // Save 0/1 in Z
```

Encoding the logical OR operator using short-circuit boolean evaluation isn't much more difficult. Here's an example of how you could do it:

```
   xor( eax, eax );   // Assume the result is false.
   cmp( eax, X );     // See if X is true.
   jne isTrue;
   cmp( eax, Y );     // See if Y is false.
   je isFalse;
isTrue:
      inc( eax );     // Set EAX to 1 (true);

isFalse:
   mov( eax, Z );     // Save 0/1 in Z
```

Although short-circuit evaluation semantics are crucial for the proper operation of certain algorithms, most of the time the logical AND and OR operands are simple variables or simple sub-expressions whose results are independent of one another and quickly computed. In such cases the cost of the conditional jumps may be more expensive than some simple straight-line code that computes the same result (this, of course, depends entirely on which processor you're using in the 80x86 family). The following code sequence demonstrates one (somewhat tricky) way to convert "Z = X && Y" to assembly code, assuming *X* and *Y* are both 32-bit integer variables:

```
   xor( eax, eax ); // Initialize EAX with zero
   cmp( X, 1 );     // Note: sets carry flag if X == 0, clears carry in all other cases.
   adc( 0, eax );   // EAX = 0 if X != 1, EAX = 1 if X = 0.
```

```
cmp( Y, 1 );     // Sets carry flag if Y == 0, clears it otherwise.
adc( 0, eax );   // Adds in one if Y = 0, adds in zero if Y != 0.
setz( al );      // EAX = X && Y
mov( eax, Z );
```

Note that you cannot use the 80x86 and instruction to merge *X* and *Y* together to test if they are both non-zero. For if *X* contained $55 (0x55) and *Y* contained $aa (0xaa) their bitwise AND (which the *and* instruction produces) is zero, even though both values are logically true and the result should be true. You may, however, use the 80x86 *or* instruction to compute the logical OR of two operands. The following code sequence demonstrates how to compute "Z = X || Y;" using the 80x86 *or* instruction:

```
xor( eax, eax );  // Clear EAX's H.O. bytes
mov( X, ebx );
or( Y, ebx );
setnz( al );      // Sets EAX to one if X || Y (in EBX) is non-zero.
```

The conditional expression in C/C++ is unusual insofar as it is the only ternary (three-operand) operator that C/C++ provides. An assignment involving the conditional expression might be

```
a = (x != y) ? trueVal : falseVal;
```

The program evaluates the expression immediately to the left of the "?" operator. If this expression evaluates true (non-zero) then the compiler returns the value of the sub-expression immediately to the right of the "?" operator as the conditional expression's result. If the boolean expression to the left of the "?" operator evaluates false (i.e., zero) then the conditional expression returns the result of the sub-expression to the right of the ":" in the expression. Note that the conditional operator does not evaluate the true expression (*trueVal* in this example) if the condition evaluates false. Likewise, the conditional operator does not evaluate the false expression (*falseVal* in this example) if the expression evaluates true. This is similar to short-circuit boolean evaluation in the "&&" and "||" operators. You encode the conditional expression in assembly as though it were an *if/else* statement, e.g.,

```
   mov( falseVal, edx );   // Assume expression evaluates false
   mov( x, eax );
   cmp( eax, y );
   jne TheyreNotEqual
   mov( trueVal, edx );    // Assumption was incorrect, set EDX to trueVal
TheyreNotEqual:
   mov( edx, a );          // Save trueVal or falseVal (as appropriate) in a.
```

C/C++ provides a set of assignment operators. The are the following:

```
=   +=   -=   &=   ^=   |=   *=   /=   %=   <<=   >>=
```

Generally, C/C++ programmers use these assignment operators as stand-alone C/C++ statements, but they may appear as subexpressions as well. If the C/C++ program uses these expressions as stand-alone statements (e.g., "x += y;") then the statement "x @= y;" is completely equivalent to "x = x @ y;" where '@' represents the operator above. Therefore, the conversion to assembly code is fairly trivial, you simply use the conversions we've

been studying throughout this section. Table 3-7 lists the equivalent operations for each of the assignment operators.

Table 3-7: Converting Assignment Operators To Assembly Language

C/C++ Operator	C/C++ Example	Equivalent To This C/C++ Code	HLA Encoding
=	x = y;	x = y;	mov(y, eax); mov(eax, x);
+=	a += b;	a = a + b;	mov(b, eax); add(eax, a);
-=	a -= b;	a = a - b;	mov(b, eax); sub(eax, a);
&=	a &= b;	a = a & b;	mov(b, eax); and(eax, a);
|=	a |= b;	a = a | b;	mov(b, eax); or(eax, a);
^=	a ^= b;	a = a ^ b;	mov(b, eax); xor(eax, a);
<<=	a <<= b;	a = a << b;	mov((type byte b), cl); shl(cl, a);
>>=	a >>= b;	a = a >> b;	mov((type byte b), cl); shr(cl, a);

*=	a *= b;	a = a * b;	mov(a, eax);
			intmul(b, eax);
			mov(eax, a);

/=	a /= b;	a = a / b;	mov(a, eax);
			cdq; // or xor(edx, edx);
			div(b);
			mov(eax, a); // Store away quotient

%=	a %= b;	a = a % b;	mov(a, eax);
			cdq; // or xor(edx, edx);
			div(b);
			mov(edx, a); // Store away remainder

The comma operator in C/C++ evaluates two subexpressions and then throws the result of the first expression away (i.e., it computes the value of the first/left expression strictly for any side effects it produces). In general, just convert both sub-expressions to assembly using the rules in this section and then use the result of the second sub-expression in the greater expression, e.g.,

```
//  x = ( y=z, a+b );

   mov( z, eax );
   mov( eax, y );
   mov( a, eax );
   add( b, eax );
   mov( eax, x );
```

3.2.5.2: Operator Precedence

The precedence of an operator resolves the ambiguity that is present in an expression involving several different operands. For example, given the arithmetic expression "4+2*3" there are two possible values we could logically claim this expression produces: 18 or 10 (18 is achieve by adding four and two then multiplying their sum by three; 10 is achieved by multiplying two times three and adding their product together with four). Now you may be thoroughly convinced that 10 is the *correct* answer, but that's only because by convention most people agree that multiplication has a higher precedence than addition, so you must do the multiplication first in this expression (that is, you've followed an existing convention in order to resolve the ambiguity). C/C++ also has its own precedence rules for eliminating ambiguity. In Table 3-5 the *precedence level* appears in the left-most column. Operators with a lower precedence level have a higher precedence and, therefore, take precedence over other operators at a lower precedence. You'll notice that the multiplication operator in C/C++ ("*") has a higher precedence than addition ("+") so C/C++ will produce 10 for the expression "4+2*3" just as you've been taught to expect.

Of course, you can always eliminate the ambiguity by explicitly specifying parentheses in your expressions. Indeed, the whole purpose of precedence is to implicitly specify where the parentheses go. If you have two operators with different precedences (say '#' and '@') and three operands, and you have an expression of the form X#Y@Z then you must place parentheses around the operands connected by the operator with the higher precedence. In this example, if '@' has a higher precedence than '#' you'd wind up with X#(Y@Z). Conversely, if '#' has a higher precedence than '@' you'd wind up with (X#Y)@Z.

An important fact to realize when converting C/C++ code into assembly language is that precedence only controls the implicit placement of parentheses within an expression. That is, precedence controls which operands we associate with a given operator. Precedence does not necessarily control the order of evaluation of the operands. For example, consider the expression "5*4+2+3". Since multiplication has higher precedence than addition, the "5" and "4" operands attach themselves to the "*" operator (rather than "4" attaching itself to the "+" operator). That is, this expression is equivalent to "(5*4)+2+3". The operator precedence, contrary to popular opinion, does not control the order of the evaluation of this expression. We could, for example, compute the sub-expression "2+3" prior to computing "5*4". You still get the correct result when computing this particular addition first.

When converting a complex expression to assembly language, the first step is to explicitly add in the parentheses implied by operator precedence. The presence of these parentheses will help guide the conversion to assembly language (we'll cover the exact process a little later in this chapter).

3.2.5.3: Associativity

Precedence defines where the parentheses go when you have three or more operands separated by different operators at different precedence levels. Precedence does not deal with the situation where you have three or more operands separated by operators at the same precedence level. For example, consider the following expression:

```
5 - 4 - 3;
```

Does this compute "5 - (4 - 3);" or does it compute " (5 - 4) -3;"? Precedence doesn't answer the question for us because the operators are all the same. These two expressions definitely produce different results (the first expression produces 5-1=4 while the second produces 1-3=-2). Associativity is the mechanism by which we determine the placement of parentheses around adjacent operators that have the same precedence level.

Operators generally have one of three different associativities: left, right, and none. C/C++ doesn't have any non-associative operators, so we'll only consider left associative and right associative operators here. Table 3-5 lists the associativity of each of the C/C++ operators (left or right). If two left associative operators are adjacent to one another, then you place the left pair of operands and their operator inside parentheses. If two right associative operators appear adjacent to one another in an expression, then you place a pair of parentheses around the right-most pair of operands and their operator, e.g.,

```
5 - 4 - 3     -becomes-    (5 - 4) - 3
x = y = z     -becomes-    x = (y = z)
```

Like precedence, associativity only controls the implied placement of the parentheses within an expression. It does not necessarily suggest the order of evaluation. In particular, consider the following arithmetic expression:

```
5 + 4 + 3 + 2 + 1
```

Because addition is left associative, the implied parentheses are as follows:

```
(((5 + 4) + 3) + 2) + 1
```

However, a compiler is not forced to first compute 5+4, then 9 + 3, then 12 + 2, etc. Because addition is commutative, a compiler can rearrange this computation in any way it sees fit as long as it produces the same result as this second expression.

3.2.5.4: Side Effects and Sequence Points

A side effect is any modification to the global state of a program other than the immediate result a piece of code is producing. The primary purpose of an arithmetic expression is to produce the expression's result. Any other changes to the system's state in an expression is a side effect. The C/C++ language is especially guilty of allowing side effects in an arithmetic expression. For example, consider the following C/C++ code fragment:

```
i = i + *pi++ + (j = 2) * --k
```

This expression exhibits four separate side effects: the decrement of `k` at the end of the expression, the assignment to `j` prior to using `j's` value, the increment of the pointer `pi` after dereferencing `pi`, and the assignment to `i` (generally, if this expression is converted to a stand-alone statement by placing a semicolon after the expression, we consider the assignment to `i` to be the purpose of the statement, not a side effect).

Another way to create side effects within an expression is via a function call. Consider the following C++ code fragment:

```
int k;
int m;
int n;

int hasSideEffect( int i, int& j )
{
```

```
   k = k + 1;
   hasSideEffect = i + j;
   j = i;
}
      .
      .
      .
   m = hasSideEffect( 5, n );
```

In this example, the call to the *hasSideEffect* function produces two different side effects: (1) the modification of the global variable *k* and the modification of the pass by reference parameter *j* (actual parameter is *n* in this code fragment). The real purpose of the function is to compute the function's return result; any modification of global values or reference parameters constitutes a side effect of that function, hence the invocation of such a function within an expression causes the expression to produce side effects. Note that although C does not provide "pass by reference" parameters as C++ does, you can still pass a pointer as a parameter and modify the dereferenced object, thus achieving the same effect.

The problem with side effects in an expression is that most C/C++ compilers do not guarantee the order of evaluation of the components that make up an expression. Many naive programmers (incorrectly!) assume that when they write an expression like the following:

```
i = f(x) + g(x);
```

the compiler will emit code that first calls function *f* and then calls function *g*. The C and C++ programming languages, however, do not specify this order of execution. That is, some compilers will indeed call *f* , then call *g*, and then add their return results together; some other compilers, however, may call *g* first, then *f*, and then compute the sum of the function return results. That is, the compiler could translate the expression above into either of the following simplified code sequences before actually generating native machine code:

```
// Conversion #1 for "i = f(x) + g(x);"

   temp1 = f(x);
   temp2 = g(x);
   i:= temp1 + temp2;

// Conversion #2 for "i = f(x) + g(x);"

   temp1 = g(x);
   temp2 = f(x);
   i = temp2 + temp1;
```

Note that issues like precedence, associativity, and commutativity have no bearing on whether the compiler evaluates one sub-component of an expression before another. For example, consider the following arithmetic expression and several possible intermediate forms for the expression:

```
   j = f(x) - g(x) * h(x);

// Conversion #1 for this expression:

   temp1 = f(x);
   temp2 = g(x);
   temp3 = h(x);
   temp4 = temp2 * temp3
   j = temp1 - temp4;
```

```
// Conversion #2 for this expression:

   temp2 = g(x);
   temp3 = h(x);
   temp1 = f(x);
   temp4 = temp2 * temp3
   j = temp1 - temp4;

// Conversion #3 for this expression:

   temp3 = h(x);
   temp1 = f(x);
   temp2 = g(x);
   temp4 = temp2 * temp3
   j = temp1 - temp4;
```

Many other combinations are possible.

The specification for the C/C++ programming languages explicitly leave the order of evaluation undefined. This may seem somewhat bizarre, but there is a good reason for this: sometimes a compiler can produce better machine code by rearranging the order it uses to evaluate certain sub-expressions within an expression. Any attempt on the part of the language designer to force a particular order of evaluation on a compiler's implementor may limit the range of optimizations possible. Therefore, very few languages explicitly state the order of evaluation for an arbitrary expression.

There are, of course, certain rules that most languages do enforce. Though the rules vary by language, there are some fairly obvious rules that most languages (and their implementation) always follow because intuition suggests the behavior. Probably the two most common rules that you can count on are the facts that all side effects within an expression occur prior to the completion of that statement's execution. For example, if the function `f` modifies the global variable `x`, then the following statements will always print the value of `x` after `f` modifies it:

```
   i = f(x);
   printf( "x= %d\n", x );
```

Another rule you can count on is that the assignment to a variable on the left hand side of an assignment statement does not get modified prior to the use of that same variable on the right hand side of the expression. I.e., the following will not write a temporary value into variable `n` until it uses the previous value of `n` within the expression:

```
n = f(x) + g(x) - n;
```

Because the order of the production of side effects within an expression is undefined in C/C++, the result of the following code is generally undefined:

```
int incN( void )
{
   incN = n;
   n := n + 1;
}
      .
      .
      .
```

```
n = 2;
printf( "%d\n", incN() + n*2 );
```

The compiler is free to first call the *incN* function (so *n* will contain three prior to executing the sub-expression "n*2") or the compiler may first compute "n*2" and then call the *incN* function. As a result, one compilation of this statement could produce the output "8" while a different compilation of this statement might produce the output "6". In both cases *n* would contain three after the execution of the *writeln* statement, but the order of computation of the expression in the writeln statement could vary.

Don't make the mistake of thinking you can run some experiments to determine the order of evaluation. At the very best, such experiments will tell you the order a particular compiler uses. A different compiler may very well compute sub-expressions in a different order. Indeed, the same compiler might also compute the components of a subexpression differently based on the context of that subexpression. This means that a compiler might compute the expression using one ordering at one point in the program and using a different ordering somewhere else *in the same program*. Therefore, it is very dangerous to "determine" the ordering your particular compiler uses and rely on that ordering. Even if the compiler is consistent in the ordering of the computation of side effects, what's to prevent the compiler vendor from changing this in a later version of the compiler?

As noted earlier, most languages do guarantee that the computation of side effects completes before certain points in your program's execution. For example, almost every language guarantees the completion of all side effects by the time the statement containing the expression completes execution. The point at which a compiler guarantees that the computation of a side effect is completed is called a *sequence point*. The end of a statement is an example of a sequence point.

In the C programming language, there are several important sequence points in addition to the semicolon at the end of a statement. C provides several important sequence points within expressions, as well. Beyond the end of the statement containing an expression, C provides the following sequence points:

```
expression1, expression2                (the C comma operator in an expression)
expression1 && expression2              (the C logical AND operator)
expression1 || expression2              (the C logical OR operator)
expression1 ? expression2 : expression3 (the C conditional expression operator)
```

C[12] guarantees that all side effects in *expression1* are completed before the computation of *expression2* or *expression3* in these examples (note that for the conditional expression, C only evaluates one of *expression2* or *expression3*, so only the side effects of one of these sub-expressions is ever done on a given execution of the conditional expression).

To understand how side effects and sequence points can affect the operation of your program in non-obvious ways, consider the following example in C:

```
int array[6] = {0, 0, 0, 0, 0, 0};
int i;
   .
   .
   .
i = 0;
array[i] = i++;
```

12.C++ compilers generally provide the same sequence points at C, although the original C++ standard did not define any sequence points.

Note that C does not define a sequence point across the assignment operator. Therefore, the C language makes no guarantees about whether the expression "i" used as an index into *array* is evaluated before or after the program increments *i* on the right hand side of the assignment operator. Note that the fact that the "++" operator is a post increment operation only implies that "i++" returns the value of *i* prior to the increment; this does not guarantee that the compiler will use the pre-increment value of *i* anywhere else in the expression. The bottom line is that the last statement in this example could be semantically equivalent to either of the following statements:

```
   array[0] = i++;
-or-
   array[1] = i++;
```

The C language definition allows either form and, in particular, does not require the first form simply because the array index appears in the expression before the post-increment operator.

To control the semantics of the assignment to *array* in this example, you will have to ensure that no part of the expression depends upon the side-effects of some other part of the expression. That is, you cannot both use the value of *i* at one point in the expression and apply the post-increment operator to *i* in another part of the expression unless there is a sequence point between the two uses. Since no such sequence point exists between the two uses of *i* in this statement, the result is undefined by the C language standard (note that the standard actually says that the result is *undefined*; therefore, the compiler could legally substitute *any* value for *i* as the array index value, though most compilers will substitute the value of *i* before or after the increment occurs in this particular example).

Though this comment appears earlier in this section, it is worth being redundant to stress an important fact: operator precedence and associativity do not control when a computation takes place in an expression. Even though addition is left associative, the compiler may compute the value of the addition operator's right operand before it computes the value of the addition operator's left operand. Precedence and associativity control how the compiler arranges the computation to produce the final result. They do not control when the program computes the subcomponents of the expression. As long as the final computation produces the results one would expect on the basis of precedence and associativity, the compiler is free to compute the subcomponents in any order and at any time it pleases.

3.2.5.5: Translating C/C++ Expressions to Assembly Language

Armed with the information from the past several sections, it is now possible to intelligently describe how to convert complex arithmetic expressions into assembly language.

The conversion of C/C++ expressions into assembly language must take into consideration the issues of operator precedence, associativity, and sequence points. Fortunately, these rules only describe which operators you must apply to which operands and at which points you must complete the computation of side effects. They do not specify the order that you must use when computing the value of an expression (other than completing the computation of side effects before a given point). Therefore, you have a lot of latitude with respect to how you rearrange the computation during your conversion. Because the C/C++ programming language has some relaxed rules with regard to the order of computation in some expression, the result of a computation that relies on other side effects between a pair of sequence points is undefined. However, just because the language doesn't define the result, some programmers will go ahead and assume that the compiler computes results on a left to right basis within the statement. That is, if two subexpressions modify the value of some variable, the programmer will probably (though errantly) assume that the left-most side effect occurs first. So if you encounter such an undefined operation in a C/C++ code sequence that you're converting to assembly, the best suggestion is to compute the result using a left-to-right evaluation of the expression. Although this is no guarantee that you'll produce

what the original programmer intended, chances are better than 50/50 that this will produce the result that programmer was expecting.

A complex expression that is easy to convert to assembly language is one that involves three terms and two operators, for example:

```
w = w - y - z;
```

Clearly the straight-forward assembly language conversion of this statement will require two *sub* instructions. However, even with an expression as simple as this one, the conversion is not trivial. There are actually *two ways* to convert this from the statement above into assembly language:

```
        mov( w, eax );
        sub( y, eax );
        sub( z, eax );
        mov( eax, w );
and
        mov( y, eax );
        sub( z, eax );
        sub( eax, w );
```

The second conversion, since it is shorter, looks better. However, it produces an incorrect result. Associativity is the problem. The second sequence above computes “W = W - (Y - Z);” which is not the same as “W = (W - Y) - Z;”. How we place the parentheses around the subexpressions can affect the result. Note that if you are interested in a shorter form, you can use the following sequence:

```
        mov( y, eax );
        add( z, eax );
        sub( eax, w );
```

This computes “W=W-(Y+Z);”. This is equivalent to “W = (W - Y) - Z;”.

Precedence is another issue. Consider the C/C++ expression:

```
X = W * Y + Z;
```

Once again there are two ways we can evaluate this expression:

```
        X = (W * Y) + Z;
or
        X = W * (Y + Z);
```

However, C/C++’s precedence rules dictate the use of the first of these statements.

When converting an expression of this form into assembly language, you must be sure to compute the subexpression with the highest precedence first. The following example demonstrates this technique:

```
// w = x + y * z;

        mov( x, ebx );
        mov( y, eax );      // Must compute y*z first since “*”
        intmul( z, eax );   //  has higher precedence than “+”.
        add( ebx, eax );
        mov( eax, w );
```

The precedence and associativity rules determine the order of evaluation. Indirectly, these rules tell you where to place parentheses in an expression to determine the order of evaluation. Of course, you can always use parentheses to override the default precedence and associativity. However, the ultimate point is that your assembly code must complete certain operations before others to correctly compute the value of a given expression. The following examples demonstrate this principle:

```
// w = x - y - z

      mov( x, eax );   // All the same operator, so we need
      sub( y, eax );   //  to evaluate from left to right
      sub( z, eax );   //  because they all have the same
      mov( eax, w );   //  precedence and are left associative.

// w = x + y * z

      mov( y, eax );      // Must compute Y * Z first since
      intmul( z, eax );   // multiplication has a higher
      add( x, eax );      // precedence than addition.
      mov( eax, w );


// w = x / y - z

      mov( x, eax );      // Here we need to compute division
      cdq();              //  first since it has the highest
      idiv( y, edx:eax ); //  precedence.
      sub( z, eax );
      mov( eax, w );

// w = x * y * z

      mov( y, eax );      // Addition and multiplication are
      intmul( z, eax );   // commutative, therefore the order
      intmul( x, eax );   // of evaluation does not matter
      mov( eax, w );
```

There is one exception to the associativity rule. If an expression involves multiplication and division it is generally better to perform the multiplication first. For example, given an expression of the form:

```
W = X/Y * Z        // Note: this is x/y × z not x/(y × z)!
```

It is usually better to compute `X*Z` and then divide the result by `Y` rather than divide `X` by `Y` and multiply the quotient by `Z`. There are two reasons this approach is better. First, remember that the `imul` instruction always produces a 64 bit result (assuming 32 bit operands). By doing the multiplication first, you automatically *sign extend* the product into the EDX register so you do not have to sign extend EAX prior to the division. This saves the execution of the `cdq` instruction. A second reason for doing the multiplication first is to increase the accuracy of the computation. Remember, (integer) division often produces an inexact result. For example, if you compute 5/2 you will get the value two, not 2.5. Computing (5/2)*3 produces six. However, if you compute (5*3)/2 you get the value seven which is a little closer to the real quotient (7.5). Therefore, if you encounter an expression of the form:

```
w = x/y*z;
```

You can usually convert it to the assembly code:

```
mov( x, eax );
imul( z, eax );   // Note the use of IMUL, not INTMUL!
idiv( y, edx:eax );
mov( eax, w );
```

Of course, if the algorithm you're encoding depends on the truncation effect of the division operation, you cannot use this trick to improve the algorithm. Moral of the story: always make sure you fully understand any expression you are converting to assembly language. Obviously if the semantics dictate that you must perform the division first, do so.

Consider the following C/C++ statement:

```
w = x - y * x;
```

This is similar to a previous example except it uses subtraction rather than addition. Since subtraction is not commutative, you cannot compute `y * z` and then subtract `x` from this result. This tends to complicate the conversion a tiny amount. Rather than a straight forward multiply and addition sequence, you'll have to load `x` into a register, multiply `y` and `z` leaving their product in a different register, and then subtract this product from `x`, e.g.,

```
mov( x, ebx );
mov( y, eax );
intmul( x, eax );
sub( eax, ebx );
mov( ebx, w );
```

This is a trivial example that demonstrates the need for *temporary variables* in an expression. This code uses the EBX register to temporarily hold a copy of `x` until it computes the product of `y` and `z`. As your expressions increase in complexity, the need for temporaries grows. Consider the following C/C++ statement:

```
w = (a + b) * (y + z);
```

Following the normal rules of algebraic evaluation, you compute the subexpressions inside the parentheses (i.e., the two subexpressions with the highest precedence) first and set their values aside. When you've computed the values for both subexpressions you can compute their sum. One way to deal with complex expressions like this one is to reduce it to a sequence of simple expressions whose results wind up in temporary variables. For example, we can convert the single expression above into the following sequence:

```
Temp1 = a + b;
Temp2 = y + z;
w = Temp1 * Temp2;
```

Since converting simple expressions to assembly language is quite easy, it's now a snap to compute the former, complex, expression in assembly. The code is

```
mov( a, eax );
add( b, eax );
mov( eax, Temp1 );
mov( y, eax );
add( z, eax );
mov( eax, Temp2 );
mov( Temp1, eax );
intmul( Temp2, eax );
```

```
mov( eax, w );
```

Of course, this code is grossly inefficient and it requires that you declare a couple of temporary variables in your data segment. However, it is very easy to optimize this code by keeping temporary variables, as much as possible, in 80x86 registers. By using 80x86 registers to hold the temporary results this code becomes:

```
mov( a, eax );
add( b, eax );
mov( y, ebx );
add( z, ebx );
intmul( ebx, eax );
mov( eax, w );
```

Yet another example:

```
x = (y+z) * (a-b) / 10;
```

This can be converted to a set of four simple expressions:

```
Temp1 = (y+z)
Temp2 = (a-b)
Temp1 = Temp1 * Temp2
X = Temp1 / 10
```

You can convert these four simple expressions into the assembly language statements:

```
mov( y, eax );      // Compute eax = y+z
add( z, eax );
mov( a, ebx );      // Compute ebx = a-b
sub( b, ebx );
imul( ebx, eax );   // This also sign extends eax into edx.
idiv( 10, edx:eax );
mov( eax, x );
```

The most important thing to keep in mind is that you should attempt to keep temporary values, in registers. Remember, accessing an 80x86 register is much more efficient than accessing a memory location. Use memory locations to hold temporaries only if you've run out of registers to use.

Ultimately, converting a complex expression to assembly language is little different than solving the expression by hand. Instead of actually computing the result at each stage of the computation, you simply write the assembly code that computes the result. Since you were probably taught to compute only one operation at a time, this means that manual computation works on "simple expressions" that exist in a complex expression. Of course, converting those simple expressions to assembly is fairly trivial. Therefore, anyone who can solve a complex expression by hand can convert it to assembly language following the rules for simple expressions.

As noted earlier, this text will not consider the conversion of floating point expressions into 80x86 assembly language. Although the conversion is slightly different (because of the stack-oriented nature of the FPU register file), the conversion of floating point expressions into assembly language is so similar to the conversion of integer expressions that it isn't worth the space to discuss it here. For more insight into this type of expression conversion, please see *The Art of Assembly Language*.

3.2.6: Control Structures in C and Assembly Language

The C and C++ languages provide several high-level *structured* control statements. Among these, you will find the `if/else` statement, the `while` statement, the `do/while` statement, the `for` statement, the `break/continue/return` statements, and the `goto` statement. C/C++ also provides the function call, but we'll deal with that control structure later in this chapter. The C++ language provides exception handling facilities. However, as you're unlikely to encounter C++'s `try/catch` statements in Win32 API documentation, we won't bother discussing the conversion of those statements into assembly language in this book. If you have need to incorporate exception handling into your HLA programs, please check out the HLA `try..exception..endtry` statements in the HLA reference manual.

One advantage of a high level assembler like HLA is that it also provides high-level, structured, control statements. Although not as sophisticated as the similar statements you'll find in C/C++ (particularly with respect to the boolean expressions the C/C++ statements allow), it's fairly trivial to convert about 75-90% of the typical C/C++ control statements you'll encounter into assembly language (when using HLA).

This book will not cover the conversion of high level control structures into low-level assembly code (i.e., using conditional jumps and comparisons rather than the high-level control structures found in HLA). If you wish to use that conversion process and you're not comfortable with it, please see *The Art of Assembly Language* for more details.

For the most part, this book assumes that the reader is already an accomplished assembly language programmer. However, because many assembly language programmers might not have bothered to learn HLA's high level control structures, the following sections will describe the semantics of the HLA structured control statements in addition to describing how to convert C/C++ control structures into their equivalent assembly language statements. Since C/C++ does not provide as many control structures as C/C++, this section will not bother describing all of HLA's high level control structures - only those that have a C/C++ counterpart. For more details on HLA's high level control statements, please consult the HLA Reference Manual.

3.2.6.1: Boolean Expressions in HLA Statements

Several HLA statements require a boolean (true or false) expression to control their execution. Examples include the `if, while,` and `repeat..until` statements. The syntax for these boolean expressions represents the greatest limitation of the HLA high level control structures. In many cases you cannot convert the corresponding C/C++ statements directly into HLA code.

HLA boolean expressions always take the following forms[13]:

```
flag_specification
!flag_specification
register
!register
Boolean_variable
!Boolean_variable
mem_reg relop mem_reg_const
```

A *flag_specification* may be one of the following symbols:

- @c carry: True if the carry is set (1), false if the carry is clear (0).
- @nc no carry: True if the carry is clear (0), false if the carry is set (1).

13.There are a few additional forms, some of which we'll cover a little later in this section..

- @z zero: True if the zero flag is set, false if it is clear.
- @nz not zero: True if the zero flag is clear, false if it is set.
- @o overflow: True if the overflow flag is set, false if it is clear.
- @no no overflow: True if the overflow flag is clear, false if it is set.
- @s sign: True if the sign flag is set, false if it is clear.
- @ns no sign: True if the sign flag is clear, false if it is set.

A register operand can be any of the 8-bit, 16-bit, or 32-bit general purpose registers. The expression evaluates false if the register contains a zero; it evaluates true if the register contains a non-zero value.

If you specify a boolean variable as the expression, the program tests it for zero (false) or non-zero (true). Since HLA uses the values zero and one to represent false and true, respectively, the test works in an intuitive fashion. Note that HLA requires such variables be of type boolean. HLA rejects other data types. If you want to test some other type against zero/not zero, then use the general boolean expression discussed next.

The most general form of an HLA boolean expression has two operands and a relational operator. Table 3-8 lists the legal combinations.

Table 3-8: Relational Operators in HLA

Left Operand	Relational Operator	Right Operand
Memory Variable or Register	= or ==	Memory Variable, Register, or Constant
	<> or !=	
	<	
	<=	
	>	
	>=	

Note that both operands cannot be memory operands. In fact, if you think of the *Right Operand* as the source operand and the *Left Operand* as the destination operand, then the two operands must be the same that *cmp* instruction allows. This is the primary limitation to HLA boolean expressions and the biggest source of problems when converting C/C++ high level control statements into HLA code.

Like the *cmp* instruction, the two operands must also be the same size. That is, they must both be byte operands, they must both be word operands, or they must both be double word operands. If the right operand is a constant, it's value must be in the range that is compatible with the left operand.

There is one other issue: if the left operand is a register and the right operand is a positive constant or another register, HLA uses an *unsigned* comparison. You will have to use HLA's type coercion operator (e.g., "(type int32 eax)") if you wish to do a signed comparison.

Here are some examples of legal boolean expressions in HLA:

```
@c
Bool_var
```

```
al
ESI
EAX < EBX
EBX > 5
i32 < -2
i8 > 128
al < i8
```

HLA uses the "&&" operator to denote logical AND in a run-time boolean expression. This is a dyadic (two-operand) operator and the two operands must be legal run-time boolean expressions. This operator evaluates true if both operands evaluate to true. Example using an HLA *if* statement:

```
if( eax > 0 && ch = 'a' ) then

   mov( eax, ebx );
   mov( ' ', ch );

endif;
```

The two *mov* statements appearing here execute only if EAX is greater than zero *and* CH is equal to the character 'a'. If either of these conditions is false, then program execution skips over these *mov* instructions.

Note that the expressions on either side of the "&&" operator may be any legal boolean expression, these expressions don't have to be comparisons using the relational operators. For example, the following are all legal expressions:

```
@z && al in 5..10
al in 'a'..'z' && ebx
boolVar && !eax
```

HLA uses *short circuit evaluation* when compiling the "&&" operator. If the left-most operand evaluates false, then the code that HLA generates does not bother evaluating the second operand (since the whole expression must be false at that point). Therefore, in the last expression, the code will not check EAX against zero if *boolVar* contains false.

Note that an expression like "eax < 0 && ebx <> eax" is itself a legal boolean expression and, therefore, may appear as the left or right operand of the "&&" operator. Therefore, expressions like the following are perfectly legal:

eax < 0 && ebx <> eax && !ecx

The "&&" operator is left associative, so the code that HLA generates evaluates the expression above in a left-to-right fashion. If EAX is less than zero, the CPU will not test either of the remaining expressions. Likewise, if EAX is not less than zero but EBX is equal to EAX, this code will not evaluate the third expression since the whole expression is false regardless of ECX's value.

HLA uses the "||" operator to denote disjunction (logical OR) in a run-time boolean expression. Like the "&&" operator, this operator expects two legal run-time boolean expressions as operands. This operator evaluates true if either (or both) operands evaluate true. Like the "&&" operator, the disjunction operator uses short-circuit evaluation. If the left operand evaluates true, then the code that HLA generates doesn't bother to test the value of the second operand. Instead, the code will transfer to the location that handles the situation when the boolean expression evaluates true. Examples of legal expressions using the "||" operator:

```
@z || al = 10
```

```
al in 'a'..'z' || ebx
!boolVar || eax
```

As for the "&&" operator, the disjunction operator is left associative so multiple instances of the "||" operator may appear within the same expression. Should this be the case, the code that HLA generates will evaluate the expressions from left to right, e.g.,

eax < 0 || ebx <> eax || !ecx

The code above executes if either EAX is less than zero, EBX does not equal EAX, or ECX is zero. Note that if the first comparison is true, the code doesn't bother testing the other conditions. Likewise, if the first comparison is false and the second is true, the code doesn't bother checking to see if ECX is zero. The check for ECX equal to zero only occurs if the first two comparisons are false.

If both the conjunction and disjunction operators appear in the same expression then the "&&" operator takes precedence over the "||" operator. Consider the following expression:

eax < 0 || ebx <> eax && !ecx

The machine code HLA generates evaluates this as

eax < 0 || (ebx <> eax && !ecx)

If EAX is less than zero, then the code HLA generates does not bother to check the remainder of the expression, the entire expression evaluates true. However, if EAX is not less than zero, then both of the following conditions must evaluate true in order for the overall expression to evaluate true.

HLA allows you to use parentheses to surround sub-expressions involving "&&" and "||" if you need to adjust the precedence of the operators. Consider the following expression:

(eax < 0 || ebx <> eax) && !ecx

For this expression to evaluate true, ECX must contain zero and either EAX must be less than zero or EBX must not equal EAX. Contrast this to the result the expression produces without the parentheses.

HLA uses the "!" operator to denote logical negation. However, the "!" operator may only prefix a register or boolean variable; you may not use it as part of a larger expression (e.g., "!eax < 0"). To achieve logical negative of an existing boolean expression you must surround that expression with parentheses and prefix the parentheses with the "!" operator, e.g.,

```
!( eax < 0 )
```

This expression evaluates true if EAX is not less than zero.

The logical not operator is primarily useful for surrounding complex expressions involving the conjunction and disjunction operators. While it is occasionally useful for short expressions like the one above, it's usually easier (and more readable) to simply state the logic directly rather than convolute it with the logical not operator.

3.2.6.2: Converting C/C++ Boolean Expressions to HLA Boolean Expressions

Although, superficially, C/C++ boolean expressions that appear within control structures look very similar to those appearing in HLA high-level structured control statements, there are some fundamental differences that will create some conversion problems. Fortunately, most boolean expressions appearing in C/C++ control structures are relatively simple and almost translate directly into an equivalent HLA expression. Nevertheless, a large percentage of expressions will take a bit of work to properly convert to a form usable by HLA.

Although HLA provides boolean expressions involving relation and logical (and/or/not) operators, don't get the impression that HLA supports generic boolean expressions as C/C++ does. For example, an expression like "(x+y) > 10 || a*b < c" is perfectly legal in C/C++, but HLA doesn't allow an expression like this. You might wonder why HLA allows some operators but not others. There is a good reason why HLA supports only a limited number of operators: HLA supports all the operations that don't require the use of any temporary values (i.e., registers). HLA does not allow any code in an expression that would require the use of a register to hold a temporary value; i.e., HLA will not modify any register values behind the assembly programmer's back. This severely limits what HLA can do since subexpressions like "(x+y)" have to be computed in a temporary register (at least, on the 80x86). The previous section presented most of the operators that are legal in an HLA boolean expression. Unfortunately, of course, C/C++ does allow fairly complex arithmetic/boolean expressions within a structured control statement. This section provides some guidelines you can use to convert complex C/C++ arithmetic/boolean expressions to HLA.

The first thing to note is that HLA only allows operands that are legal in a *cmp* instruction around one of the relational operators. Specifically, HLA only allows the operands in Table 3-9 around a relational operator.

Table 3-9: Legal Operands to a Relational Operator in an HLA Expression

Left Operand	Relational Operator	Right Operand
reg	<	reg
reg	<=	mem
reg	=	const
mem	==	reg
mem	<>	const
	!=	
	>	
	>=	

If you need to convert a boolean expression like "(x+y) > 10" from C/C++ into HLA, the most common approach is to compute the sub-expression "(x+y)" and leave the result in a register, then you can compare that register against the value 10, e.g.,

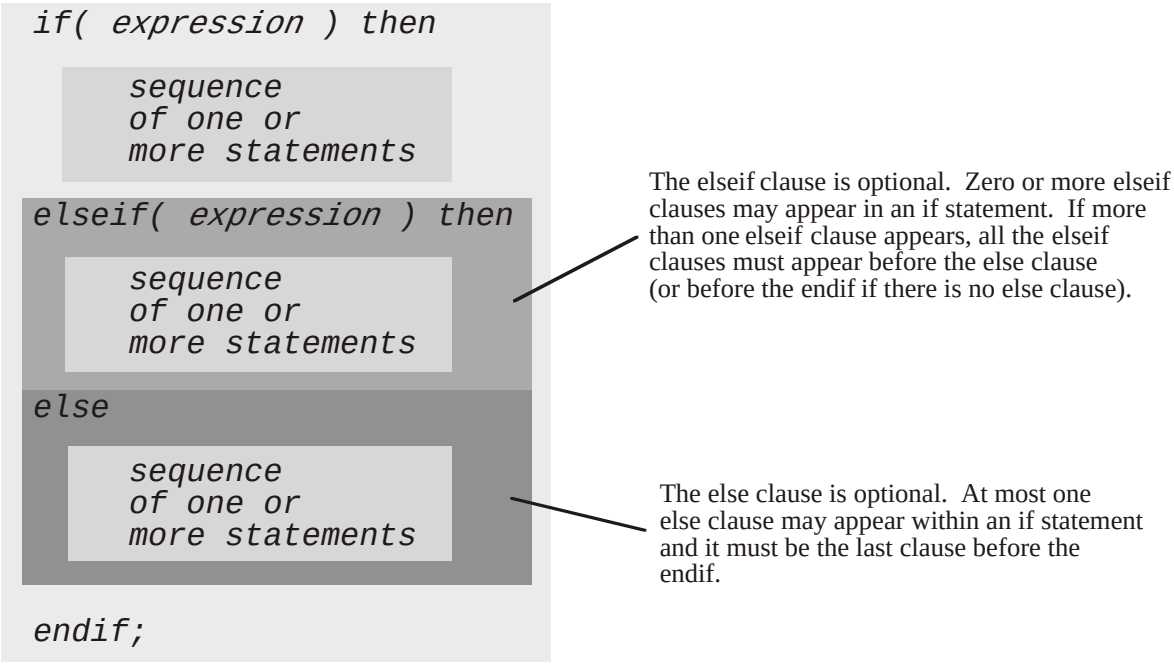
```
mov( x, eax );
add( y, eax );
if( eax > 10 ) then
   .
   .
   .
endif;
```

Unfortunately, the syntax of various high level control structures in HLA don't allow you to place the statements that compute the result before the control structure; we'll take a look at these problems in the sections that follow.

3.2.6.3: The IF Statement

The HLA IF statement uses the syntax shown in Table 3-2.

Figure 3-2: HLA IF Statement Syntax



The expressions appearing in an *if* statement must take one of the forms from the previous sections. If the boolean expression is true, the code after the *then* executes, otherwise control transfers to the next *elseif* or *else* clause in the statement.

Since the *elseif* and *else* clauses are optional, an *if* statement could take the form of a single *if..then* clause, followed by a sequence of statements, and a closing *endif* clause. The following is such a statement:

```
if( eax = 0 ) then

   stdout.put( "error: NULL value", nl );

endif;
```

If, during program execution, the expression evaluates true, then the code between the *then* and the *endif* executes. If the expression evaluates false, then the program skips over the code between the *then* and the *endif*.

Another common form of the *if* statement has a single *else* clause. The following is an example of an *if* statement with an optional *else* clause:

```
if( eax = 0 ) then

   stdout.put( "error: NULL pointer encountered", nl );

else

   stdout.put( "Pointer is valid", nl );

endif;
```

If the expression evaluates true, the code between the *then* and the *else* executes; otherwise the code between the *else* and the *endif* clauses executes.

You can create sophisticated decision-making logic by incorporating the *elseif* clause into an *if* statement. For example, if the CH register contains a character value, you can select from a menu of items using code like the following:

```
if( ch = 'a' ) then

   stdout.put( "You selected the 'a' menu item", nl );

elseif( ch = 'b' ) then

   stdout.put( "You selected the 'b' menu item", nl );

elseif( ch = 'c' ) then

   stdout.put( "You selected the 'c' menu item", nl );

else

   stdout.put( "Error: illegal menu item selection", nl );

endif;
```

Although this simple example doesn't demonstrate it, HLA does not require an *else* clause at the end of a sequence of *elseif* clauses. However, when making multi-way decisions, it's always a good idea to provide an *else* clause just in case an error arises. Even if you think it's impossible for the *else* clause to execute, just keep in mind that future modifications to the code could void this assertion, so it's a good idea to have error reporting statements in your code.

The C/C++ *if* statement is similar, but certainly not identical to, the HLA *if* statement. First of all, the C/C++ if statement is based on an older language design that allows only a single statement after an *if* or *else*. That is, C/C++ supports the following syntaxes for the *if/else* statement:

```
   if( boolean_expression)
      << single statement >>;

   if( boolean_expression )
      << single statement >>;
   else
      << single statement >>;
```

If you need to attach more than a single statement to a C/C++ *if* or *else*, you have to use a compound statement. A compound statement consists of a sequence of zero or more statements surrounded by braces. This means that there are six possible forms of the *if* statement you will find in a typical C/C++ program, as the following syntactical examples demonstrate:

```
1)
   if( boolean_expression)
      << single statement >>;

2)
   if( boolean_expression)
```

```
   {
      << zero or more statements >>
   }

3)
   if( boolean_expression )
      << single statement >>;
   else
      << single statement >>;

4)
   if( boolean_expression )
   {
      << zero or more statements >>
   }
   else
      << single statement >>;

5)
   if( boolean_expression )
      << single statement >>;
   else
   {
      << zero or more statements >>
   }

6)
   if( boolean_expression )
   {
      << zero or more statements >>
   }
   else
   {
      << zero or more statements >>
   }
```

To convert either of the first two forms to HLA is relatively easy. Simply convert the boolean expression to HLA form (including placing any necessary arithmetic computations before the *if* statement), convert the statement or statements attached to the *if* to their HLA equivalents, and then place an *endif* after the last statement attached to the *if*. Here are a couple of examples that demonstrate this conversion for the first two cases:

```
// if( a >= 0 )
//     ++a;

   if( a > 0 ) then

      inc( a );

   endif;

// if( (x*4) >= y && z < -5 )
// {
//    x = x - y;
//    ++z;
//  }
```

```
   mov( x, eax );
   shl( 2, eax );  // x*4
   if( eax >= y && z < -5 ) then

      mov( y, eax );
      sub( eax, x );
      inc( eax );

   endif;
```

Converting one of the other *if/else* forms from C/C++ to HLA is done in a similar fashion except, of course, you also have to include the *else* section in the HLA translation. Here's an example that demonstrates this:

```
// if( a < 256 )
// {
//    ++a;
//    --b;
// }
// else
// {
//    --a;
//    ++b;
// }

   if( a < 256 ) then

      inc( a );
      dec( b );

   else

      dec( a );
      inc( b );

   endif;
```

The C/C++ language does not directly support an *elseif* clause as HLA does, however, C/C++ programs often contain "else if" chains that you may convert to an HLA *elseif* clause. The following example demonstrates this conversion:

```
// if( x >= (y | z))
//    ++x;
//  else if( x >= 10 )
//    --x;
//  else
//  {
//     ++y;
//     --z;
//  }

   mov( y, eax );
   or( z, eax );
   if( x >= eax ) then
```

```
      inc( x );

   elseif( x >= 10 ) then

      dec( x );

   else

      inc( y );
      dec( z );

   endif;
```

Sometimes a C/C++ else-if chain can create some conversion problems. For example, suppose that the boolean expression in the "else if" of this example was "x >= (y & z)" rather than an expression that is trivially convertible to HLA. Unfortunately, you cannot place the computation of the temporary results immediately before the *elseif* in the HLA code (since that section of code executes when *if* clause evaluates true). You could place the computation before the *if* and leave the value in an untouched register, but this scheme has a couple of disadvantages - first, you always compute the result even when it's not necessary (e.g., when the *if* expression evaluates true), second, it consumes a register which is not good considering how few registers there are on the 80x86. A better solution is to use an HLA nested *if* rather than an *elseif*, e.g.,

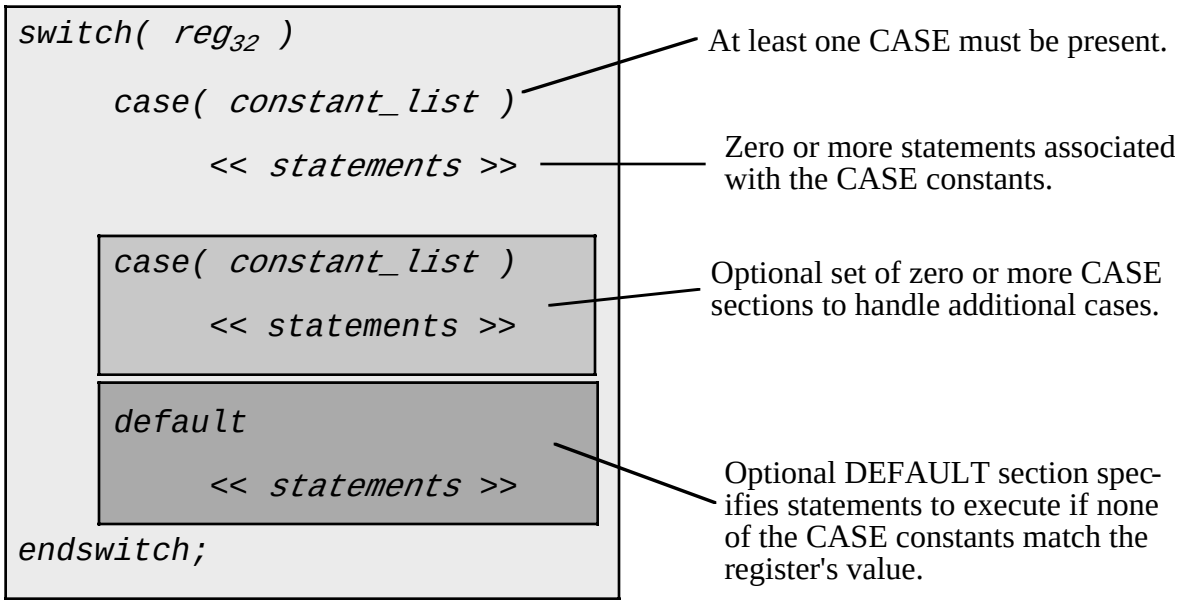
```
// if( x >= (y | z))
//    ++x;
//  else if( x >= (y & z) )
//    --x;
//  else
//  {
//     ++y;
//     --z;
//  }

   mov( y, eax );
   or( z, eax );
   if( x >= eax ) then

      inc( x );

   else

      mov( y, eax );
      and( z, eax );
      if( x >= eax ) then

         dec( x );

      else

         inc( y );
         dec( z );

      endif;

   endif;
```

3.2.6.4: The SWITCH/CASE Statement

The HLA programming language doesn't directly provide a multi-way decision statement (commonly known as a *switch* or *case* statement). However, the HLA Standard Library provides a *switch / case / default / endcase* macro that provides this high level control statement in HLA. If you include the *hll.hhf* header file (which *stdlib.hhf* automatically includes for you), then you can use the *switch* statement exactly as though it were a part of the HLA language.

The HLA Standard Library *switch* statement has the following syntax:

Figure 3-3: Syntax for the Switch..case..default..endswitch Statement



Like most HLA high level language statements, there are several restrictions on the *switch* statement. First of all, the *switch* clause does not allow a general expression as the selection value. The *switch* clause will only allow a value in a 32-bit general purpose register. In general you should only use EAX, EBX, ECX, EDX, ESI, and EDI because EBP and ESP are reserved for special purposes.

The second restriction is that the HLA *switch* statement supports a maximum of 256 different case values. Few *switch* statements use anywhere near this number, so this shouldn't prove to be a problem. Note that each *case* in Figure 3-3 allows a constant list. This could be a single unsigned integer value or a comma separated list of values, e.g.,

```
      case( 10 )
-or-
      case( 5, 6, 8 )
```

Each value in the list of constants counts as one case constant towards the maximum of 256 possible constants. So the second *case* clause above contributes three constants towards the total maximum of 256 constants.

Another restriction on the HLA *switch* statement is that the difference between the largest and smallest values in the case list must be 1,024. Therefore, you cannot have *case*s (in the same *switch* statement) with values like 1, 10, 100, 1,000, and 10,000 since the difference between the smallest and largest values, 9999, exceeds 1,024.

The *default* section, if it appears in a *switch* statement, must be the last section in the *switch* statement. If no *default* section is present and the value in the 32-bit register does not match one of the *case* constants, then control transfers to the first statement following the *endswitch* clause.

Here is a typical example of a *switch..endswitch* statement:

```
switch( eax )

   case( 1 )

      stdout.put( "Selection #1:" nl );
      << Code for case #1 >>

   case( 2, 3 )

      stdout.put( "Selections (2) and (3):" nl );
      << code for cases 2 & 3 >>

   case( 5,6,7,8,9 )

      stdout.put( "Selections (5)..(9)" nl );
      << code for cases 5..9 >

   default

      stdout.put( "Selection outside range 1..9" nl );
      << default case code >>

endswitch;
```

The *switch* statement in a program lets your code choose one of several different code paths depending upon the value of the case selection variable. Among other things, the *switch* statement is ideal for processing user input that selects a menu item and executes different code depending on the user's selection.

The HLA *switch* statement actually supports the semantics of the Pascal *case* statement (as well as multi-way selection statements found in various other languages). The semantics of a C/C++ *switch* statement are slightly different. As it turns out, HLA's *switch* macro provides an option for selecting either Pascal or C/C++ semantics. The *hll.hhf* header file defines a special compile-time boolean variable, *hll.cswitch*, that controls which form of the *switch* statement HLA will use. If this compile-time variable contains false (the default), then HLA uses Pascal semantics for the *switch* statement. If this compile-time variable contains true, then HLA uses C/C++ semantics. You may set this compile-time variable to true or false with either of the following two statements:

```
?hll.cswitch := true;  // Enable C/C++ semantics for the switch statement.
?hll.cswitch := false; // Enable Pascal semantics for the switch statement.
```

The difference between C/C++ and Pascal semantics has to do with what will happen when the statements within some *case* block reach the end of that block (by hitting another *case* or the *default* clause). When using Pascal semantics, HLA automatically transfers control to the first statement following the *endswitch* clause upon hitting a new case. In the previous example, if EAX had contained one, then the switch statement would execute the code sequence:

```
      stdout.put( "Selection #1:" nl );
      << Code for case #1 >>
```

Immediately after the execution of this code, control transfers to the first statement following the *endswitch* (since the next statement following this fragment is the "case(2,3)" clause).

If you select C/C++ semantics by setting the *hll.cswitch* compile-time variable to true, then control does not automatically transfer to the bottom of the *switch* statement; instead, control falls into the first statement of the next *case* clause. In order to transfer control to the first statement following the *endswitch* at the end of a *case* section, you must explicitly place a *break* statement in the code, e.g.,

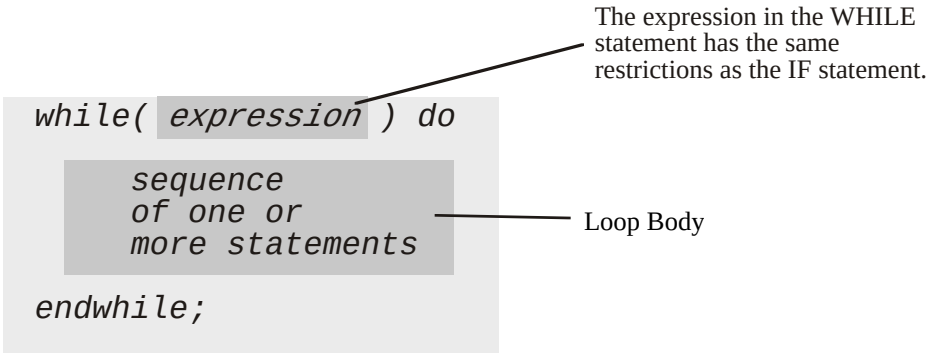
```
?hll.cswitch := true;  // Enable C/C++ semantics for the switch statement.
switch( eax )

   case( 1 )

      stdout.put( "Selection #1:" nl );
      << Code for case #1 >>
      break;

   case( 2, 3 )

      stdout.put( "Selections (2) and (3):" nl );
      << code for cases 2 & 3 >>
      break;

   case( 5,6,7,8,9 )

      stdout.put( "Selections (5)..(9)" nl );
      << code for cases 5..9 >
      break;

   default

      stdout.put( "Selection outside range 1..9" nl );
      << default case code >>

endswitch;
```

Note that you can alternately switch between C/C++ and Pascal semantics throughout your code by setting the *hll.cswitch* compile-time variable to true or false at various points throughout your code. However, as this makes the code harder to read, it's generally not a good idea to do this on a frequent basis. You should pick one form or the other and attempt to stick with it as much as possible. Pascal semantics are actually a little bit nicer (and safer) plus you get to continue using the *break* statement to break out of a loop containing a *switch* statement. On the other hand, some C/C++ *switch* statements need the ability to flow from one case to another, so if you're translating such a statement from C/C++ to HLA, the C/C++ *switch* statement format is easier to deal with. Of course, the purpose of this chapter is not to teach you how to convert a C/C++ Windows program to HLA, but rather to help you read and understand C/C++ documentation. In real life, if you have to convert a C/C++ *switch* statement to assembly language you're probably better off explicitly creating a jump table and using an indirect jump implementation of the *switch* statement (see *The Art of Assembly Language* for details).

3.2.6.5: The WHILE Loop

The HLA *while* statement uses the basic syntax shown in Figure 3-4.

Figure 3-4: HLA WHILE Statement Syntax



The *while* statement evaluates the boolean expression. If it is false, control immediately transfers to the first statement following the *endwhile* clause. If the value of the expression is true, then the CPU executes the body of the loop. After the loop body executes, control transfers back to the top of the loop where the *while* statement retests the loop control expression. This process repeats until the expression evaluates false.

The C/C++ statement uses a similar syntax and identical semantics. There are two principle differences between the HLA while loop and the C/C++ variant: (1) HLA uses "while(expr) do ... endwhile;" whereas C/C++ uses "while(expr) *single_statement*;", as with the C/C++ *if* statement, if you want to attach more than a single statement to the *while* you have to create a compound statement (using braces); (2) HLA's boolean expressions are limited compared to C/C++ boolean expressions (see the discussion in the section on converting boolean expressions from C/C++ to HLA and the section on the *if* statement for details).

One problem with converting C/C++ statements to HLA is the conversion of complex boolean expressions. Unlike an *if* statement, we cannot simply compute portions of a boolean expression prior to the actual test in the *while* statement, i.e., the following conversion doesn't work:

```
// while( (x+y) < z )
// {
//    printf( "x=%d\n", x );
//    ++x;
//    y = y + x;
// }

   mov( x, eax );   // Note: this won't work!
   add( y, eax );
   while( eax < z ) do
      stdout.put( "x=", x, nl );
      inc( x );
      mov( x, eax );
      add( eax, y );
   endwhile;
```

The problem with this conversion, of course, is that the computation of "x+y" needed in the boolean expression only occurs once, when the loop first executes, not on each iteration as is the case with the original C/C++ code. The easiest way to solve this problem is to use the HLA *forever..endfor* loop and a *breakif* statement:

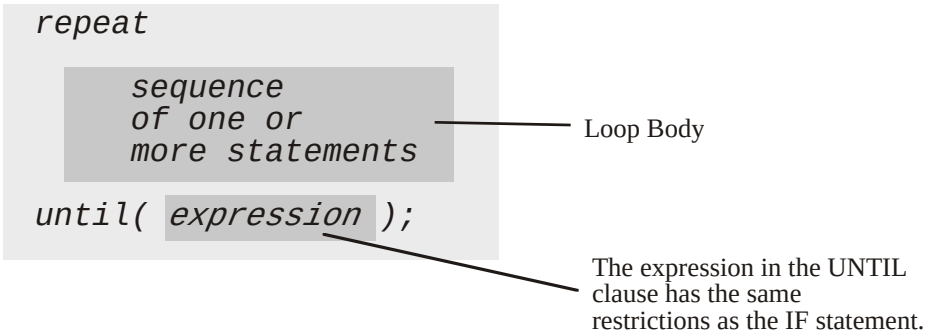
```
// while( (x+y) < z )
// {
//    printf( "x=%d\n", x );
//    ++x;
//    y = y + x;
// }

   forever
      mov( x, eax );
      add( y, eax );
    breakif( eax < z );
      stdout.put( "x=", x, nl );
      inc( x );
      mov( x, eax );
      add( eax, y );
   endfor;
```

3.2.6.6: The DO..WHILE Loop

The C/C++ *do..while* loop is similar to the *while* loop except it tests for loop termination at the bottom of the loop rather than at the top of the loop (i.e., it executes the statements in the loop body at least once, regardless of the value of the boolean control expression the first time the program computes it). Like the *while* loop, the *do..while* loop repeats the execution of the loop body as long as the boolean expression evaluates true. HLA does not provide an exact equivalent of the *do..while* loop, but it does provide a *repeat..until* loop. The difference between these two loops is that a *do..while* loop repeats as long as (while) the expression evaluates true, the *repeat..until* loop repeats until the expression evaluates true (that is, it repeats the loop as long as the expression evaluates false).

The HLA *repeat..until* statement uses the syntax shown in Figure 3-5.

Figure 3-5: HLA repeat..until Statement Syntax



To convert a C/C++ *do..while* statement to an HLA *repeat..until* statement, you must adjust for the semantics of the loop termination condition. Most of the time, the conversion is immediately obvious; in those few cases where you've got a complex boolean expression whose negation is not instantly obvious, you can always use the HLA "!(...)" (not) operator to negate the result of the boolean expression, e.g.,

```
// do
// {
//    <<some code fragment>>
```

```
// }while( (x < 10) && (y > 5 ));

   repeat

      <<some code fragments converted to HLA>>

   until( !((x<10) && (y>5)) );
```

One advantage of the *do..while* loop over C/C++'s while loop is that statements appearing immediately before the *while* clause (and after the *do* clause) will generally execute on each iteration of the loop. Therefore, if you've got a complex boolean expression that tests for loop termination, you may place the computation of portions of that expression immediately before the HLA *until* clause, e.g.,

```
// do
// {
//    printf( "x=%d\n", x );
//    ++x;
//    y = y + x;
// }while( (x+y) < z )

   repeat
      stdout.put( "x=", x, nl );
      inc( x );
      mov( x, eax );
      add( eax, y );

      mov( x, eax );
      add( y, eax );
   until( !(eax < z) );
```

The only time this will not work is if there is a *continue* (or an HLA *continueif*) statement in the loop. The *continue* statement directly transfers control to the loop termination test in the *until* clause. Since *continue* statements in C/C++ appear so infrequently, the best solution is to replace the *continue* with a *jmp* instruction that transfers control the first statement that begins the execution of the termination test expression.

3.2.6.7: The C/C++ FOR Loop

The C/C++ statement *for* statement is a specialized form of the *while* loop. It should come as no surprise, then, that the conversion to HLA is very similar to that for the *while* loop conversion. The syntax for the C/C++ for loop is the following:

```
   for( expression1; expression2; expression3 )
      statement;
```

This C/C++ statement is complete equivalent to the following C/C++ sequence:

```
   expression1;
   while( expression2 )
   {
      statement;
      expression3;
   }
```

Although you can convert a C/C++ *for* statement to an HLA *while* loop, HLA provides a *for* statement that is syntactically similar to the C/C++ *for* statement. Therefore, it's generally easiest to convert such C/C++ statements into HLA *for* statements. The HLA *for* loop takes the following general form:

```
for( Initial_Stmt; Termination_Expression; Post_Body_Statement ) do

   << Loop Body >>

endfor;
```

The following gives a complete example:

```
for( mov( 0, i ); i < 10; add(1, i )) do

   stdout.put( "i=", i, nl );

endfor;


// The above, rewritten as a while loop, becomes:

mov( 0, i );
while( i < 10 ) do

   stdout.put( "i=", i, nl );

   add( 1, i );

endwhile;
```

There are a couple of important differences between the HLA for loop and the C/C++ for loop. First of all, of course, the boolean loop control expression that HLA supports has the usual restrictions. If you've got a complex boolean expression in a C/C++ loop, your best bet is to convert the *for* loop into a C/C++ *while* loop and then convert that *while* loop into an HLA *forever..endfor* loop as the section on the *while* loop describes.

The other difference between the C/C++ and HLA for loops is the fact that C/C++ supports arbitrary arithmetic expressions for the first and third operands whereas HLA supports a single HLA statement. 90% of the C/C++ for loops you'll encounter will simply assign a constant to a variable in the first expression and increment (or decrement) that variable in the third expression. Such for loops are very easy to convert to HLA as the following example demonstrates:

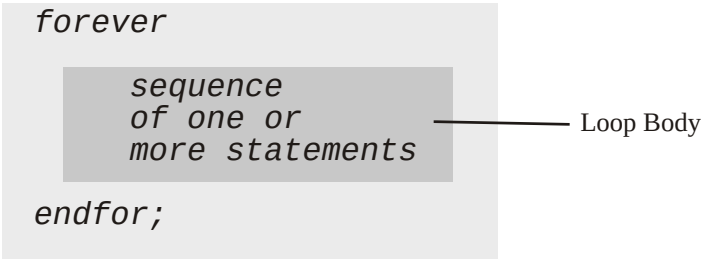
```
// for( i=0; i<10; ++i )
// {
//    printf( "i=%d\n", i );
// }

   for( mov( 0, i ); i<10; inc(i) ) do
      stdout.put( "i=", i , nl );
   endfor;
```

C/C++ allows a bizarre form of the for statement to create an infinite loop. The C/C++ convention for an infinite loop uses the following syntax:

```
for(;;)
   statement;
```

HLA does not allow this same syntax for it's *for* loop. Instead, HLA provides an explicit statement for creating infinite loops: the *forever..endfor* statement. Figure 3-6 shows the syntax for the *forever* statement.

Figure 3-6: HLA forever..endfor Loop Syntax



Although *for(;;)* and *forever..endfor*, by themselves, create infinite loops, the truth is that most of the time a program that employs these statements also uses a *break, breakif*, or *return* statement in order to exit the loop somewhere in the middle of the loop. The next section discusses the *break* and *breakif* statements. A little bit later we'll look at C/C++'s *return* statement.

3.2.6.8: Break and Continue

C/C++ supports two specialized forms of the goto statement that immediately exits, or repeats the execution of, the loop containing these statements. The *break* statement exits the loop that contains the statement; the *continue* statement transfers control to the loop control expression (or simply to the top of the loop in the case of the infinite loop). As you've seen earlier, the *break* statement also ends a case sequence in the C/C++ switch statement.

HLA also provides the *break* and *continue* statements that have the same semantics within a loop. Therefore, you can trivially translate these two statements from C/C++ to HLA. HLA also provides *breakif* and *continueif* statements that will test a boolean expression and execute the *break* or *continue* only if the expression evaluates true. Although C/C++ doesn't provide a direct counterpart to these two HLA statements, you'll often see C/C++ statements like the following that you can immediately translate to an HLA *breakif* or *continueif* statement:

```
if( C_expression ) break;
if( C_expression ) continue;
```

3.2.6.9: The GOTO Statement

The C/C++ *goto* statement translates directly into an 80x86 *jmp* instruction. A C/C++ *goto* statement typically takes the following form:

```
goto someLabel;
   .
   .
   .
```

```
someLabel:  // The label may appear before the goto statement!
```

This usually translates to the following HLA code:

```
    jmp someLabel;
        .
        .
        .
someLabel:
```

The only difference, besides substituting *jmp* for *goto*, is the fact that *goto* labels have their own *namespace* in C/C++. In HLA, however, statement labels share the same namespace as other local variables. Therefore, it is possible (though rare) that you'll get a "duplicate name" error if you use the same name in your HLA code that appears in the C/C++ program. If this happens, make a minor change to the statement label when translating the code to HLA.

3.3: Function Calls, Parameters, and the Win32 Interface

This section begins the second major portion of this chapter and, in fact, represents the most important material in this chapter from the perspective of an assembly language programmer: how C/C++ function calls translate into assembly language and how an HLA programmer would call a function written in C/C++. This information represents the major point of this chapter since all Win32 API calls are calls to C code. Furthermore, most Windows documentation that explains the Win32 API explains it in terms of C/C++ function calls, in order to understand how one makes calls to the Win32 API from assembly language, you must understand how C/C++ implements these function calls. Explaining that is the purpose of this section.

3.3.1: C Versus C++ Functions

There are some very important differences, both semantic and syntactical, between functions written in C and functions written in C++. The Win32 API uses the C calling and naming convention. Therefore, all the Win32 API documentation also uses the C calling and naming convention. Therefore, that's what we will concentrate on in this chapter.

C++ functions do offer several advantages over C functions. Function overloading is a good example of such a feature. However, function overloading (using the same function name for different functions and differentiating the actual functions by their parameter lists) requires the use of a facility known as *name mangling* in order to generate unique names that the linker can use. Unfortunately, there is no standard for name mangling among C++ compilers, so every one of them does it differently. Therefore, you rarely see assembly code (or other languages for that matter) interfacing with C++ functions.

In order to allow mixed-language programming with C++ (that is, the use of multiple programming languages on the same project), the C++ language defines a special "C" function syntax that allows you to tell the compiler to generate C linkage rather than C++. This is done with the C++ *extern* attribute:

```
extern "C"
{
    extern char* RetHW( void );
};
```

Please consult a C++ reference manual or your compiler's documentation for more details. Since the Win32 API doesn't use the C++ calling convention, we won't consider it any farther here.

Another useful C++ feature that this chapter will discuss, when appropriate, is pass by reference parameters (since HLA also supports this feature). However, the Win32 API doesn't use any C++ features, so when this chapter gets around to discussing pass by reference parameters, it will mainly be for your own edification.

3.3.2: The Intel 80x86 ABI (Application Binary Interface)

Several years ago, Intel designed what is known as the 80x86 Application Binary Interface, or ABI. The purpose of the ABI was to provide a standard that compiler designers to use to ensure interoperability between modules written in different languages. The ABI specifies what registers a function call should preserve (and which registers a function can modify without preserving), where functions return their results, alignment of data objects in structures, and several other conventions. Since Microsoft's C/C++ compilers (the ones used to compile Windows) adhere to these conventions, you'll want to be familiar with this ABI since the Win32 API uses it.

3.3.2.1: Register Preservation and Scratch Registers in Win32 Calls

The Intel 80x86 ABI specifies that functions must preserve the values of certain registers across a function call. If the function needs to modify the value of any of those registers, it must save the register's value and restore it before returning to the caller. The registers that must be preserved across calls are EBX, ESI, EDI, and EBP. This means two things to an assembly language programmer calling an Win32 function: first of all, Windows preserves the values of these registers across a Win32 API call, so you can place values in these registers, make an OS call, and be confident that they contain the same value upon return. The second implication has to do with *callback functions*. A callback function is a function you write whose address you pass to Windows. At various times Windows may choose to call that function directly. Such callback functions must obey the register preservation rules of the Intel 80x86 ABI. In particular, such callback functions must preserve the value of the EBX, ESI, EDI, and EBP registers.

On the flip side, the Intel 80x86 ABI specifies that a function may freely modify the values of the EAX, ECX and EDX registers without preserving them. This means that you can generally count on Win32 API functions disturbing the values in these registers; as you'll see in a moment, most Win32 API functions return a function result in the EAX register, so it's almost always wiped out. However, most Win32 API functions wipe out the values in ECX and EDX as well. If you need the values of any of these registers preserved across a Win32 API call, you must save their values yourself.

3.3.2.2: The Stack Pointer (ESP)

The ESP register is a special case. Function calls to the Win32 API generally do not preserve ESP because they remove any parameters from the stack that you push onto the stack prior to calling the API function. However, you can generally assume that ESP is pointing at an appropriate top of stack upon return from the function. In particular, any values you push onto the stack before pushing any API parameters (e.g., register values you want to preserve) will still be sitting on the stack when the function returns. Functions that follow the Intel 80x86 ABI do not arbitrarily mess with the value in the ESP register.

All Win32 API functions assume that the stack is aligned on a double-word boundary (that is, ESP contains a value that is an even multiple of four). If you call a Win32 API function and ESP is not aligned at a double-word address, the Win32 API function will fail. By default, HLA automatically emits code at the beginning of each procedure to ensure that ESP contains a value that is an even multiple of four bytes. However, many program-

mers choose to disable this code (to make their programs slightly more efficient). If you do this, always make sure that ESP contains a value whose L.O. two bits contain zeros (that is, an even multiple of four) before calling any Win32 API functions.

3.3.2.3: The Direction Flag

All Win32 functions assume that the direction flag is clear when you call them. The Win32 programming convention is to set the direction flag when you need it set and then immediately clear it when you are done using it in that state. Therefore, in all code where you have not explicitly set the direction flag yourself, you can assume that the direction flag is clear. You code should adhere to this policy as well (and always make sure the direction flag is clear when you make a Win32 API call). You can also assume that the direction flag is clear whenever Windows calls one of your callback routines.

3.3.2.4: Function Return Results

Table 3-10 lists the places that functions should return their result (depending on the size of the function return result). The Win32 API generally adheres to this convention. If a function returns more than eight bytes, Win32 API functions generally require that you pass a pointer (i.e., the address of) some block of memory where the function will store the final result.

Table 3-10: 80x86 ABI Function Return Result Locations

Size of Function Result in Bytes	Returned Here
1	al
2	ax
4	eax
8	edx:eax
other	See Compiler Documentation

3.3.2.5: Data Alignment and Padding

The Intel 80x86 ABI generally expects objects to appear at addresses in memory that are an even multiple of their natural size up to four bytes in length (i.e., byte objects may appear at any address, word objects always appear at even addresses, and larger objects are aligned on double-word addresses). This is true for static objects, automatic variables (local variables within a function), and fields within structures. Although this convention is easily circumvented by setting compiler options, the Win32 API pretty much adheres to this convention throughout.

If an object would normally start at an address that is not an even multiple of it's *natural size*[14] (up to four bytes), then the Microsoft C compiler will align that object at the next highest address that is an even multiple of

14.The natural size of an object is the size of the object if it's a scalar, the size of an element if it's an array, or the size of the largest field (up to four bytes) if it's a structure.

the object's native size. For data, the compiler usually fills (*pads*) the empty bytes with zeros, though you should never count on the values (or even presence) of padding bytes.

Parameters passed on the stack to a function are a special case. Parameters are always an even multiple of four bytes (this is done to ensure that the stack remains double-word aligned in memory). If you pass a parameter that is smaller than four bytes to some function, the Microsoft C compiler will pad it out to exactly four bytes. Likewise, if you pass a larger object that is not an even multiple of four bytes long, the compiler will pad the object with extra bytes so its length is an even multiple of four bytes long.

For information on padding within structures, please see the section on the `struct` data type earlier in this chapter.

3.3.3: The C, Pascal, and Stdcall Calling Conventions

There are many different function calling conventions in use today. Of these different calling conventions, three are of interest to us, the so-called *C, Pascal,* and *Stdcall* calling conventions. The C and Stdcall calling conventions are of interest because they're the ones that Win32 API calls use. The Pascal calling convention is of interest because that's the default calling convention that HLA uses.

The Pascal calling convention is probably the most efficient of the three and the easiest to understand. In the Pascal calling sequence, a compiler (or human programmer writing assembly code) pushes parameters on the stack as they are encountered in the parameter list when processing the parameters in a left-to-right fashion. Another nice feature of the Pascal calling sequence is that the procedure/function is responsible for removing the parameters from the stack upon return from the procedure; so the caller doesn't have to explicitly do this upon return. As an example, consider the following HLA procedure prototype and invocation:

```
// Note: the "@pascal" attribute is optional, since HLA generally uses
// the pascal calling convention by default.

procedure proc( i:int32; j:int32; k:int32 ); @pascal; @external;
   .
   .
   .
proc( 5, a, eax );
```

Whenever HLA encounters the high-level call to `proc` appearing in this example, it emits the following "pure" assembly code:

```
pushd( 5 );
push( a );    // Assumption: a is a 32-bit variable that is type compatible with int32
push( eax );
call proc;
```

Note that you have the choice of using HLA's high-level calling syntax or manually pushing the parameters and calling the procedure directly. HLA allows either form; the high-level calling syntax is generally easier to read and understand and it's less likely you'll make a mistake (that invariably hangs the program) when using the high level syntax. Some assembly programmers, however, prefer the low-level syntax since it doesn't hide what is going on.

The C calling convention does two things differently than the Pascal calling convention. First of all, C functions push their parameters in the opposite order than Pascal (e.g., from right to left). The second difference is that C functions do not automatically pop their parameters from the stack upon return. The advantage of the C

calling convention is that it allows a variable number of parameters (e.g., for C's *printf* function). However, the price for this extra convenience is reduced efficiency (since the caller has to execute extra instructions to remove the parameters from the stack).

Although Windows is mostly written in C, most of the Win32 API functions do not use the C calling convention. In fact, only the API functions that support a variable number of parameters (e.g., *wsprintf*) use the C calling convention. If you need to make a call to one of these API functions (or you want to call some other function that uses the C calling convention), then you've got to ensure that you push the parameters on the stack in the reverse order of their declaration and you've got to remove them from the stack when the function returns. E.g.,

```
// int cProc( int i, int j, int k );
//    .
//    .
//    .
// cProc( a, 5, 2 );

   pushd( 2 );     // push last parameter first!
   pushd( 5 );
   push( a );      //assumes a is a dword variable.
   call cProc;
   add( 12, esp ); // Remove three dword parameters from stack upon return.
```

HLA supports the C calling convention using the *@cdecl* procedure attribute, e.g.,

```
procedure cProc( i:int32; j:int32; k:int32 ); @cdecl; @external;
```

HLA's high-level procedure call syntax will automatically push the actual parameters on the stack in the appropriate order (i.e., in reverse). However, you are still responsible for removing the parameter data from the stack upon returning from the procedure call:

```
cProc( a, 5, 2 );  // Pushes 2, then 5, then a onto the stack
add( 12, esp );    // Remove parameter data from the stack.
```

Don't forget that all procedure parameters are an even multiple of four bytes long. Therefore, when removing parameter data from the stack the value you add to ESP must reflect the fact that the Intel ABI rounds parameter sizes up to the next multiple of four bytes.

The last parameter passing mechanism of immediate interest to us is the *Stdcall* (standard call) parameter passing mechanism. The Stdcall scheme is a combination of the C and Pascal calling sequences. Like the C calling sequence, the Stdcall scheme pushes the parameters on the stack in the opposite order of their declaration. Like the Pascal calling sequence, the procedure automatically removes the parameters from the stack before returning. Therefore, the caller does not have to remove the parameter data from the stack (thus improving efficiency by a small amount). Most of the Win32 API functions use the Stdcall calling convention. In HLA, you can use the *@stdcall* procedure attribute to specify the Stdcall calling convention, e.g.,

```
procedure stdProc( i:int32; j:int32; k:int32 ); @stdcall; @exter-
nal;
```

HLA's high level procedure call syntax will automatically push the parameters on the stack in the proper (i.e., reverse) order:

```
stdProc( a, 5, 2 );
```

Of course, you can also manually call a Stdcall procedure yourself. Be sure to push the parameters in the reverse order!

```
pushd( 2 );
pushd( 5 );
push( a );
call stdProc;
```

Notice that this code does not remove any parameters from the stack. That is the function's job.

Some older HLA code (written before the @stdcall facility was added to the language) simulates the Stdcall calling convention by reversing the parameters in the procedure declaration (indeed, some of the HLA standard library code takes this one step farther and uses macros to swap the parameters prior to making calls to these procedures). Such techniques are obsolete and you shouldn't employ them; however, since there is some code lying around that does this, you should be aware of why it does this.

3.3.4: Win32 Parameter Types

Almost all Win32 parameters are exactly four bytes long. This is true even if the formal parameter is one byte (e.g., a *char* object), two bytes (a *short int*), or some other type that is smaller than four bytes. This is done to satisfy the Intel 80x86 ABI and to keep the stack pointer aligned on a four-byte boundary. Since all parameters are exactly four bytes long, a good question to ask is "how do you pass smaller objects, or objects whose size is not an even multiple of four bytes, to a Win32 API function?" This section will briefly discuss this issue.

Whenever you pass a byte parameter to some function, you must pad that byte out to four bytes by pushing an extra three bytes onto the stack. Note that the procedure or function you call cannot assume that those bytes contain valid data (e.g., the procedure/function cannot assume those three bytes all contain zeros). It is perfectly reasonable to push garbage bytes for the upper three bytes of the parameter. HLA will automatically generate code that pushes a byte-sized actual parameter onto the stack as a four-byte object. Most of the time, this code is relatively efficient. Sometimes, however, HLA may generate slightly less efficient code in the interest of safety. For example, if you pass the BH register as a byte-sized parameter (a reasonable thing to do), there is no way that HLA can push BH onto the stack as a double word with a single instruction. Therefore, HLA will emit code like the following:

```
sub( 4, esp );    // make room for the parameter
mov( bh, [esp] ); // Save BH in the L.O. byte of the object on top of stack.
```

Notice that the upper three bytes of this double-word on the stack will contain garbage. This example, in particular, demonstrates why you can't assume the upper three bytes of the double word pushed on the stack contain zeros. In this particular case, they contain whatever happened to be in those three bytes prior to the execution of these two instructions.

Passing the AL, BL, CL, or DL register is fairly efficient on the 80x86. The CPU provides a single byte instruction that will push each of these eight-bite values onto the screen (by passing the entire 32-bit register that contains these registers:

```
push( eax );  // Passes al.
push( ebx );  // Passes bl.
push( ecx );  // Passes cl,
push( edx );  // Passes dl
```

Passing byte-sized memory objects is a bit more problematic. Your average assembly language programmer would probably write code like the following:

```
push( (type dword byteVar) );  // Pushes byteVar plus three following bytes
call funcWithByteParam;
```

HLA, because it observes safety at the expense of efficiency, will not generate this code. The problem is that there is a *tiny* chance that this will cause the system to fail. This situation could occur if *byteVar* is located in memory within the last three bytes of a page of memory (4096 bytes) and the next page in memory is not readable. That would raise a memory access violation. Quite frankly, the likelihood of this ever occurring is so remote that your average programmer would ignore the possibility of it ever happening. However, compilers cannot be so cavalier. Even if the chance that this problem will occur is remote, a compiler must generate safe code (that will never break). Therefore, HLA actually generates code like the following:

```
push( eax );            // Make room for parameter on stack.
push( eax );            // Preserve EAX's value
mov( byteVar, al );
mov( al, [esp+4] );     // Save byteVar's value into parameter location
pop( eax );             // Restore EAX's value.
call funcWithByteParam; // Call the function.
```

As you can see, the code that HLA generates to pass a byte-sized object as a parameter can be pretty ugly (note that this is only true when passing variables).

Part of the problem with generating code for less-than-dword-sized parameters is that HLA promises to never mess with register values when passing parameters[15]. HLA provides a special procedure attribute, *@use*, that lets you tell HLA that it is okay to modify the value of a 32-bit register if doing so will allow HLA to generate better code. For example, suppose *funcWithByteParam* had the following external declaration:

```
procedure funcWithByteParam( b:byte ); @use EAX; @external;
```

With this declaration, HLA can generate better code when calling the function since it can assume that it's okay to wipe out the value in EAX:

```
// funcWithByteParam( byteVar );

mov( byteVar, eax );
push( eax );
call funcWithByteParam;
```

Because the Intel ABI specifies that EAX (and ECX/EDX) are scratch registers and any function following the Intel ABI is free to modify their values, and because the Win32 functions follow the Intel ABI, and because most Win32 API functions return a function return result in EAX (thereby guaranteeing that they wipe out EAX's value on any Win32 API call), you might wonder why you (or the HLA Standard Library) shouldn't just always specify "@use EAX;" on every Win32 function declaration. Well, there is a slight problem with doing this. Consider the following function declaration and invocation:

15.Indeed, the only time HLA messes with any register value behind your back is when invoking a class method. However, HLA well-documents that fact that class method and procedure calls may wipe out the values in ESI and EDI.

```
procedure func( b:char; c:char; d:boolean ); @use eax; @external;
      .
      .
      .
func( charVar, al, boolVar );
```

Here's code similar to what HLA would generate for this function call:

```
mov( charVar, al );
push( eax );
push( eax );
mov( boolVar, al );
push( eax );
call func;
```

Do you see the problem? Passing the first parameter (when using the `@pascal` calling convention) wipes out the value this code passes as the second parameter in this function invocation. Had we specified the `@cdecl` or `@stdcall` calling convention, then passing the third parameter would have done the dirty deed. For safety reasons, the HLA Standard Library that declares all the Win32 API functions does not attach the `@use` procedure attribute to each procedure declaration. Therefore, certain calls to Win32 API routines (specifically, those that pass memory variables that are less than four bytes long as parameters) will generate exceedingly mediocre code. If having smaller programs[16] is one of your primary goals for writing Windows applications in assembly language, you may want to code calls containing such parameters manually.

If a parameter object is larger than four bytes, HLA will automatically round the size of that object up to the next multiple of four bytes in the parameter declaration. For example, `real80` objects only require ten bytes to represent, but when you pass one as a parameter, HLA sets aside 12 bytes in the stack frame. When HLA generates the code to pass a `real80` object to a procedure, it generates the same code it would use to pass two double word variables and a word variable; in other words, the code needed to pass the last two bytes could get ugly (for the same reasons we've just covered). However, since there aren't any Win32 API functions that expect a `real80` parameter, this shouldn't be an issue.

Table lists the typical C/C++ data types, their HLA equivalents, and how much space they consume when you pass them as parameters to a Win32 API function.

Table 3-11: Space Consumed by Various C Types When Passed as Parameters

C Type	Corresponding HLA Types	Space Consumed on Stack	Padding
char	char, byte, int8[a]	four bytes	three bytes
short	word, int16	four bytes	two bytes
int	dword, int32	four bytes	none
long	dword, int32	four bytes	none
long long	qword, int64	eight bytes	none

16. In the big picture, this extra code is not going to affect the running time of your code by a significant factor. Win32 API functions are sufficiently slow to begin with that the few extra clock cycles consumed by the "safe" code is insignificant.

C Type	Corresponding HLA Types	Space Consumed on Stack	Padding
unsigned char	char, byte, uns8	four bytes	none
unsigned short	word, uns16	four bytes	none
unsigned	dword, uns32	four bytes	none
unsigned int	dword, uns32	four bytes	none
unsigned long	dword, uns32	four bytes	none
unsigned long long	qword, uns64	eight bytes	none
float	real32	four bytes	none
double	real64	eight bytes	none
long double	real64 (on some compilers)	eight bytes	none
	real80 (on other compilers)	twelve bytes	two bytes

a.Some compilers have an option that lets you specify the use of unsigned char as the default. In this case, the corresponding HLA type is uns8.

3.3.5: Pass by Value Versus Pass by Reference

The C programming language only supports *pass by value* parameters. To simulate pass by reference parameters, C requires that you explicitly take the address of the object you wish to pass and then pass this address through a pass by value parameter that is some pointer type. The following code demonstrates how this is done:

```
/* C code that passes some parameter by reference via pointers */

int someFunc( int *ptr )
{
   *ptr = 0;
}
      .
      .
      .
/* Invocation of this function, assume i is an int */

   someFunc( &i );
```

This function passes the address of `i` as the value of the `ptr` parameter. Within `someFunc`, the function dereferences this pointer and stores a zero at the address passed in through the pointer variable (since, in this example, we've passed in the address of `i`, this code stores a zero into the `i` variable).

HLA, like the C++ language, directly supports both pass by value and pass by reference parameters[17]. So when coding a prototype for some Win32 API function that has a pointer parameter, you've got the choice of specifying a pointer type as a value parameter or the pointer's base type as a reference parameter in the HLA dec-

17.Actually, HLA supports several different parameter passing mechanisms. However, pass by value and pass by reference are the only ones that are of interest when calling Win32 API functions, so we'll discuss only those here. See the HLA reference manual for more details on the more advanced parameter passing mechanisms.

laration. The *someFunc* C function in the current example could be encoded with any of the following to HLA declarations:

```
   // A "typeless" variable declaration (since pointers are always 32-bit values)
   // that passes the parameter by value:

   procedure someFunc( ptr:dword );
   begin someFunc;

      mov( ptr, eax );
      mov( 0, (type dword [eax]) );

   end someFunc;



   // A typed version passing a pointer to an int32 object as a value parameter:

type
   pInt32 :pointer to int32;
      .
      .
      .
   procedure someFunc( ptr:pInt32 );
   begin someFunc;

      mov( ptr, eax );
      mov( 0, (type dword [eax]) );

   end someFunc;



   // A version using pass by reference parameters

   procedure someFunc( var ptr:int32 );
   begin someFunc;

      mov( ptr, eax );
      mov( 0, (type dword [eax]) );

   end someFunc;
```

Note that the function's body is exactly the same in all three cases. The function has to grab the address passed on the stack and store a zero at that memory address (just as the C code does). If you manually call *someFunc* (that is, if you use low-level assembly syntax rather than HLA's high-level procedure calling syntax), then the code you write to call any of these three versions is also identical. It is

```
// someFunc( i );

   lea( eax, i );   // Take the address of i, could use "pushd( &i );" if i is static.
   push( eax );     // This code assumes that it is okay to wipe out EAX's value.
   call someFunc;   // We're also assuming @pascal or @stdcall convention here.
```

The difference between these procedure declarations is only evident when you use HLA's high level procedure calling syntax. When the parameter is a double word or pointer value, the caller must explicitly write the code to calculate the address of the actual parameter and pass this computed address as the parameter's value, as the following example demonstrates:

```
// procedure someFunc( ptr:dword );
// -or-
// procedure someFunc( ptr:pInt32 );
//
// call someFunc, passing the address of "i":

   lea( eax, i );
   someFunc( eax );
```

When calling a procedure that has a pass by reference parameter, all you need do is pass the variable itself. HLA will automatically generate the code that takes the address of the variable:

```
// procedure someFunc( var ptr:int32 );

   someFunc( i );
```

If i is a `static, storage,` or `readonly` variable without any indexing applied to it, then HLA generates the following code for this statement:

```
   push( &i );
   call someFunc;
```

However, if the actual parameter (i in this case) is an indexed static object, or is a local variable, then HLA will have to generate code like the following:

```
   push( eax );
   push( eax );
   lea( eax, i );
   mov( eax, [esp+4] );
   pop( eax );
   call someFunc;
```

This happens because HLA promises not to mess with register values when passing parameters. Of course, you can improve the quality of the code that HLA generates by using the "@use" procedure attribute, remembering the caveats given earlier:

```
// procedure someFunc( var i:int32 ); @use EAX;

   someFunc( i );  // Assume i is a local (automatic) variable

// is equivalent to

   lea( eax, i );
   push( eax );
   call someFunc;
```

HLA's pass by reference parameter passing mechanism requires that you specify a memory address as an actual reference parameter. So what happens if you run into a situation when the address you want to pass is in a register and you've specified a pass by reference parameter? If you try to call the function with code like the following HLA will complain that you've not specified a valid memory address:

```
someFunc( esi );
```

The trick is to give HLA what it wants: a memory address. This is easily achieved by specifying the following function call to `someFunc`:

```
someFunc( [esi] );
```

This generates the following assembly code:

```
push( esi );
call someFunc;
```

HLA usually requires the type of the actual parameter (the parameter you pass in a procedure call) to *exactly* match the type of the formal parameter (the parameter appearing in the declaration of the procedure). You cannot pass the address of a `char` variable as a parameter when the original function calls for a `boolean` variable (even though both parameter types are one byte in length). There are a couple of exceptions worth noting. You may pass a `byte` variable as an actual parameter whenever the formal parameter is one byte in length. Similarly, HLA will allow an actual parameter whose type is `word` if the formal parameter's size is two bytes and HLA will allow an actual `dword` parameter whenever the formal parameter is four bytes. Also, if the formal parameter is a `byte, word,` or `dword` type, then HLA will allow you to pass an actual parameter that is one byte long, two bytes long, or four bytes long, respectively. HLA will also allow an anonymous memory object (e.g., "[eax]") as an actual parameter for any pass by reference parameter; such a parameter will simply pass the value of the specified register as the address for the reference parameter.

One feature that HLA supports as a convenience (especially for Win32 API function calls) is that if you pass a pointer variable as an actual pass by reference parameter, where the formal type of the reference parameter is the base type of the pointer, HLA will go ahead and pass the value of the pointer rather than returning an error (or passing the address of the pointer variable), e.g., the following demonstrates this:

```
type
   pi  :pointer to int32;
      .
      .
      .
   procedure hasRefParm( var i:int32 );
   begin hasRefParm;
      .
      .
      .
   end hasRefParm;

static
   myInt :int32;
   pInt  :pi;
      .
      .
      .
```

```
    hasRefParm( myInt );  // Computes and passes address of myInt.
        .
        .
        .
    hasRefParm( pInt );   // Passes the value of the pInt pointer variable
```

The choice of whether to pass a parameter as a pointer by value or as a variable by reference is mainly a matter of convenience. If you are usually passing an actual parameter that is a memory variable whose type matches the formal parameter's type, then pass by reference is probably the way to go. However, if you're doing pointer arithmetic or constantly passing the address of objects whose type doesn't exactly match the formal parameter's type (and you're sure you know what you're doing when you do this), then passing a pointer by value is probably going to be more convenient.

Many Win32 API functions accept the address of some buffer as a parameter. Often, the prototype for the function specifies the pointer type as "void *". This means that the caller is supplying the address of a block of memory and the abstract type attached to that block of memory is irrelevant to the compiler. HLA also provides a special form of the pass by reference parameter passing mechanism that suspends type checking on the actual parameters you pass to the procedure. Consider the following HLA procedure prototype:

```
procedure untypedVarParm( var parm:var ); @external;
```

Specifying "var" as the parameter's type tells HLA that this is an untyped pass by reference parameter. The caller can supply any memory address as the parameter and HLA will pass that address on to the function. Like normal pass by reference parameters, the actual parameter you supply to this function must be a memory location, you cannot supply a constant or a register as an operand (though you can specify "[reg_{32}]" as a parameter and HLA will pass the value of the specified 32-bit general purpose register as the memory address). This special pass by reference form is especially useful when passing Win32 API functions the address of some buffer where it can place data that Windows returns to the caller. There is, however, one big "gotcha" associated with untyped pass by reference parameters: HLA always passes the address of the variable you pass the function. This is true even if the variable you pass as a parameter is a pointer variable. The following is syntactically acceptable to HLA, but probably doesn't do what the programmer expects:

```
    procedure hasUntypedParm( var i:var );
    begin hasUntypedParm;
        .
        .
        .
    end hasUntypedParm;

static
    myInt :int32;
    pInt  :pi;
        .
        .
        .
    hasUntypedParm( myInt );  // Computes and passes address of myInt.
        .
        .
        .
    hasUntypedParm( pInt );   // Computes and passes address of pInt
```

In particular, note that this code does not pass the value of the pointer variable in the second call. Instead, it takes the address of the pointer variable and passes that address on to the *hasUntypedParm* procedure. So take care

when choosing to use untyped pass by reference parameters; their behavior is slightly different than regular pass by reference parameters as this example shows.

There is one important issue that HLA programmers often forget: *HLA string variables are pointers!* Most Win32 API functions that return data via a pass by reference parameter return character (string) data. It's tempting to be lazy and just declare all pass by reference parameters as untyped parameters. However, this can create havoc when calling certain Win32 API functions that return string data. Consider the following Win32 API procedure prototype:

```
static
   GetFullPathName: procedure
   (
          lpFileName    : string;
          nBufferLength : dword;
      var lpBuffer      : var;
      var lpFilePart    : var
   );
   @stdcall; @returns( "eax" ); @external( "__imp__GetFullPathNameA@16" );
```

This function stores a zero-terminated string into the block of memory pointed at by `lpBuffer`. It might be tempting to call this procedure as follows:

```
static
   s  :string;
   fp :pointer to char;
      .
      .
      .
   stralloc( 256 );
   mov( eax, s );
      .
      .
      .
   GetFullPathName( "myfile.data", 256, s, fp );
   mov( s, ebx );                                  // GetFullPathName returns the actual
   mov( eax, (type str.strRec [ebx]).length );     // string length in EAX.
```

The objective of this code is (obviously) to have the call to `GetFullPathName` place the full path name of the *myfile.data* file into the string variable `s`. Unfortunately, this code does not work as advertised. The problem is that the `lpBuffer` variable is an untyped reference parameter. As a result, the call to `GetFullPathName` takes the address of whatever variable you pass it, even if that variable is a pointer variable. Since strings are four-byte pointers (that contain the address of the actual character data), this example code doesn't do what the author probably intended. Rather than passing the address of the character string data buffer as you might expect, this code passes the address of the four-byte pointer variable `s` to `GetFullPathName` as the buffer address. On return, this function will have overwritten the pointer value (and probably the values of other variables appearing in memory immediately after `s`). Notice how the original example of this function call appearing earlier in the chapter handled this situation:

```
static
   fullName :string;
   namePtr  :pointer to char;
      .
      .
      .
```

```
stralloc( 256 );       // Allocate sufficient storage to hold the string data.
mov( eax, fullName );
   .
   .
   .
mov( fullName, edx );  // Get the address of the data buffer into EDX!
GetFullPathName
(
   "myfile.exe",                          // File to get the full path for.
   (type str.strRec [edx]).MaxStrLen,     // Maximum string size
   [edx],                                 // Pointer to buffer
   namePtr                                // Address of base name gets stored here
);
mov( fullName, edx );                     // Note: Win32 calls don't preserve EDX
mov( eax, (type str.strRec [edx]).length  // Set the actual string length
```

We'll return to this issue a little later in this chapter when we discuss the conversion of Win32 API function prototypes from C to HLA.

The C language always passes arrays by reference. Whenever the C language sees the name of a function without an index operator ("[...]") attached to it, C substitutes the address of the first element of the array for that array. Similarly, if you specify some array as a formal parameter in a C function declaration, C assumes that you will actually be passing in a pointer to an element of that array type[18].

Structures, on the other hand, C always passes by value (unless, of course, you explicitly take the address of a `struct` object using the address-of operator and pass that pointer to the `struct` as your parameter value). Win32 API functions always pass pointers to structures (that is, they expect you to pass structures by reference rather than by value), so when you create a prototype for a Win32 API function call that has a `struct` as a parameter, you'll always specify a pointer to the structure or pass it by reference in the HLA declaration.

3.4: Calling Win32 API Functions

The Windows operating system consists of several dynamic linked library (DLL) modules in memory. Therefore, when you call a Win32 API function, you're not actually calling that function directly. Indeed, unless you declare your function in a special way, there may be two levels of indirection involved before you get to the actual Win32 kernel code within the DLL. This section will give you a quick primer on Win32 DLLs and how to design your Win32 API function prototypes in HLA to make them slightly more efficient.

The phrase "dynamic linked library" means that linkage to a library module is done at run-time. That is, supplying the run-time address of the library function in question could, technically, be done after your program begins execution. The linking process normally involves patching the address fields of all the call instructions that reference a given function in some library code being linked. However, at run-time (that is, after Windows has loaded your program into memory and begun its execution), it's impractical to attempt to locate every call to some function so that you can modify the address field to point at the new location of that function in memory. The solution to this problem is to provide a single object that has to be changed in memory to provide the linkage, put that object in a known location, and then update that single object whenever dynamically linking in the the function. By having a "single point of contact" the OS can easily change the address of that contact object.

There are two ways to add such a "single point of contact" to a machine code program. The first way is to use a pointer that holds the address of the ultimate routine to call. The application code, when it wants to invoke the

18.As noted earlier, C does not differentiate pointer or array access syntax. Both are identical to C, for the most part. This is how C gets away with passing all arrays as a pointer to their first element.

Win32 API function (or any other function in some DLL) would simply issue an indirect call through this pointer. The second way is to place a *jmp* instruction at a known location and modify the *jmp* instruction's address operand field to point at the first instruction of the desired function within the DLL. The indirect call mechanism is a little more efficient, but it requires encoding a special form of the call instruction whenever you call a function in the DLL (and many compilers will not generate this special form of the *call* instruction by default, if they provide the option to do it at all). The use of the *jmp* instruction mechanism is more compatible with existing software development tools, but this scheme requires the execution of an extra instruction in order to transfer control to the actual function in the DLL (specifically, you have to execute the *jmp* instruction after "calling" the function). Windows, as it turns out, combines both of these mechanisms when providing an interface to the Win32 API functions. The API interface consists of an indirect *jmp* instruction that transfers control to some location specified by a double-word pointer. The linking code can use any form of the *call* (or other control transfer) instruction to transfer control to the indirect *jmp* instruction. Then the indirect *jmp* transfers control to the actual function specified by the pointer variable. The operating system can dynamically change the target address of the function within the DLL by simply changing this pointer value in memory.

Of course, there is an efficiency drawback to this scheme. Not only must the code execute that extra *jmp* instruction, but an indirect *jmp* is typically slower than a direct *jmp*. So Windows' solution is the slowest of the three forms: you pay the price for the extra *jmp* instruction and the extra cost associated with the use of indirection. Fortunately, one of the advantages of assembly language is that you can easily circumvent this extra work.

Consider the following HLA procedure prototype to the Win32 *ExitProcess* function:

```
procedure ExitProcess( uExitCode:uns32 ); @stdcall; @external( "_ExitProcess@4" );
```

The *_ExitProcess@4* label is actually the label of an indirect *jmp* instruction that will be linked in with any program that calls *ExitProcess*. In HLA form, the code at the address specified by the *_ExitProcess@4* label looks something like the following (assuming labels like "_ExitProcess@4" were legal in HLA):

```
_ExitProcess@4: jmp( _imp__ExitProcess@4 );
```

The "__imp__ExitProcess@4" symbol is the name of a double word pointer variable that will contain the address of the actual *ExitProcess* function with the Win32 OS kernel, i.e.,

```
static
   _imp__ExitProcess@4 :dword;  // Assuming "@" was actually legal within an HLA ID.
```

Note that the library files (e.g., *kernel32.lib*) that you link your programs with contain definitions for both the symbols *_ExitProcess@4* and *_imp__ExitProcess@4*. The "standard" symbols (e.g., *_ExitProcess@4*) refer to the indirect *jmp* instruction. The symbols with the "_imp_" prefix refer to the double word pointer variable that will ultimately hold the address of the actual kernel code. Therefore, you can circumvent the execution of the extra *jmp* instruction by calling the kernel function indirectly through this pointer yourself, e.g.,

```
   call( _imp__ExitProcess@4 );   // Assuming "@" was actually legal within an HLA ID.
```

The major problem with this approach is that it doesn't allow the use of the HLA high level function call syntax. You would be forced to manually push any parameter(s) on the stack yourself when using this scheme. In a moment, you'll soon see how to circumvent this problem. Another minor issue is that HLA doesn't allow the "@" symbol in an identifier (as all the previous code fragments have noted). This, too, is easily corrected.

HLA allows you to declare both external procedures and variables. We'll use that fact to allow external linkage to both the jmp instruction (that is effectively the Win32 API function's entry point) and the pointer to the variable. The following two declarations demonstrate how you can do this:

```
static
   _imp__ExitProcess :dword; @external( "_imp__ExitProcess@4" );

procedure ExitProcess( uExitCode:uns32 ); @stdcall; @external( "_ExitProcess@4" );
```

HLA also allows the declaration of procedure variables. A procedure variable is a four-byte pointer to a given procedure. HLA procedure variables are perfect for Win32 API declarations because they allow you to use HLA's high level syntax for procedure calls while making an indirect call through a pointer. Consider the following declaration:

```
static
   ExitProcess :procedure( uExitCode:uns32 );
                @stdcall; @external( "_imp__ExitProcess@4" );
```

With this declaration, you can call `ExitProcess` as follows:

```
ExitProcess( 0 );
```

Rather than calling the code beginning with the indirect `jmp` instruction, this HLA high level procedure call does an indirect call through the `_imp__ExitProcess@4` pointer. Since this is both convenient and efficient, this is the scheme this book will generally employ for all Win32 API function calls.

3.5: Win32 API Functions and Unicode Data

Before discussing how to create HLA procedure prototypes for all the Win32 API functions, a short digression is necessary in order to understand certain naming conventions in the Win32 API. For many years, there were actually two different Win32 OS families: the Windows 95/98/2000ME family and the Windows NT/2000/XP family. The branch starting with Windows 95 was based on Windows 3.1 and MS-DOS to a large extent. The OS branch that started with NT was written mostly from scratch without concern about legacy (DOS) compatibility. As such, the internal structure of these two operating system families was quite different. One area where the difference is remarkable is with respect to character data. The Windows 95 family uses standard eight-bit ANSI (ASCII) characters internally while the NT branch uses Unicode internally. Unicode, if you're unfamiliar with it, uses 16-bit character codes allowing the use of up to 65,536 different characters. The beauty of Unicode is that you can represent most character symbols in use by various languages with a single character code (unlike ASCII, which only supports 128 different character values and isn't even really adequate for English, much less English plus dozens of other languages). Since Microsoft was interested in producing an international operating system, Unicode seemed like the right way to go.

Unicode has some great advantages when it comes to write applications that work regardless of the natural language of the application's users. However, Unicode also has some serious disadvantages that prevent it from immediately taking over the world:

- Few software tools directly support Unicode, so it is difficult to develop Unicode-enabled applications (though this is changing as time passes).
- Unicode data requires twice as much storage as ANSI data. This has the effect of doubling the size of many databases and other applications that manipulate a considerable amount of character data.

- Because Unicode characters are twice as long as ANSI characters, processing Unicode data typically takes twice as long as processing ANSI/ASCII characters (a serious defect to most assembly language programmers).
- Many programs that manipulate character data use look-up tables and bit maps (character sets) to operate on that data. An ASCII-based look-up table requires 128 bytes, an ANSI look-up table typically requires 256 bytes, a Unicode-based look-up table would require 65,536 bytes (making it impractical to use a look-up table for all but the most specialized of cases when using Unicode). Even implementing a character set using a power set (i.e., a bit map) would require 8,192 bytes; still too large for most practical purposes.
- There are nowhere near as many Unicode-based string library functions available as there are for ASCII/ANSI based strings. For example, the HLA Standard Library provides almost no Unicode-based string functions at all (actually, it provides none, but a few routines will work with Unicode-based strings).
- Another problem with using Unicode is that HLA v1.x provides only basic support for Unicode characters[19]. At the time this was being written, HLA supported the declaration of *wchar* and *wstring* constants and variables as well as Unicode character and string literal constants (of the form u'A' and u"AAA"). You could also initialize wchar and wstring static objects as the following example demonstrates. However, HLA constant expression parser does not (as of this writing) support Unicode string operations nor does the HLA Standard Library provide much in the way of Unicode support. The following is an example of static initialization of Unicode data (see the HLA reference manual for more details):

```
static
   wCharVar   :wchar := u'w';
   wStringVar :wstring := u"Unicode String";
```

For all these reasons, and many more, Microsoft realized (while designing Windows NT) that they couldn't expect everyone to switch completely over to Unicode when writing applications for Windows NT (or when using applications written for Windows NT). Therefore, Microsoft's engineers provided duomorphic[20] interfaces to Windows NT that involve character data: one routine accepts and returns ANSI data, another routine accepts and returns Unicode data. Internally, of course, Windows NT doesn't really have two sets of routines. Instead, the ANSI routines simply convert incoming data from ANSI to Unicode and the outgoing data from Unicode to ANSI.

In Microsoft Visual C++ (and other high level languages) there is a little bit of macro trickery used to hide the fact that the application has to choose between the Unicode-enabled and the ANSI versions of the Win32 API function calls. By simply changing one macro definition in a properly-written C++ program, it's possible to switch from ANSI to Unicode or from Unicode to ANSI with no other changes to the program. While the same trick is theoretically possible in assembly language (at least, in HLA), the dearth of a good set of Unicode library functions reduces this to the status of an interesting, but not very useful, trick. Therefore, this book will concentrate on producing ANSI-compatible applications with a small discussion of how to do Unicode applications when doing so becomes more practical in assembly language.

Windows duomorphic interface only applies to those functions that accept or return character data. A good example of such a routine is the Win32 API *DeleteFile* function that has the following two interfaces:

```
procedure DeleteFile( lpFileName :string ); @stdcall; @external( "_DeleteFileA@4" );
```

19.Least you use this as an argument against using HLA, note that HLA actually provides *some* Unicode support. Most assemblers provide no Unicode support whatsoever at all.

20.Two-faced.

```
// -or-

   procedure DeleteFile( lpFileName :wstring ); @stdcall; @external( "_DeleteFileW@4" );
```

If you look closely at these two declarations, you'll notice that the only difference between the two is a single character appearing in the external name and the type of the parameter. One of the external names has an "A" (for ANSI) immediately before the "@" while the other has a "W" (for Wide) immediately before the "@" character in the name. Wide, in this context, means a two-byte character format; so the name with the embedded "W" is the Unicode version of the function's name.

The presence of the "A" or the "W" at the end of the function's name in the external declaration (i.e., just before the "@", whose purpose the next section covers) determines whether the function is the ANSI version or the Unicode version ("A"=ANSI, "W"=Unicode). There is only one catch: when reading C/C++ documentation about the Windows API, you'll generally see the function referred to as "DeleteFile" (or whatever), not "DeleteFileA" or "DeleteFileW". So how can you tell whether the function's external name requires the "A" or "W"? Well, if any of the parameters involves character or string data, it's a given that the function will have ANSI and Unicode counterparts. If you're still not sure, you can always run Microsoft's *dumpbin.exe* utility on one of the Win32 API interface libraries (e.g., *kernel32.lib, gdi32.lib, user32.lib*, etc.) to extract all the exported names:

```
dumpbin /exports kernel32.lib
```

This command lists all the Win32 API function names that the *kernel32.lib* library module exports. If you save the output of this command to a text file (by using I/O redirection) you can search for a particular function name with nearly any text editor. Once you find the filename, if there is an "A" or "W" at the end of the name, you know that you've got a duomorphic function that deals with ANSI or Unicode characters. If no such character appears, then the function only works with binary (non-character) data.

Please note that the official name for a Win32 API function does not include the "A" or "W" suffix. That is, the Win32 documentation refers only to names like `DeleteFile`, never to names like `DeleteFileA` or `DeleteFileW`. The assumption is that an application is only going to use one of the two different character types. Either all character data is ANSI, in which case the application will call those functions with the "A" suffix, or all character data is in Unicode form and the application will call those functions with the "W" suffix. Although it's easy enough to switch between the two in an assembly language program, it's probably a good idea to stick to one form or another in a given application (less maintenance issues that way). The examples in this book will all use the ANSI forms of these functions, since assembly language better supports eight-bit character codes.

This book will also adopt the Win32 convention of specifying the API function names without the "A" or "W" suffix. That is, we'll call functions like `DeleteFile` and `GetFullPathName` and not worry about whether it's ANSI or Unicode on each call. The choice will be handled in the declaration of the prototype for the particular Win32 API function. This makes it easy (well, easier) to change from one character format to another should the need arise in the future.

For the most part, this book will stick to the ANSI character set because HLA provides much better support for that character set. If you need to use Unicode in your programs, you'll need to adjust the Win32 API prototypes and HLA `char/string` declarations accordingly.

Note that the names that have the "A" and "W" suffixes are really external names only. C/C++ documentation doesn't mention these suffixes. Again, if you're unsure whether the suffix is necessary, run the *dumpbin* program to get the actual library name.

3.6: Win32 API Functions and the Parameter Byte Count

As you've seen in a few examples appearing in this chapter, the external Win32 API function names typically have an at-sign ("@") and a number appended to the end of the function's external name. This numeric value specifies the number of bytes passed as parameters to the functions. Since most parameters are exactly four bytes long, this number (divided by four) usually tells you how many parameters the API function requires (note that a few API calls have some eight-byte parameters, so this number isn't always an exact indication of the number of parameters, but it does hold true the vast majority of the time).

Note that the names that have the "@nn" suffix are really external names only. C/C++ documentation doesn't mention this suffix. Furthermore, since HLA doesn't allow you to embed at signs ("@") into identifiers, you cannot use these external names as HLA identifiers. Fortunately, HLA's `@external` directive allows you to specify any arbitrary string as the external symbol name.

This book will also adopt the Win32 convention of specifying the API function names without the "@nn" suffix. That is, we'll call functions like `DeleteFile` and `GetFullPathName` and not worry about tacking on the number of bytes of parameters to the name. The full name will be handled in the external prototype declaration for the particular Win32 API function. If you need to determine the exact constant for use in an external declaration, you can run the Microsoft *dumpbin* program on the appropriate .LIB file to determine the actual suffix.

3.7: Creating HLA Procedure Prototypes for Win32 API Functions

Although the HLA distribution includes header files that provide prototypes for most of the Win32 API functions (see the next section for details), there are still some very good reasons why you should be able to create your own HLA external declarations for a Win32 function. Here is a partial list of those reasons:

- HLA provides most, but not all, of the Win32 API Prototypes (e.g., as Microsoft adds new API calls to Windows, HLA's header files may become out of date).
- Not every HLA prototype has been thoroughly tested (there are over 1,500 Win32 API function calls one could make and some of those are quite esoteric). There very well could be a defect in the prototype of some function that you want to call.
- The choice of data type for a give API function may not exactly match what you want to use (e.g., it could specify an `uns32` type when you'd prefer the more general `dword` type).
- You may disagree with the choice of passing a parameter by reference versus passing a pointer by value.
- You may disagree with the choice of an untyped reference parameter versus a typed reference parameter.
- You may disagree with the choice of an HLA string type versus a character buffer.

There are certainly some other reasons for abandoning HLA's external prototypes for various Win32 API functions. Whatever the reason, being able to create an HLA prototype for these functions based on documentation that provides a C prototype is a skill you will need. The following subsections condense the information appearing in the previous sections, toss out a few new ideas, and discuss the "hows and whys" of Win32 API prototyping in HLA.

3.7.1: C/C++ Naming Conventions Versus HLA Naming Conventions

Before jumping in and describing how to translate C/C++ prototypes into HLA format, a slight digression is necessary. Sometimes, you'll run into a minor problem when translating C/C++ code to HLA: identifiers in a C/

C++ program don't always map to legal identifiers in an HLA program. Another area of contention has to do with the fact that Microsoft's programmers have created many user-defined types that the Windows system uses. More often than not, these type names are isomorphisms (that is, a different name for the same thing; for example, Microsoft defines dozens, if not hundreds, of synonyms for *dword*). However, if you understand Microsoft's naming conventions, then figuring out what HLA types to substitute for all these Microsoft names won't prove too difficult.

HLA and C/C++ use roughly the same syntax for identifiers: identifiers may begin with an alphabetic (uppercase or lowercase) character or an underscore, and zero or more alphanumeric or underscore characters may follow that initial character. Given that fact, you'd think that converting C/C++ identifiers to HLA would be fairly trivial (and most of the time, it is). There are, however, two issues that prevent the translation from being completely trivial: HLA reserved words and case sensitivity. We'll discuss these issues shortly.

Even when a C/C++ identifier maps directly to a legal HLA identifiers, questions about that identifier , its readability, applicability, etc., may arise. Unfortunately, C/C++ naming conventions that have been created over the years tend to be rather bad conventions (remember, C was create circa-1970, back in the early days of "software engineering" before people really studied what made one program more readable than another). Unfortunately, there is a lot of inertia behind these bad programming conventions. Someone who is not intimately familiar with those conventions may question why a book such as this one (which covers a different language than C/C++) would continue to propagate such bad programming style. The reason is practical: as this chapter continues to stress, there is a tremendous amount of documentation written about the Win32 API that is C-based. While there is an aesthetic benefit to renaming all the poorly-named identifiers that have become standards in C/C++ Windows source files, doing so almost eliminates the ability to refer to non-HLA based documentation on the Win32 API. That would be a much greater loss than having to deal with some poorly named identifiers. For that reason alone, this book attempts to use standard Windows identifiers (which tend to follow various C/C++ naming conventions) whenever referring to those objects represented by the original Windows identifiers. Changes to the Windows naming scheme are only made where absolutely necessary. However, this book will only use the Windows naming conventions for pre-existing, reknown, Windows (and C/C++) identifiers. This book will adopt the standard "HLA naming convention" (given a little later) for new identifiers.

One problem with C/C++ naming conventions is that they are inconsistent. This is because there isn't a single C/C++ naming convention, but several that have sprung up over the years. Some of them contain mutually exclusive elements, still it isn't unusual to see several of the conventions employed within the same source file. Since the main thrust of this chapter is to prepare you to read Win32 API documentation, the sections that follow will concentrate on those conventions and problems you'll find in typical Windows documentation.

3.7.1.1: Reserved Word Conflicts

The HLA language defines hundreds of reserved words (this is reasonable, since there are hundreds of machine instructions in the 80x86 instruction set, though there is no arguing against that fact that HLA has a large number of reserved words above and beyond the machine instructions). Since not all of HLA's reserved words are reserved words in C/C++, it stands to reason that there are some programs out there than inadvertently use HLA reserved words as identifiers in their source code. This fact is just as true for the Win32 API definitions appearing in Microsoft's C/C++ header files as it is for application programs. There will be some C/C++ identifiers in the Win32 C/C++ documentation that we will not be able to use simply because they are HLA reserved words. Fortunately, such occurrences are rare. This book will deal with such issues on a case-by-case basis, providing a similar name that is not an HLA reserved word when such a conflict arises.

3.7.1.2: Alphabetic Case Conflicts

Another point of conflict between HLA identifiers and C/C++ identifiers is the fact that C/C++ is a *case sensitive* language whereas HLA is a *case neutral* language. HLA treats upper and lower case characters as distinct, but will not allow you to create an identifier that is the same as a different identifier except for alphabetic case. C/C++, on the other hand, will gladly let you create two different identifiers whose only difference is the case of alphabetic characters within the symbols. Worse, some C/C++ programmers have convinced themselves that it's actually a good idea to take advantage of this "feature" in the language (hint: it's a *terrible* idea to do this, it make programs harder to read and understand). Regardless of your beliefs of the utility of this particular programming style, the fact remains that C/C++ allows this (and some programmers take advantage of it) while HLA does not. The question is "which identifier do we modify and how do we modify it?"

Most of the time there is a case neutrality violation in a C/C++ program (that is, two identifiers are the same except for alphabetic case), it's usually the situation where one of the identifiers is either a type definition or a constant definition (the other identifier is usually a function or variable name). This isn't true all the time, but it is true in the majority of the cases where this conflict occurs. When such a conflict occurs, this book will use the following convention (prioritized from first to last):

- If one of the conflicting identifiers is a type name, we'll convert the name to all lowercase characters and append "_t" to the name (a common Unix convention).
- If one of the conflicting identifiers is a constant (and the other is not a type), we'll convert the name to all lowercase and append "_c" to the name (an HLA convention, based on the Unix convention).
- If neither of the above conditions hold, we'll give one of the identifiers a more descriptive name based on what the identifier represents/contains/specifies rather than on its classification (e.g., type of symbol).

A good convention to follow with respect to naming identifiers is the "telephone test." If you can read a line of source code over the telephone and have the listener understand what you're saying without explicitly spelling out an identifier, then that identifier is probably a decent identifier. However, if you have to spell out the identifier (especially when using phrases like "upper case" and "lower case" when spelling out the name), then you should consider using a better name. HLA, of course, prevents abusing and misusing alphabetic case in identifiers (being a case neutral language), so it doesn't even allow one to create identifiers that violate the telephone test (at least, from an alphabetic case perspective).

3.7.1.3: Common C/C++ Naming Conventions

If you search on the Internet for "C Naming Conventions" you'll find hundreds of pages extolling the benefits of that particular web page author's favorite C naming scheme. It seems like nearly every C programmer with an opinion and a web page is willing to tell the world how identifiers should be designed in C. The really funny thing is that almost every one of these pages that specifies some naming convention is mutually exclusive with every other such scheme. That is, if you follow the rules for naming C identifiers found at one web site, you're invariably break one or more rules at nearly every other site that provides a C naming convention. So much for convention; so much for standards.

Interestingly enough, the one convention that nearly everybody agrees upon is also, perhaps, the *worst* naming convention ever invented for programming language identifiers. This is the convention of using all uppercase characters for identifiers that represent constant values. The reason everyone agrees on this one convention is fairly obvious to someone who has been programming in the C programming language for some time: this is one of the few naming conventions proposed by Kernighan and Ritchie in their original descriptive text *The C Pro-*

gramming Language. In *Programming in C: A Tutorial* by Brian W. Kernighan, Mr. Kernighan describes this choice thusly:

> *Good style typically makes the name in the #define upper case; this makes parameters more visible.*

This quote probably offers some insight into why Kernighan and Ritchie proposed the use of all uppercase for constants in the first place. One thing to keep in mind about this naming convention (using all upper case for #define symbols) was that it was developed in the very early 1970s. At the time, many mainframes and programming languages (e.g., FORTRAN) only worked with uppercase alphabetic characters. Therefore, programmers were used to seeing all uppercase alphabetic characters in a source file and lowercase was actually unusual (despite the fact that C was developed on an interactive system that supported lower case). In fact, Kernighan and Ritchie really got it backwards - if they'd wanted the parameters to stand out, they should have made them all uppercase and made the #define name lower case. Another interesting thing to note from this quote was that the all uppercase convention was specifically created for *macros*, not *manifest constants*. The "good style" Brian Kernighan was referring to was an attempt to differentiate the macro name from the macro parameters. Manifest constants (that is, the typical constants you create with a #define definition) don't have parameters, so there is little need to differentiate the name from the macro's parameter list (unless, of course, Mr. Kernighan was treating the remainder of the line as the "parameters" to the #define statement).

Psychologists have long known (long before computer programming languages became popular) that uppercase text is much harder to read than lower case text. Indeed, to a large extent, the whole purpose of uppercase alphabetic text is to slow the reader down and make them take notice of something. All uppercase text makes material harder to read, pure and simple. Don't believe this? Try reading the following:

> PSYCHOLOGISTS HAVE LONG KNOWN (LONG BEFORE COMPUTER PROGRAMING LANGUAGES BECAME POPULAR) THAT UPPERCASE TEXT IS MUCH HARDER TO READ THAN LOWER CASE TEXT. INDEED, TO A LARGE EXTENT, THE WHOLE PURPOSE OF UPPERCASE ALPHBETIC TEXT IS TO SLOW THE READER DOWN AND MAKE THEM TAKE NOTICE OF SOMETHING. ALL UPPERCASE TEXT MAKES MATERIAL HARDER TO READ, PURE AND SIMPLE. DON'T BELEVE THIS? TRY REREADING THIS PARAGRPH.

There are four intentional spelling mistakes in the previous paragraph. Did you spot them all the first time you read this paragraph? They would have been much more obvious had the text been all lowercase rather than all uppercase. Reading all upper case text is so difficult, that most readers (when faced with reading a lot of it) tend to "short-circuit" their reading and automatically fill in words once they've read enough characters to convince them they've recognized the word. That's one of the reasons it's so hard to immediately spot the spelling mistakes in the previous paragraph. Identifiers that cause a lack of attention to the details are obviously problematic in a programming language and they're not something you want to see in source code. A C proponent might argue that this isn't really much of a problem because you don't see as much uppercase text crammed together as you do in the paragraph above. However, some long identifiers can still be quite hard to read in all upper case; consider the following identifier taken from the C/C++ *windows.inc* header file set: CAL_SABBREVMONTHNAME1. Quick, what does it mean?

Sometimes it is useful to make the reader slow down when reading some section of text (be it natural language or a synthetic language like C/C++). However, it's hard to argue that every occurrence of a constant in a source file should cause the reader to slow down and expend extra mental effort just to read the name (thus, hopefully, determining the purpose of the identifier). The fact that it is a constant (or even a macro) is far more easily conveyed using some other convention (e.g., the "_c" convention that this book will adopt).

Now some might argue that making all macro identifiers in a program stand out is a good thing. After all, C's macro preprocessor is not very good and it's macro expander produces some unusual (non-function-like) seman-

tics in certain cases. By forcing the programmer to type and read an all-uppercase identifier, the macro's designer is making them note that this is not just another function call and that it has special semantics. An argument like this is valid for macros (though a suffix like "_m" is probably a better way to do this than by using all uppercase characters in the identifier), but is completely meaningless for simple manifest constants that don't provide any macro parameter expansion. All in all, using all uppercase characters for identifiers in a program is a bad thing and you should avoid it if at all possible.

This text, of course, will continue to use all uppercase names for well-known constants defined in Microsoft's C/C++ header files. The reason is quite simple: they are documented in dozens and dozens of Windows programming books and despite the fact that such identifiers are more difficult to read, changing them in this text would prove to be a disaster because the information appearing herein would not be compatible with most of the other books on Windows programming in C/C++[21]. For arbitrary constant identifiers (i.e., those that are not predefined in the C/C++ Windows header files), this book will generally adopt the "_c" convention for constants.

One C/C++ naming convention that is specified by the original language definition is that identifiers that begin and end with an underscore are reserved for use by the compiler. Therefore, C/C++ programmers should not use such identifiers. This shouldn't prove to be too onerous for HLA programmers because HLA imposes the same restriction (identifiers beginning and ending with an underscore are reserved for use by the compiler and the HLA Standard Library).

Beyond these two conventions that have existed since the very first operational C compilers, there is very little standardization among the various "C Naming Conventions" documents you'll find on the Internet. Many suggestions that one document makes are style violations another document expressedly forbids. So much for standardized conventions! The problem with these myriad of non-standardized "standards" is that unless you include the style guide in the comments of a source file, the guidelines you're following are more likely to confuse someone else reading the source file who is used to operating under a different set of style guidelines.

Perhaps one of the most confusing set of style guidelines people come up with for C/C++ programs is what to do about alphabetic case. Wise programmers using alphabetic case differences for formatting only. They never attach meaning to the case of alphabetic characters within an identifier. All upper case characters for constants is fairly easy to remember (because it is so well ingrained in just about every C/C++ style guide ever written), but how easy is it to remember that "variables must be written in mixed case starting with a lower case character" and "Names representing types must be in mixed case beginning with an uppercase character"? There are some so-called style guidelines that list a dozen different ways to use alphabetic case in an identifier to denote one thing or another. Who can remember all that? What happens when someone comes along and doesn't intimately know the rules? Fortunately, you see little of this nonsense in Windows header files.

As noted earlier in this document, a common (though not ubiquitous) Unix/C programming convention is to append the suffix "_t" to type identifiers. This is actually an excellent convention (since it emphasizes the classification of an identifier rather than its type, scope, or value). The drawback to this scheme is that you rarely see it used consistently even within the same source file. An even bigger drawback is that you almost never see this naming convention in use in Windows code (Windows code has a nasty habit of using all uppercase to denote type names, as well as constant, macro, and enum identifiers, thus eliminating almost any mnemonic value the use of all uppercase might provide; about the only thing you can say about an all-uppercase symbol in a Windows program is that it's probably not a variable or a function name). Once again, this book will use standard Windows identifiers when referencing those IDs, but will typically use the Unix convention of the "_t" suffix when creating new type names.

21.Note, however, that there is a precedent for changing the Win32 API identifiers around when programming in a language other than C/C++. Borland's documentation for Delphi, for example, changes the spelling of many Windows identifiers to something more reasonable (note, however, that Pascal is a case insensitive language and some changes were necessary for that reason alone).

Without question, the most common naming convention in use within C/C++ Windows applications is the use of *Hungarian Notation*. Hungarian notation uses special prefix symbols to denote the type of the identifier. Since Hungarian notation is so prevalent in Windows programs, it's worthwhile to spend some time covering it in detail...

3.7.1.4: Hungarian Notation

Hungarian notation is one of those "innovations" that has come out of Microsoft that lots of people love and lots of people hate. Originally developed by Charles Simonyi in a technical paper (searching on the Internet for "Hungarian Notation Microsoft Charles Simonyi" is bound to turn up a large number of copies of his paper (or links to it), Hungarian notation was adopted internally within Microsoft and the popularized outside Microsoft via the Windows include files and Charles Petzold's "Programming Windows" series of books (which push Hungarian notation). As a result of these events, plus the large number of programmers that "cut their teeth" at Microsoft and went on to work at companies elsewhere, Hungarian notation is very a popular naming convention and it's difficult to read a C/C++ Windows program without seeing lots of examples of this notation.

Hungarian notation is one of those conventions that everyone loves to hate. There are lots of good, technical, reasons for not using Hungarian notation. Even many proponents of Hungarian notation will admit that it has its problems. However, people don't use it simply because Microsoft pushes it. In spite of the problems with Hungarian notation, the information it provides is quite useful in large programs. Even if the convention wasn't that useful, we'd still need to explore it here because you have to understand it in order to read C/C++ code; still, because it is somewhat useful, this book will even adopt a subset of the Hungarian notation conventions on an "as useful" basis.

Hungarian notation is a naming convention that allows someone reading a program to quickly determine the type of a symbol (variable, function, constant, type, etc.) without having to look up the declaration for that symbol. Well, in theory that's the idea. To someone accustomed to Hungarian notation, the use of this convention can save some valuable time figuring out what someone else has done. The basic idea behind Hungarian notation is to add a concise prefix to every identifier that specifies the type of that identifier (actually, full Hungarian notation specifies more than that, but few programmers use the full form of Hungarian notation in their programs). In theory, Hungarian notation allows programmers to create their own type prefixes on an application by application basis. In practice, most people stick to the common-predefined type prefixes (tags) let it go at that.

An identifier that employs Hungarian notation usually takes the following generic form:

```
prefix tag qualifier baseIdentifier
```

Each of the components of the identifier are optional, subject of course, to the limitation that the identifier must contain something. Interestingly enough, the *baseIdentifier* component (the name you'd normally think of as the identifier) is itself optional. You'll often find Hungarian notation "identifiers" in Windows programs that consist only of a possible prefix, tag, and/or qualifier. This situation, in fact, is one of the common complaints about Hungarian notation - it encourages the use of meaningless identifiers in a source file. The *baseIdentifier* in Hungarian notation is the symbol you'd normally use if you weren't using Hungarian notation. For the sake of example, we'll use *Variable* in the examples that follow as our base identifier.

The *tag* component in the Hungarian notation is probably the most important item to consider. This item specifies the base type, or use, of the following symbol. Table 3-12 lists many of the basic tags that Windows pro-

grams commonly use; note that Hungarian notation does not limit a program to these particular types, the user is free to create their own tags.

Table 3-12: Basic Tag Values in Hungarian Notation

Tag	Description
f	Flag. This is a true/false boolean variable. Usually one byte in length. Zero represents false, anything else is true.
ch	Character. This is a one-byte character variable.
w	Word. Back in the days of 16-bit Windows systems (e.g., Windows 3.1), this tag meant a 16-bit word. However, as a perfect demonstration of one of the major problems with Hungarian notation, the use of this prefix became ambiguous when Win32 systems started appearing. Sometimes this tag means 16-bit short, sometimes it means a 32-bit value. This prefix doesn't provide much in the way of meaningful information in modern Windows systems.
b	Byte. Always a one-byte value.
l	Long. This is generally a long integer (32 bits).
dw	Double Word. Note that this is not necessarily the same thing as an "l" object. In theory, the usage of this term is as ambiguous as "w", though in 80x86 Windows source files this is almost always a 32-bit double word object.
u	Unsigned. Typically denotes an unsigned integer value (usually 32 bits). Sometimes you will see this symbol used as a prefix to one of the other integer types, e.g., uw is an unsigned word.
r	Real. Four-byte single precision real value.
d	Double. Eight-byte double precision real value.
bit	A single bit. Typically used with field names that are bit fields within some C struct.
bm	Bit map. A collection of bits (e.g., pixel values).
v	Void. An untyped object. Typically used only with the pointer prefix (see the discussion of prefixes). Untyped pointers are always 32 bit objects under Win32.
st	String. Object is a Pascal string with a length prefix.
sz	String, zero terminated. Object is a C/C++ zero terminated string object.

In Table 3-12 you see the basic type values commonly associated with symbols employing Hungarian notation. Table lists some modifier prefixes you may apply to these types (though there is no requirement that a tag

appear after one of the prefixes, a lone prefix followed by the base identifier is perfectly legal, though not as readable as an identifier consisting of a prefix, tag, and base identifier).

Table 3-13: Common Prefix Values in Hungarian Notation

Prefix	Description
p	Pointer to some type.
lp	Long pointer to some type. Today, this is a synonym for "p". Back in the days of 16-bit Windows system, an "lp" object was 32 bits and a "p" object was 16 bits. Today, both pointer types are identical and are 32 bits long. Although you'll see this prefixed used quite a bit in existing code and header files, you shouldn't use this prefix in new code.
hp	Huge pointer to some type. Yet another carry-over from 16-bit Windows days. Today, this is synonymous with lp and p. You shouldn't use this prefix in new code.
rg	Lookup table. Think of an index into an array as a function parameter, the function's result (i.e., the table entry) is the *range* of that function, hence the designation "rg". This one is not common in many Windows programs.
i	An index (e.g., into an array). Also commonly employed for `for` loop control variables.
c	A count. cch, for example, might be the number of characters in some array of characters.
n	A count. Same as c but more commonly used to avoid ambiguity with ch.
d	The difference between two instances of some type. For example, dX might be the difference between to x-coordinate values.
h	A Handle. Handles are used through Windows to maintain resources. Many Win32 API functions require or return a handle value. Handles are 32-bit objects under Win32.
v	A global variable. Many programmers use 'g' rather than 'v' to avoid confusion with the 'v' basic tag specification.
s	A static variable (local or global)
k	A `const` object.

Here are some examples of names you'll commonly see (e.g., from the Windows.hhf header file) that demonstrate the use of these identifiers:

```
char *lpszString;    // Pointer to zero-terminated string of characters.
int  *pchAnswer;     // Pointer to a single character holding an answer.
HANDLE hFile;        // Handle of a file.
```

Note all of these prefixes and tags are equally popular in Windows programs. For example, you'll rarely see the "k" prefix specification in many Windows source files (instead, the programmer will probably use the common C/C++ uppercase convention to denote an identifier). Also, many of the prefix/tag combinations are ambiguous. Fortunately, few people (including the Windows header files) push Hungarian notation to the limit. Usually, you'll just see a small subset of the possibilities in use and there is little ambiguity.

Another component of Hungarian notation, though you'll rarely see this used in real life, is a qualifier. More often that not, qualifiers (if they appear at all) appear in place of the base identifier name. Table lists some of the common qualifiers used in Hungarian notation.

Table 3-14: Common Qualifiers in Hungarian Notation

Qualifier	Description
First	The first item in a set, list, or array that the program is working with (this does not necessarily indicate element zero of an array). E.g., iFirstElement.
Last	The last item in a set, list, or array that the program has worked upon (this does not necessarily indicate the last element of an array or list). E.g., pchLastMember. Note that Last index objects are always valid members of the set, list, or array. E.g., array[iLastIndex] is always a valid array element.
Min	Denotes the minimum index into a set, list, or array. Similar to First, but First specifies the first element you're dealing with rather than the first object actually present.
Max	Denotes an upper limit (plus one, usually) into an array or list.

There are many, many different variants of Hungarian notation. A quick perusal of the Internet will demonstrate that almost no one really agrees on what symbols should be tags, prefixes, or qualifiers, much less what the individual symbols in each of the classes actually mean. Fortunately, the Windows header files are fairly consistent with respect to their use of Hungarian notation (at least, where they use Hungarian notation), so there won't be much difficulty deciphering the names from the header files that we'll use in this book.

As for using Hungarian notation in new identifiers appearing in this book, that will only happen when it's really convenient to do so. In particular, you'll see the "h" (for Handle) and "p" (for pointer) prefixes used quite a bit. Once in a while, you may see some other bits and pieces of Hungarian notation in use (e.g., b, w, and dw for `byte`, `word`, and `dword` objects). Beyond that, this book will attempt to use descriptive names, or at least, commonly used names (e.g., `i, j,` and `k` for array indexes) rather than trying to explain the identifier with a synthetic prefix to the identifier.

3.8: The w.hhf Header File

Provided with the HLA distribution is the *w.hhf* include file that define most of the Win32 API functions, constants, types, variables, and other objects you'll ever want to reference from your assembly language code. All in all, you're talking well over 30,000 lines of source code! It is convenient to simply stick an HLA `#include` statement like the following into your program and automatically include all the Windows definitions:

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

The problem with doing this is that it effectively increases the size of your source file by 30,000 lines of code. Fortunately, recent modifications to HLA have boosted the compile speed of this file to about 25,000 lines/second, so it can process this entire include file in just a few seconds. Most people don't really care if an assembly takes two or three seconds, so including everything shouldn't be a problem (note that the inclusion of all this code does not affect the size of the executable nor does it affect the speed of the final program you're writing; it only affects the compile-time of the program). For those who are bothered by even a few seconds of compile time, there is a solution.

A large part of the problem with the HLA/Windows header files is that the vast majority of the time you'll never use more than about 10% of the information appearing in these header files. Windows defines a tremendous number of esoteric types, constants, and API functions that simply don't get used in the vast majority of Windows applications. If we could pick out the 10% of the definitions that you were actually going to use on your next set of projects, we could reduce the HLA compilation overhead to almost nothing, far more acceptable to those programmers that are annoyed by a few seconds of delay. To do this, you've got to extract the declarations you need and put them in a project-specific include file. The examples in this book won't bother with this (because compiling the *w.hhf* file is fast enough), but feel free to do this if HLA compile times bother you.

3.9: And Now, on to Assembly Language!

Okay, we've probably spent enough time discussing C/C++ in a book dedicated to writing Windows programs in assembly language. The aim of this chapter has been to present a firm foundation for those who need to learn additional Windows programming techniques by reading C/C++ based documentation. There is no way a single chapter (even one as long as this one) can completely cover all the details, but there should be enough information in this chapter to get you well on your way to the point of understanding how to interface with Windows in assembly language by reading C/C++ documentation on the subject. Now, however, it's time to turn our attention to writing actual Windows applications in assembly language.

Chapter 4: The RadASM IDE for HLA

4.1: Integrated Development Environments

An integrated development environment (IDE) traditionally incorporates a text editor, a compiler/assembler, a linker, a debugger, a project manager, and other development tools under the control of a single main application. *Integrated* doesn t necessarily mean that a single program provides all these functions. However, the IDE does automatically run each of these applications as needed. An application developer sees a single user interface to all these tools and doesn t have to learn different sets of commands for each of the components needed to build an application.

The central component of most IDEs is the editor and project manager. A *project* in a typical IDE is a related collection of files that contain information needed to build a complete application. This could, for example, include assembly language source files, header files, object files, libraries, resource files, and binary data files. The point of an IDE project is to collect and manage these files to make it easy to keep track of them.

Most IDEs manage the files specific to a given project by placing those files in a single subdirectory. Shared files (such as library and shared object code files) may appear elsewhere but the files that are only used for the project generally appear within the project directory. This makes manipulation of the project as a whole a bit easier.

RadASM, created by Ketil Olsen, is a relatively generic integrated development environment. Many IDEs only work with a single language or a single compiler. RadASM, though designed specifically for assembly language development, works with a fair number of different assemblers. The nice thing about this approach is that you may continue to use RadASM when you switch from one assembler to another. This spares you the effort of having to learn a completely new IDE should you want to switch from one assembler to another (e.g., switching between FASM, MASM, NASM, TASM, and HLA is relatively painless because RadASM supports all of these assemblers; however, were you to switch from SpASM to one of these assemblers you d have to relearn the IDE because SpASM has it s own unique IDE). One drawback to generic IDEs is that they aren t as well integrated with the underlying toolset as an IDE designed around a specific toolset. RadASM, however, is extremely customizable, allowing you to easily set it up with different assemblers/compilers or even modify it according to your own personal tastes. By properly setting up RadASM to work with HLA, you can speed up the development of assembly language software under Windows. A fair percentage of this chapter deals with setting up and customizing the RadASM environment to suit your preferences and typical Windows assembly development.

4.2: Traditional (Command Line) Development in Assembly

RadASM isn t a *integrated* development environment in the sense that it integrates all of its functionality into the same program. Instead, RadASM is a shell program that invokes other applications in response to requests or commands within the IDE. For example, to compile an HLA program, RadASM does not include a copy of the HLA compiler within RadASM - it simply runs the HLA.EXE compiler in response to an assemble/compile command. Part of the work involved in customizing RadASM is to define the command line parameters to send to various tools like the HLA.EXE compiler. Therefore, in order to properly set up RadASM to operate HLA, you need to understand how to use HLA (and other tools) from a command line interface. Indeed, there are many times when it s much more convenient to use HLA from the command line rather than from an IDE. For these two reasons, it s important to first begin the discussion of using RadASM by describing how to use HLA without RadASM - straight from the command line.

In this book, we ll take the approach of always supporting both RadASM and command-line projects. This approach has the disadvantage of negating some of the benefits of using an IDE (i.e., automatic project maintenance) but it has the bigger advantage of being more flexible and not tying you down to using RadASM (should you prefer not to use RadASM for any reason, temporary or permanent). Fortunately, RadASM is flexible enough to support just about any command-line based development scheme you can throw at it, so supporting both schemes is not all that difficult.

This book assumes that you re already comfortable using the HLA compiler from the command line. This is not an unreasonable assumption because this book also assumes that you already know HLA and using HLA generally involves compiling files using command line tools. If you re not already comfortable with HLA command-line options, please take a moment to review these options in the HLA reference manual.

For very simple projects, those involving only a single code file (and, possibly, a few include files), building an HLA project into an executable is fairly trivial - simply specify the program s name on the HLA command line and let HLA take care of all the rest of the work for you, e.g.,

```
hla ProgramToCompile
```

Assuming *ProgramToCompile.hla* is a complete, compilable, HLA source file, this command will produce the *ProgramToCompile.exe* executable file. For many trivial projects (including many of those projects appearing early in this text), this is actually the most convenient way to produce an executable file from the HLA source file(s). However, as your projects grow more complex and wind up consisting of multiple source files that you compile separately and then link together, attempting to build the final executable by manually issuing a series of commands from the command line becomes rather tedious. Therefore, most programmers use the *make* subsystem to build their projects and they create make files to control the compilation of their projects. We ll use that same approach in this book.

4.3: HLA Project Organization

This book will adopt the (reasonable) convention of placing each HLA project in its own subdirectory. A given project directory will contain the following files and directories:

- All source files specific to the project (this includes make files, *.hla, .hhf, .rc, .rap* [RadASM project] and other files created specifically for this project, but does not include any standard library header or generic library files that all projects use).
- A makefile file that follows the template given in Chapter One.
- A Tmp subdirectory where HLA can place temporary files it creates during compilation (normally these files wind up in the same directory as the HLA source files; placing them in the Tmp directory prevents clutter of the main project directory).
- A Bak subdirectory where backup files can be kept.
- A Doc directory where project-related documentation can be found.

Every project in this book will have a makefile associated with it. You can either build the project by typing make from a command prompt window or you can build the project by selecting an appropriate option from the RadASM make menu (which processes this very same makefile). The WPA (Windows Programming in Assembly) subdirectory that holds all the projects associated with this book also has a makefile that will automatically build all projects associated with this text (this is useful, for example, for easily verifying that modifications to the HLA compiler don t break any of the example programs in this book)[1].

The RadASM IDE provides the ability to maintain projects itself. The examples in this book will not use the built-in RadASM project make facility for a couple of reasons: (1) the *make* program is more powerful, especially for larger projects involving many files; (2) this book has to create a makefile anyway (for command line processing of HLA projects) and maintaining (and keeping consistent) two different project management systems is problematic; and, finally, (3) the makefile scheme is a little more flexible.

The drawback to using makefiles to maintain the project is that you've got to manually create the makefile; RadASM won't do this for you automatically (as it does with its own projects). Fortunately, 90% of your makefile creations will simply be copying an existing makefile to your project's directory, editing the file, and changing the filenames from the previous project to the current project (indeed, this operation is so common that you'll find a generic makefile in the "sniplets" RadASM directory provide with the accompanying CD-ROM. You can easily create a copy of this generic makefile from RadASM's "Tools > Sniplets" menu, as you'll see soon enough).

4.4: Setting Up RadASM to Work With HLA

RadASM is a relatively generic integrated development environment for assembly language development. This single programs supports the HLA, MASM, TASM, NASM, and FASM assemblers. Each of these different assemblers feature different tool sets (executable programs), command line parameters, and ancillary tools. In order to control the execution of these different programs, the RadASM system uses ".INI" files to let you specifically configure RadASM for the assembler(s) you're using. HLA users will probably want to make modifications to two different ".INI" files that RadASM reads: *radasm.ini* and *hla.ini*. You'll find these two files in the subdirectory containing the *radasm.exe* executable file. Both files are plain ASCII text files that you can edit with any regular text editor (including the editor that is built into RadASM).

The RadASM package includes an ".RTF" (Word/Wordpad) documentation file that explains the basic format of these ".INI" files that RadASM uses. Readers interested in making major changes to these ".INI" files, or those attempting to adopt RadASM to a different assembler, will want to read that document. In this chapter, we'll explore the modifications to a basic set of ".INI" files that a typical HLA user might want to make. The assumption is that you're starting with the stock *radasm.ini* and *hla.ini* files that come with RadASM and you're wanting to customize them to support the development paradigm that this book proposes.

4.4.1: The RADASM.INI Initialization File

The *radasm.ini* file specifies all the generic parameters that RadASM uses. In particular, this ".INI" file specifies initial window settings, file histories, OS and language information, and menu entries for certain user modifiable menus. RadASM, itself, actually modifies most of the information in this ".ini" file. However, there are a few entries an HLA user will need to change and a couple of entries an HLA user may want to change. We'll discuss those sections here.

Note: there is a preconfigured radasm.ini file found in the WPA samples subdirectory. This initialization file is compatible with all the sample programs found in this book and is a good starting point should you decide to make your own customizations to RadASM.

The first item of interest in the *radasm.ini* file is the "[Assembler]" section. This section in the ".INI" file specifies which assemblers RadASM supports and which assembler is the default assembler it will use when creating new projects. By default, the "[Assembler]" section takes the following form:

1. The WPA subdirectory is found in the "Examples" module of the HLA download on Webster.

```
[ Assembler]
Assembler=masm,fasm,tasm,nasm,hla
```

The first assembler in this list is the default assembler RadASM will use when creating a new project. The standard *radasm.ini* file is set up to assume that MASM is the default assembler (the first assembler in the list is the default assembler). HLA users will probably want to tell RadASM to use HLA as the default assembler, this is easily achieved by changing the Assembler= statement to the following:

```
[ Assembler]
Assembler=hla,masm,fasm,tasm,nasm
```

Changing the default assembler is the only necessary change that you ll need to make. However, there are a few additional changes you ll probably want that will make using RadASM a little nicer. Again, by default, RadASM assumes that you re developing MASM32 programs. Therefore, the help menu contains several entries that bring up help information for MASM32 users. While some of this information is, arguably, of interest to HLA users, a good part of the default help information doesn t apply at all to HLA. Fortunately, RadASM s *radasm.ini* file lets you specify the entries in RadASM s help menu and where to locate the help files for those menu entries. The [MenuHelp] and [F1-Help] sections specify where RadASM will look when the user requests help information (by selecting an item from the Help menu or by pressing the F1 key, respectively). The default *radasm.ini* file specifies these two sections as follows:

```
[ MenuHelp]
1=&Win32 Api,0,H,$H\Win32.hlp
2=&X86 Op Codes,0,H,$H\x86eas.hlp
3=&Masm32,0,H,$H\Masm32.hlp
4=$Resource,0,H,$H\Rc.hlp
5=A&gner,0,H,$H\Agner.hlp

[ F1-Help]
F1=$H\Win32.hlp
CF1=$H\x86eas.hlp
SH1=$H\Masm32.hlp
CSF1=$H\Rc.hlp
```

Each numbered line in the [MenuHelp] section corresponds to an entry in RadASM s Help menu. These entries must be have sequential numbers starting from one and these numbers specify the order of the item in the Help menu (the order in the *radasm.ini* file does not specify the order of the entries in the Help menu, you do not have to specify the [MenuHelp] entries in numeric order, RadASM will rearrange them according to the numbers you specify). Entry entry in the [MenuHelp] section takes the following form:

```
menu# = Menu Text, accelerator, H, helpfile
```

where menu# is a numeric value (these values must start from one and there can be no gaps in the set), Menu Text is the text that RadASM will display in the menu for that particular item, accelerator is a Windows accelerator key value (generally, this is zero, meaning no accelerator value), H is required by RadASM to identify this as a Help entry, and helpfile is the path to the help file to display (or a program that will bring up a help file).

You may have noticed the ampersand character (&) in the menu text. The ampersand precedes the character you can press on the keyboard to select a menu item when the menu is opened. For example, pressing X when the menu is open (with the [HelpMenu] items in this example) selects the X86 Op Codes menu entry.

You will note that the paths in the [MenuHelp] section all begin with $H . This is a RadASM shorthand for the path where RadASM can find all the help files. There is no requirement that you use this shortcut or even place all your help files in the same directory. You could just also specify the path to a particular help file using a fully qualified pathname like *c:\hla\doc\Win32.hlp*. However, it s often convenient to specify paths using the various shortcuts that RadASM provides. RadASM supplies the shortcuts found in Table 4-1.

Table 4-1: Path Shortcuts for Use in RadASM ".INI" Files

Shortcut	Meaning
$A=	Path to where RadASM is installed
$B=	Where RadASM finds binaries and executables (e.g., c:\hla)
$D=	Where RadASM finds "Addin" modules. Usually $A\AddIns.
$H=	Where RadASM finds "Help" files. Default is $A\Help, but you'll probably want to change this to $B\Doc.
$I=	Where RadASM finds include files. Default is $A\Include, but you'll probably want to change this to $B\include.
$L=	Where RadASM finds library files. Default is $A\Lib but you'll probably want to change this to $B\hlalib.
$R=	Path where RadASM is started (e.g., c:\RadASM).
$P=	Where RadASM finds projects. This is usually $R\Projects.
$S=	Where RadASM find snippets. This is usually $R\Snippets.
$T=	Where RadASM finds templates. This is usually $R\Templates
$M=	Where RadASM finds keyboard macros. This is usually $R\Macro

You can define several of these variables in the *hla.ini* file. See the next section for details.

As noted earlier, the default help entries are really intended for MASM32 users and do not particularly apply to HLA users. Therefore, it s a good idea to change the [MenuHelp] entries to reflect the location of some HLA-related help files. Here are the [MenuHelp] entries that might be more appropriate for an HLA installation (assuming, of course, you ve placed all these help files in a common directory on your system):

```
[MenuHelp]
1=&Win32 Api,0,H,$H\Win32.hlp
2=&Resource,0,H,$H\Rc.hlp
3=A&gner,0,H,$H\Agner.hlp
4=&HLA Reference,0,H,$H\PDF\HLARef.pdf
5=HLA Standard &Library,0,H,$H\pdf\HLAStdlib.pdf
6=&Kernel32 API,0,H,$H\pdf\kernelref.pdf
7=&User32 API,0,H,$H\pdf\userRef.pdf
8=&GDI32 API,0,H,$H\pdf\GDIRef.pdf
```

Here s a suggestion for the F1, Ctrl-F1, Shift-F1, and Ctrl-Shift-F1 help items:

```
[ F1-Help]
F1=$H\Win32.hlp
CF1=$H\PDF\HLARef.pdf
SF1=$H\pdf\HLAStdlib.pdf
CSF1=$H\Rc.hlp
```

These are probably the extent of the changes you ll want to make to the radasm.ini file for HLA use; there are, however, several other options you can change in this file, please see the *radASMini.rtf* file that accompanies the RadASM package for more details on the contents of this file.

4.4.2: The HLA.INI Initialization File

The *hla.ini* file is actually where most of the customization for HLA takes place inside RadASM. This file lets you customize RadASM s operation specifically for HLA[2]. The `hla.ini` file appearing in the WPA subdirectory (on the accompanying CD-ROM or in the Webster HLA/Examples download file) contains a set of default values that provide a good starting point for your own customizations.

Note: although *hla.ini* provides a good starting point for a system, you will probably need to make changes to this file in order for it to work on your specific system. Without these changes, RadASM may not work on your system.

Without question, the first section to look at in the *hla.ini* file is the section that begins with `[Paths]` . This is where you tell RadASM the paths to various directories where it expects to find various files it needs (see Table 4-1 for the meaning of these various path values). A typical `[Paths]` section might look like the following:

```
[ Paths]
$A=C:\Hla
$B=$A
$D=$R\AddIns
$H=$A\Doc
$I=$A\Include
$L=$A\hlalib
$P=$R\Hla\Projects
$S=$R\Hla\Sniplets
$T=$R\Hla\Templates
$M=$R\Hla\Macro
```

Note that the `$A` prefix specifies the path where RadASM can find the executables for HLA. In fact, RadASM does not run HLA directly (remember, we re going to have the make program run HLA for us), but the application path (`$A`) becomes a prefix directory we ll use for defining other directory prefixes. **Be sure to check this path** in your copy of the `hla.ini` file and verify that it points at your main HLA subdirectory (usually C:\HLA though this may be different if you ve installed HLA elsewhere).

The `$R` prefix specifies the path to the subdirectory containing RadASM. RadASM automatically sets up this prefix, you don t have to explicitly set its value. The remaining subdirectory paths are based off either the `$A` prefix or the `$R` prefix.

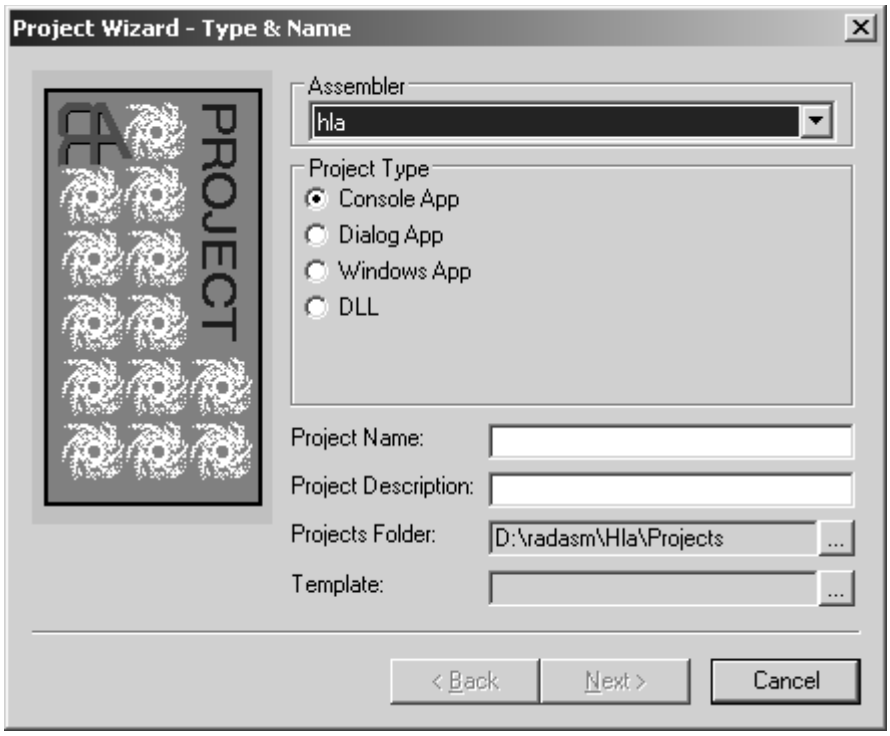
The [Project] section of the hla.ini file is where the fun really begins. This section takes the following form in the default file provided in the WPA subdirectory:

2. There are comparable initialization files for MASM, TASM, NASM, FASM, and other assemblers that RadASM supports.

```
[ Project]
Type=Console App,Dialog App,Windows App,DLL
Files=hla,hhf,rc,def
Folders=Bak,Res,Tmp,Doc
MenuMake=Build,Build All,Compile RC,Check Syntax,Run
Group=1
GroupExpand=1
```

The line beginning with *Type=* specifies the type of projects RadASM supports for HLA. The default configuration supports console applications (Console App), dialog applications (Dialog App), Windows applications (Windows App), and dynamic linked library (DLL). The names are arbitrary, though other sections of the *hla.ini* file will use these names. Whenever you create a new project in HLA, it will create a list of Project Type names based on the list of names appearing after *Type=* in the *[Project]* section. Adding a string to this comma-separated list will add a new name to the project types that the RadASM user can select from (note, however, that to actually support these project types requires some extra work later on in the *hla.ini* file). Figure 4-1 shows what the New Project dialog box in RadASM displays in response to the entries on the Type=... line.

Figure 4-1: RadASM Project Types



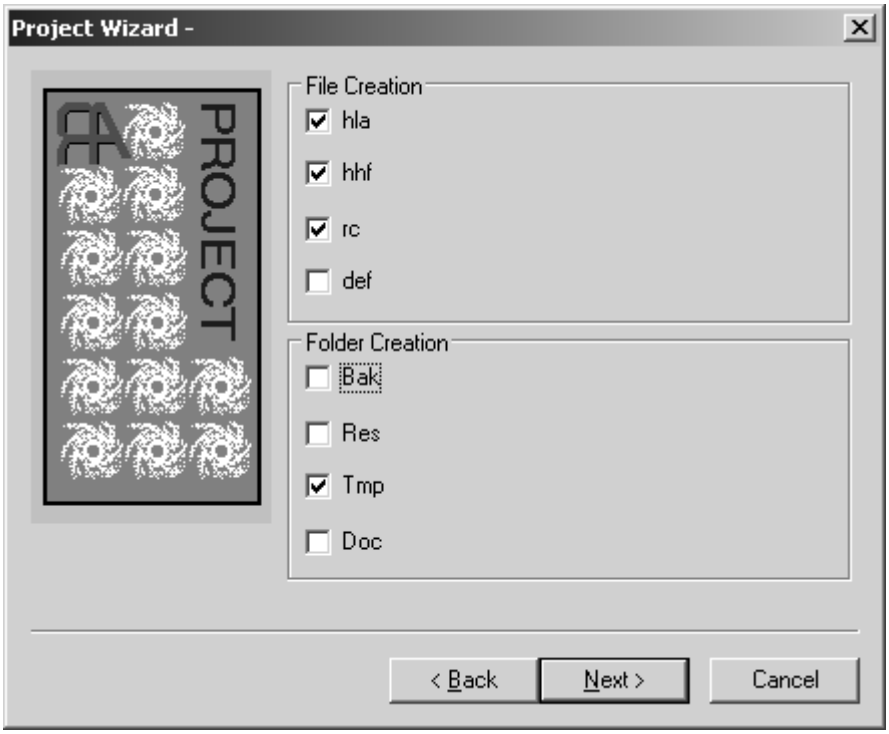
The line beginning with Files= in the [Project] section specifies the suffixes for the files that RadASM will associate with this project. The hla and hhf entries, of course, are the standard file types that HLA uses. The .rc file type is for *resource compiler* files (we ll talk about the resource compiler in a later chapter). If you want to be able to create additional file types and include them in a RadASM project, you would add their suffix here.

The Folders=... statement tells RadASM what subdirectories it should allow the user to create when they start a new project. The make file system we re going to use will assume the presence of a Tmp directory,

hence that option needs to be present in the list. the bak , res , and doc directories let the user create those subdirectories.

Figure 4-2 shows the dialog box that displays the information found on the Files= and Folders= lines. By checking the appropriate boxes in the File Creation group, the RadASM user can tell RadASM to create a file with the project s name and the appropriate suffix as part of the project. Similarly, by checking the appropriate boxes in the Folder Creation group, the RadASM user can tell RadASM to create the appropriate directories.

Figure 4-2: File and Folder Creation Dialog Box in RadASM



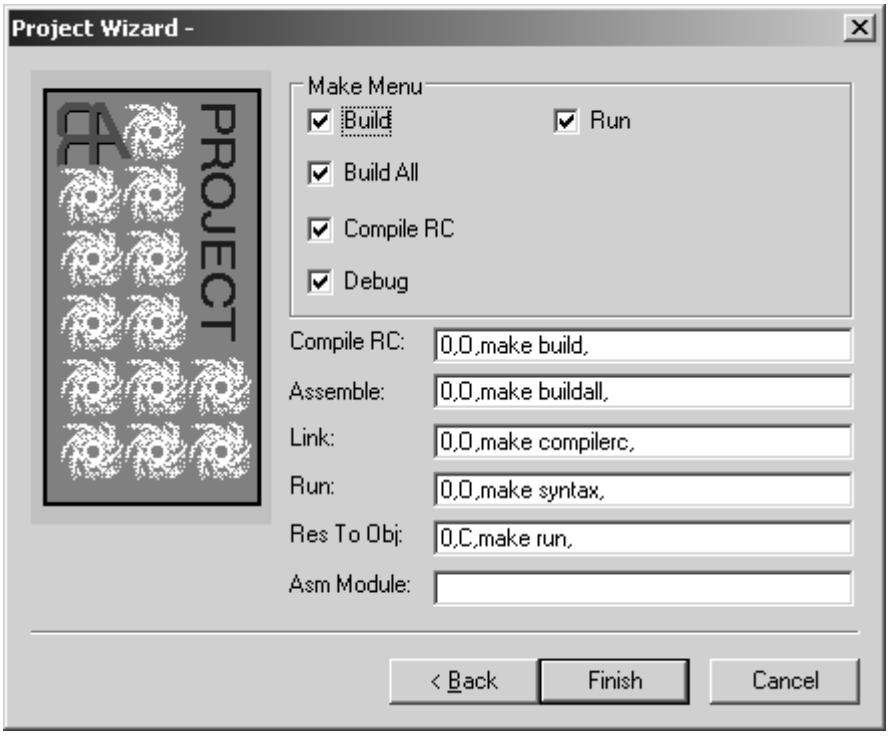
The MenuMake=... line specifies the IDE options that will be available for processing the files in this project. Unlike the other options, you cannot specify an arbitrary list of commands here. RadASM provides five items in the Make menu that you can modify, you don t have the option of adding additional items here (you can disable options if you want fewer, though). Originally, these five slots were intended for the following commands:

- Compile RC (compile a resource file)
- Assemble (assemble the currently open file)
- Link (create an EXE from the object files)
- Run (execute the EXE file, building it if necessary)
- Res To Obj (convert a resource file to an object file)

Because these options aren t as applicable to HLA projects as they are to MASM projects (on which the original list was built), the default *hla.ini* file co-opts these make items for operations that make more sense for the way we re going to be building HLA applications in this book. You can actually turn these make items on or off on a project by project basis (for certain types of projects, certain make objects may not make sense). Figure 4-3 shows the dialog box that RadASM displays and presents this information. Note that in the version used here, RadASM only displays the correct labels for the check boxes in the Make Menu group. The labels on the text

entry boxes should also be Build , Build All , Compile RC , Check Syntax , and Run (in that order), but these labels turn out to be hard-coded to the original MASM specifications. Fortunately, you won t normally use these text entry boxes (the default text appearing in them appears in the *hla.ini* file), so you can ignore the fact that they are mis-labelled here.

Figure 4-3: RadASM Make Menu Selection Dialog Box



For each of the project types you specify on the `Type=...` line in the `[Project]` section, you must add a section to the hla.ini file, using that project type s name, that tells RadASM how to deal with projects of that type. In the hla.ini file we re discussing here, the project types are Console App , Dialog App , Windows App , and DLL so we will need to have sections names `[Console App]` , `[Dialog App]` , `[Windows App]` , and `[DLL]` . It also turns out that RadASM requires one additional section named `[MakeDefNo-Project]` that RadASM uses to process files in the IDE that are not associated with a specific project.

When running RadASM, you can have exactly one project open (or no project at all open, just some arbitrary files) at a time. This project will be one of the types specified on the `Type=...` line in the `[Project]` section. Based on the open project, RadASM may execute a different set of commands for each of the items in the Make menu; the actual commands selected are specified in the project-specific sections of the *hla.ini* file. Here s what the `[MakeDefNoProject]` section looks like:

```
[ MakeDefNoProject]
MenuMake=1,1,1,1,1
1=0,O,make build,
2=0,O,make buildall,
3=0,O,make compilerc,
4=0,O,make syntax,
5=0,O,make run,
11=0,O,make dbg_build,
12=0,O,make dbg_buildall,
```

```
13=0,O,make dbg_compilerc,
14=0,O,make dbg_syntax,
15=0,C,make dbg_run,
```

The MenuMake=... line specifies which items in the RadASM Make menu will be active when a project of this type is active in RadASM (or, in the case of *MakeDefNoProject*, when no project is loaded). This is a list of boolean values (true=1, false=0) that specify whether the menu items in the Make menu will be active or deactivated. Each of these values correspond to the items on the MenuMake line in the [Project] section (in our case, this corresponds to Build , Build All , Compile RC , Syntax Check , and Run , in that order). A 1 activates the corresponding menu item, a zero deactivates it. For most HLA project types, we ll generally leave all of these options enabled. The exception is DLL; normally you don t run DLLs so we ll disable the run option when building DLL projects.

The remaining lines specify the actions RadASM will take whenever you select one of the items from the Make menu. To understand how these items work, let s first take a look at another section in the hla.ini file, the *[MenuMake]* section:

```
[ MenuMake]
1=&Build,55,M,1
2=Build &All,31,M,2
3=&Compile RC,91,M,3
4=&Syntax,103,M,4
5=-,0,M,
6=&Run,67,M,5
```

Each item in the *[MenuMake]* section corresponds to a menu entry in the Make menu. The numbers specify the index to the menu entry (e.g., 1= specifies the first menu item, 2= specifies the second menu item, etc.). The first item after the n= prefix specifies the actual text that will appear in the Make menu. If this text is just the character - then RadASM displays a menu separator for that particular entry. As you can see, the default menu entries are Build , Build All , Compile RC , Syntax , and Run .

The next item, following the menu item text, is the accelerator value. These are magic values that specify keystrokes that do the same job as selecting items from the menu. For example, 55 (in the Build item) corresponds to Shift+F5, 31 (in Build All) corresponds to F5. We ll discuss accelerators in a later chapter. So just ignore (and copy verbatim) these files for right now.

The third item on each line is always the letter M . This tells RadASM that this is a make menu item.

The fourth entry on each line is probably the most important. This is the command to execute when someone selects this particular menu item. This is either some text containing the command line to execute or a numeric index into the current project type. As you can see in this example, each of the commands use an index value (one through five in this example). These numbers correspond to the lines in each of the project sections. For example, if you select the Build option from the Make menu, RadASM notes that it is to execute command #1. It goes to the current project type section and locates the line that begins with 1=... and executes that operation, e.g.,

```
1=0,O,make build,
```

In a similar vein, selecting Build All from the Make menu instructs RadASM to execute the command that begins with 2=... in the current project type s section (i.e., *2=0,O,make buildall,*). And so on.

The lines in the project type section are divided into two groups, those that begin with 1, 2, 3, 4, or 5 and those that begin with 11, 12, 13, 14, or 15. The [MenuMake] command index selects one of the commands

from these two groups based on whether RadASM is producing a release build or a debug build. Release builds always execute the command specified by the [MenuMake] command index (i.e. 1-5). If you're building a debug version, then RadASM executes the commands in the range 11-15 in response to command indexes 1-5. We'll ignore debug builds for the time being (we'll discuss them in a later chapter on debugging). So for right now, we'll always assume that we're building a release image.

The fields of each of the indexed commands in the project type section have the following meanings:

```
index = delete_option, output_option, command, files
```

The *delete_option* item specifies which files to delete before doing the build. If this entry is zero, then RadASM will not delete any files before the build. Because we're having a make file do the actual build for us, and it can take care of cleaning up any files that need to be deleted first, we'll always put a zero here when using RadASM with HLA.

The *output_option* item is either C, O (that's an oh not a zero), or zero. This specifies whether the output of the command will go to a Windows console window (C), the RadASM output window (O, which is oh), or the output will simply be thrown away (zero). We'll usually want the output sent to RadASM's output window, so most of the time you'll see the letter O (oh) here.

The *command* entry is the command line text that RadASM will pass on to windows whenever you execute this command. This can be any valid command prompt operation. For our purposes, we'll always use a make command with a single parameter to specify the type of make operation to perform. Here are the commands we're going to support in RadASM:

- Build - make build
- Build All - make buildall
- Compile RC - make compilerc
- Syntax - make syntax
- Run - make run

Now it's up to the makefile to handle each of these various commands properly (using the standard makefile scheme we defined in the first chapter).

This may seem like a considerable amount of indirection -- why not just place the commands directly in the *[MenuMake]* section? However, this scheme is quite flexible and makes it easy to adjust the options on a project type by project type basis (in fact, it's even possible to set these options on a project by project basis).

With this discussion out of the way, it's time to look at the various project type sections. Without further ado, here they are:

```
[Console App]
Files=1,1,1,1,0,0
Folders=1,0,1,0
MenuMake=1,1,1,1,1,0,0,0
1=0,O,make build,
2=0,O,make buildall,
3=0,O,make compilerc,
4=0,O,make syntax,
5=0,C,make run,
11=0,O,make build,
12=0,O,make buildall,
13=0,O,make compilerc,
```

```
14=0,O,make syntax,
15=0,C,make run,
```

Console applications, by default, want to create an .HLA file and a .HHF file, a BAK folder and a TMP folder. All menu items are active for building and running console apps (that is, there are five ones after *MenuMake*). Finally, the commands (1=... 2=... , etc.) are all the standard build commands.

```
[ Dialog App]
Files=1,1,1,0,0
Folders=1,1,1
MenuMake=1,1,1,1,1,0,0,0
1=0,O,make build,
2=0,O,make buildall,
3=0,O,make compilerc,
4=0,O,make syntax,
5=0,C,make run,
11=0,O,make build,
12=0,O,make buildall,
13=0,O,make compilerc,
14=0,O,make syntax,
15=0,C,make run,
```

By default, dialog applications will create HLA, HHF, RC, and DEF files and they will create a BAK and a TMP subdirectory. All five menu items will be active and dialog apps use the standard command set.

```
[ Windows App]
Files=1,1,1,1,0
Folders=1,1,1,1
MenuMake=1,1,1,1,1,0,0,0
1=0,O,make build,
2=0,O,make buildall,
3=0,O,make compilerc,
4=0,O,make syntax,
5=0,C,make run,
11=0,O,make build,
12=0,O,make buildall,
13=0,O,make compilerc,
14=0,O,make syntax,
15=0,C,make run,
```

By default, window applications will create HLA, HHF, RC, and DEF files and they will create a BAK, RES, DOC, and a TMP subdirectory. All five menu items will be active and dialog apps use the standard command set.

The hla.ini file allows you to control several other features in RadASM. The options we ve discussed in this chapter are the crucial ones you must set up, most of the remaining options are of an aesthetic or non-crucial nature, so we won t bother discussing them here. Please see the RadASM documentation (the RTF file mentioned earlier) for details on these other options.

Once you ve made the appropriate changes to the hla.ini file (and, of course, you ve made a backup of your original file, right?), then you can copy the file to the RadASM subdirectory and replace the existing hla.ini file with your new one. After doing this, RadASM should operate with the new options when you run RadASM..

4.4.3: Additional Support for RadASM on the CD-ROM

In addition to the *hla.ini* and *radasm.ini* files, there are several additional files of interest to RadASM users provided on the accompanying CD-ROM (or in the *WPA\RadASM* subdirectory of the Examples module download on the HLA download page at http://webster.cs.ucr.edu). This section will describe these additional files.

To install most of the auxiliary files appearing in the *WPA\RadASM* directory all you have to do is copy them to the corresponding folders in your RadASM folders. For example, you d normally just copy the *hla.ini* and *radasm.ini* files directly from the *WPA\RadASM* folder to your RadASM folder (or whatever folder contains the RadASM installation). In the WPA\RadASM folder you ll find a help folder. This folder contains several additional help files (to which the new *radasm.ini* file refers), so you ll want to copy these files into your RadASM s help folder[3]. The *WPA\RadASM* folder contains several API files and two subdirectories (Sniplets and Templates). You should copy the API files to your RadASM\HLA folder (after verifying that the api files are newer), you should copy the files in the Templates directory to the *RadASM\HLA\Templates* directory, and you should copy the files from the *Sniplets\Code* directory into your *RadASM\HLA\Sniplets\Code* subdirectory.

The Templates folder contains project templates you might find useful (for example, there is a template for creating a generic Win32 assembly application that is a great tool for starting new projects). The *Sniplets\Code* folder contains a generic makefile and a generic win32 application whose code you can cut and paste into a file to create customized versions of these files.

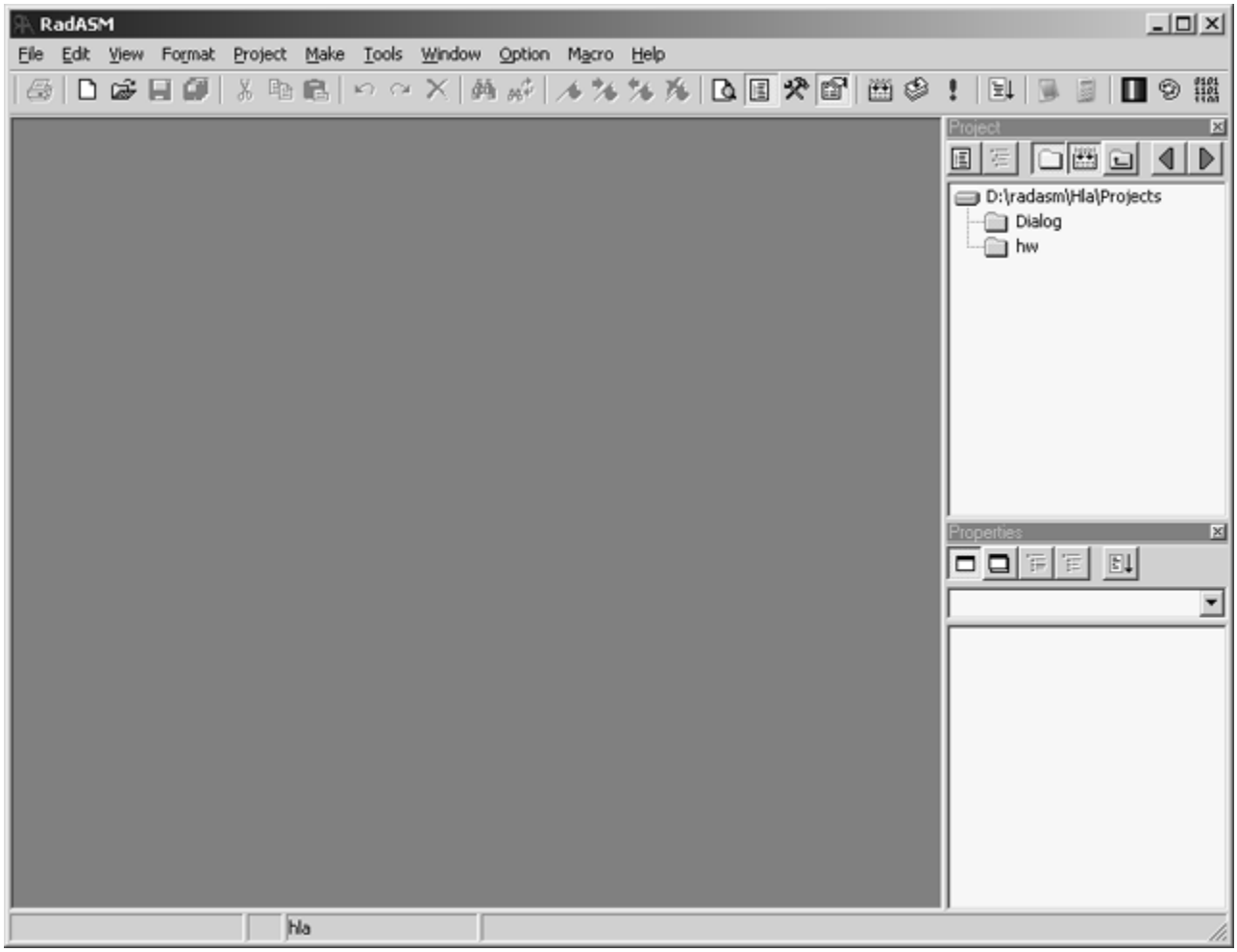
Note that because RadASM is under continuous development, it s quite possible that a version of RadASM you re downloading contains newer versions of the help, template, or sniplet files. Therefore, it s a good idea to make backups of the old files first, rather than simply replacing them with the files from the *WPA\RadASM* subdirectory. That way, should it turn out that RadASM ships with a new file, you can easily recover without downloading the RadASM package again. For the most part, however, you ll find that the files appearing in the *WPA\RadASM* subdirectory are new files that aren t part of the standard RadASM distribution.

4.5: Running RadASM

Like most Windows applications, you can run RadASM by double-clicking on its icon or by double clicking on a RadASM Project file (.rap suffix). Simply double-clicking on the RadASM icon brings up a window similar to the one appearing in Figure 4-4.

3. It might not be a bad idea to check the dates on any of the original files you re replacing to make sure that they re older than the files you re replacing them with.

Figure 4-4: RadASM Opening Screen



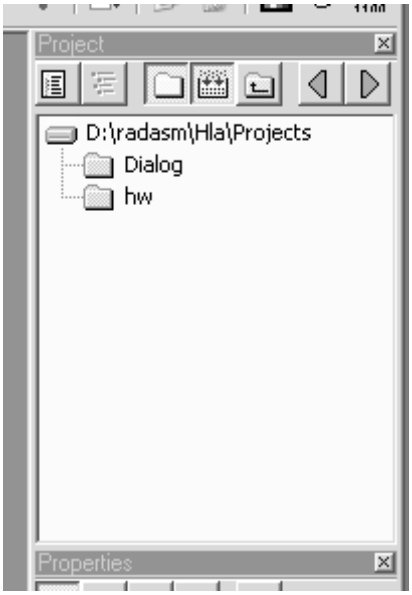
The main portion of the RadASM window is broken down into three panes. The larger of the three panes is where text editing takes place. The upper right hand pane is the project management window. The pane in the lower right hand corner lists the properties of the currently opened project.

4.5.1: The RadASM Project Management Window

The project management window initially lists the project folders you ve created; you can select an existing project by double-clicking on the project s folder in this window. For example, RadASM ships with two sample projects, *Dialog* (that creates a small dialog box application) and *hw* (that creates a small Hello World console

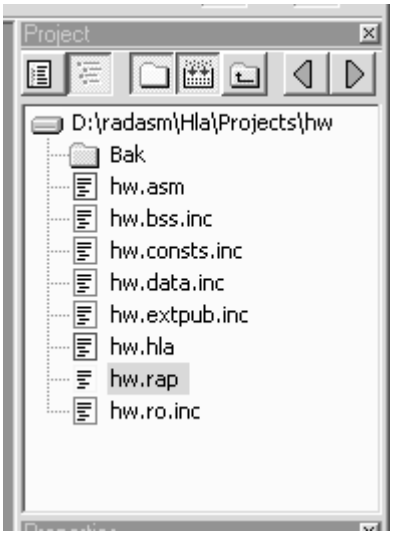
application). Assuming you re running RadASM prior to creating any new projects beyond these two default projects, the Project pane will look something like Figure 4-5.

Figure 4-5: Default RadASM Project Pane



Double-clicking on the hw folder opens the folder containing that project. This changes the pane to look something like that appearing in Figure 4-6.

Figure 4-6: RadASM Project Pane With hw Folder Opened



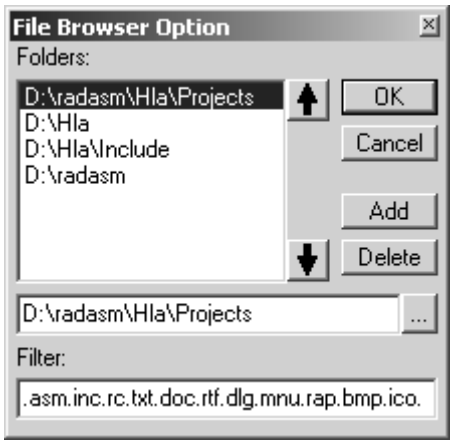
By default, RadASM does not show all the files present in the folder you ve opened. Instead, RadASM filters out files that don t have a certain file suffix. By default, RadASM only displays files with the following suffixes:

- .asm
- .inc
- .rc

- .txt
- .doc
- .rtf
- .dlg
- .mnu
- .rap
- .bmp
- .ico
- .cur
- .hla
- .hhf

This list is actually designed to generically handle all file types for every assembler that RadASM works with. HLA users might actually want to drop .asm and .inc from this list as files with these suffixes are temporary files that HLA produces (much like .obj files, which don t normally appear in this list). You can change the filter suffixes in one of two places. The first place is in the *radasm.ini* file. Search for the `[FileBrowser]` section and edit the line that begins with `Filter=...` . You can delete or add suffixes to your heart s content on this line. The second way to change the default filters, arguably the easiest way, is within RadASM itself. From the application s menu, select Option>File Browser (that is, select the File Browser menu item from the Option menu). This brings up the dialog box appearing in Figure 4-7. The text edit box at the bottom of this dialog window (labelled Filter:) lets you edit the suffixes that RadASM uses for filtering files in the Project window pane.

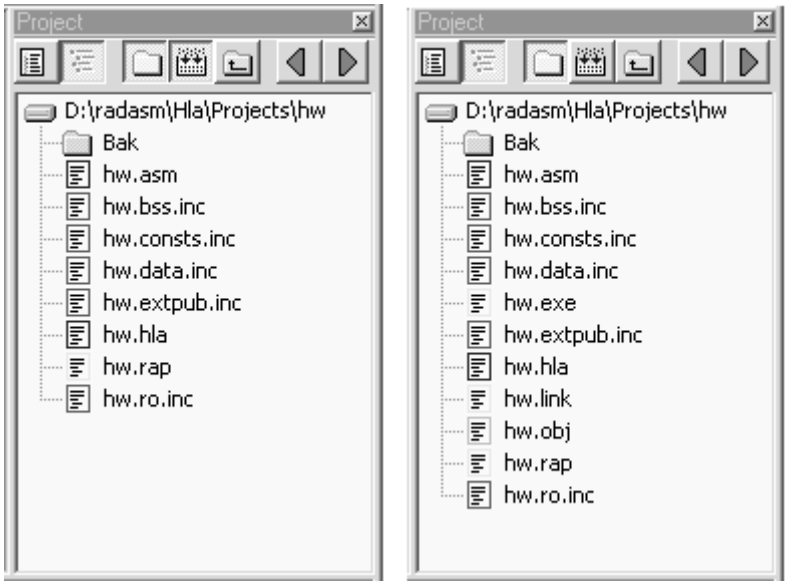
Figure 4-7: RadASM File Browser Options Dialog Box



By default, RadASM only displays those files whose file suffixes appears in the filter list. If, for some reason, you need to see all files that appear in a project subdirectory, you can turn the file filtering off. There is a toolbar button at the top of the Project window pane that lets you activate or deactivate file filtering. Clicking on this button toggles the display mode. So clicking on this button once will deactivate file filtering, to display all the files

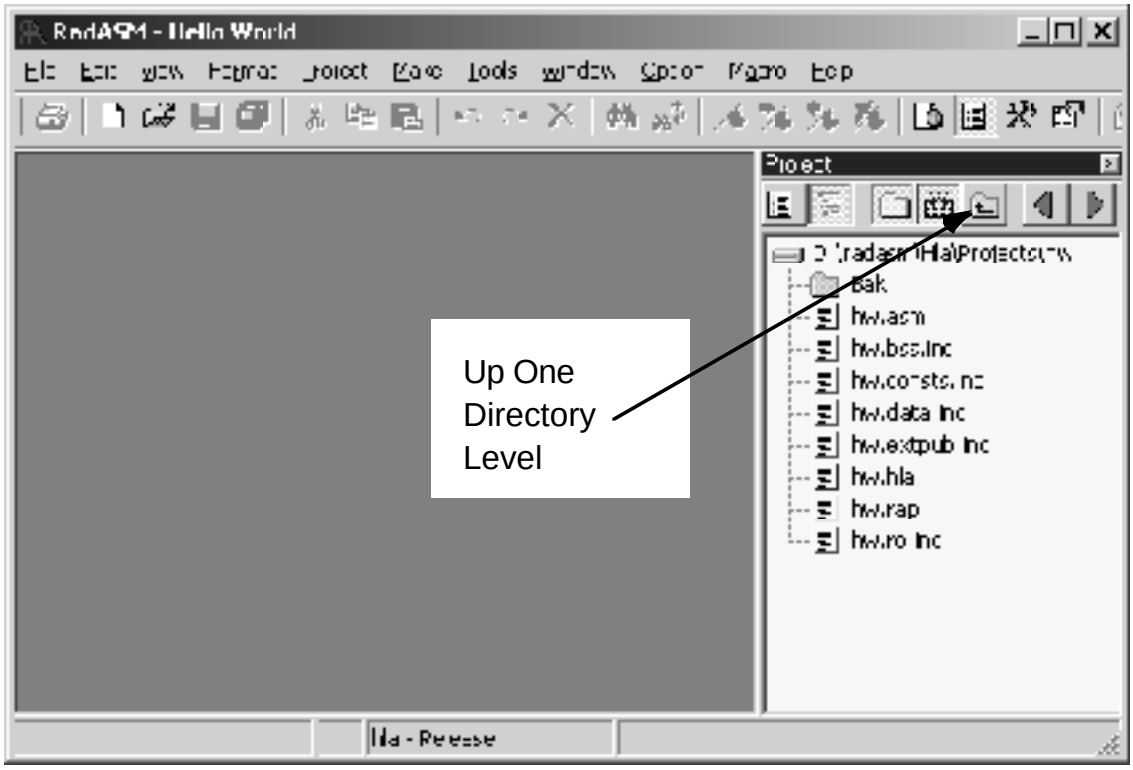
in the directory, clicking on this button a second time reactivates file filtering. Figure 4-8 shows the effects of clicking on this button.

Figure 4-8: File Filtering in RadASM's Project Pane



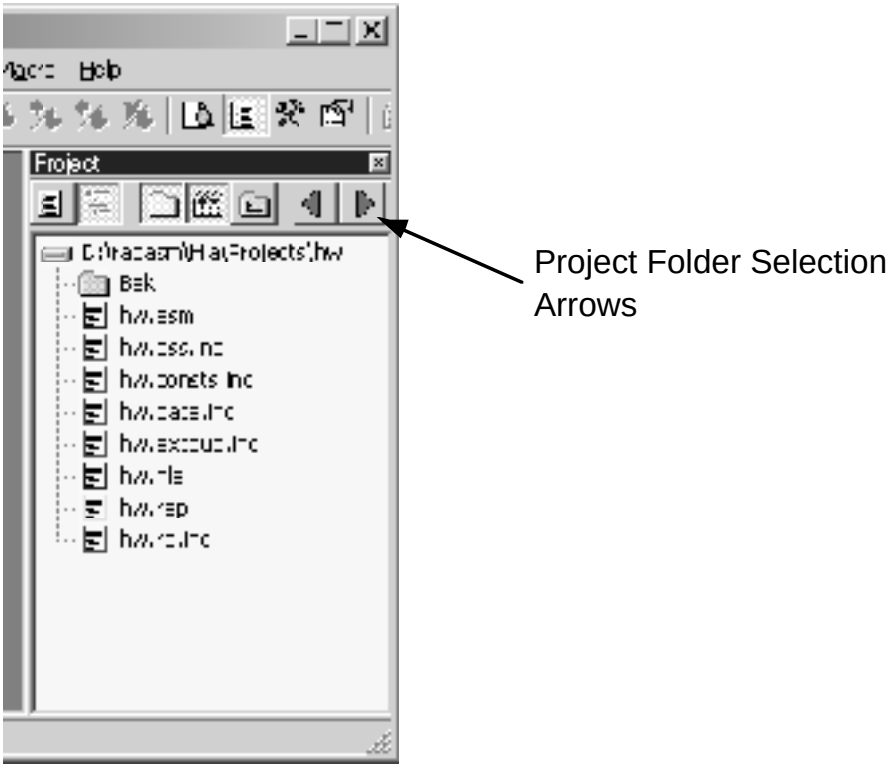
If you ve descended into a subdirectory by double-clicking on it s folder icon and you decide to return to an upper level directory, you can move to that upper level directory by clicking on the Up One Level button in the RadASM Project pane (see Figure 4-9).

Figure 4-9: RadASM "Up One Level" Directory Navigation Button



The left and right arrow buttons allow you to quickly scan through several different directories in the system (see Figure 4-10). By default, RadASM displays a couple of interesting (HLA-related) subdirectories in the Project pane when you scan through the list using the left and right arrows in the Project pane. In general, however, you ll want to customize the directories RadASM visits when you press these two arrow buttons. You can add (or change) directory paths in the `[FileBrowser]` section of the radasm.ini file, though it s probably easier to select the Option>File Browser menu item to open up the File Browser Option dialog box and make your changes there (see Figure 4-7). The Folders: list in the File Browser Option dialog box lists all the directories that RadASM will rotate through when you press the left and right buttons in the Project window pane. You can add, delete, edit, and rearrange the items in this list.

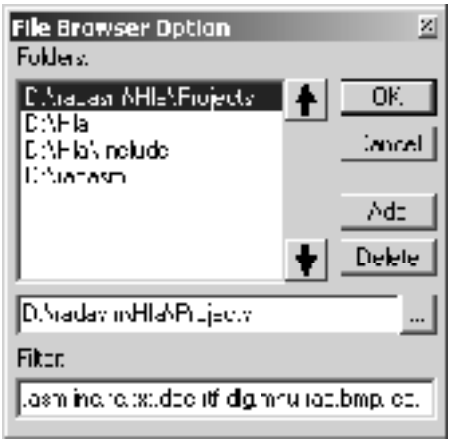
Figure 4-10: Project Folder Selection Arrows



To edit an existing entry, click on that entry with the mouse and then edit the directory path appearing in the text edit box immediately below the Folders: list (see Figure 4-7). You may either type in the path directly, or

browse for the path by pressing the browse button immediately to the right of the text entry box (see Figure 4-11).

Figure 4-11: The RadASM File Browser Option "Browse" Button



To delete an entry from the File Browser Option list, select that item with the mouse and then press the Delete button appearing in the File Browser Option Window. To add a new entry to the list, press the Add button and then type the path into the text edit box (or use the browse button to locate the subdirectory you want to add). **Note**: do not type the new entry in and then press Add . This sequence will change the currently selected item and then add a new, blank, entry. The correct sequence is to first press the Add button, and then edit the blank entry that RadASM creates.

The remaining buttons in the Project window are only applicable to open projects. Note that opening a project folder is not the same thing as opening a RadASM project. To open a RadASM project you must either create a new project or open an existing .rap file. For example, you can open the Hello World project in the *hw* directory by double-clicking on the *hw.rap* file that appears in the project window. Opening the hw.rap file does two things to the RadASM windows: first, it displays the hw.hla source file in the editor window and, second, it switches the Project window pane from File Browser mode to Project Browser mode. In project browser mode RadASM displays only the files you ve explicitly added to the project. Any incidental or generated files will not appear here (unless you explicitly add them). For example, whereas the File Browser mode displays several .inc and .asm files (assuming you ve not removed these suffixes from the file filter), the Project Browser mode only displays the *hw.hla* file because this is the only file that was originally added to the project. Another difference between the file browser and project browser modes is the fact that RadASM displays the files in pseudo-directories according to the file s type. For example, it displays the *hw.hla* file under the sub-heading Assembly (see Figure 4-12). The *hw.rap* project is a relatively simple project, only having a single assembly file. The Dialog.rap project (that appears in the Dialog project folder) is a slightly more complex application, having a couple of resource files in addition to an assembly file (see Figure 4-13). Note that you can flatten RadASM s view of these files by pressing the Project Groups button in the Project window pane (see Figure 4-14). Pressing this button a second time restores the project groups display.

Figure 4-12: Project Window "Project Browser Mode"

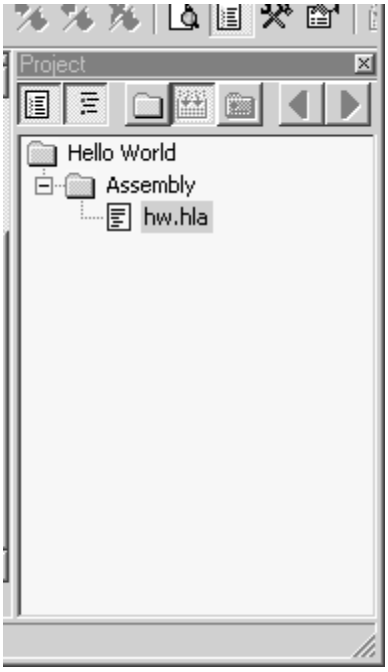


Figure 4-13: Dialog.rap Project Browser Display

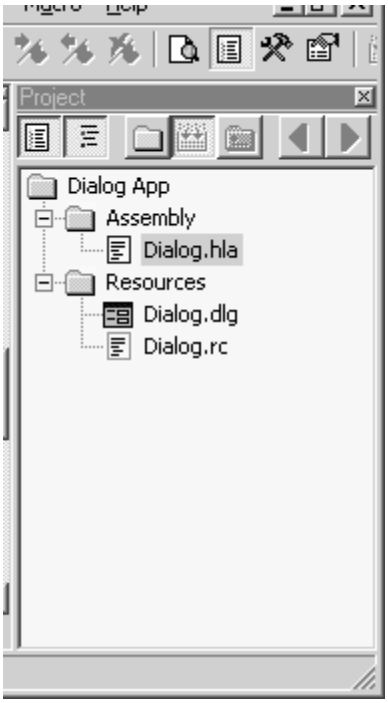
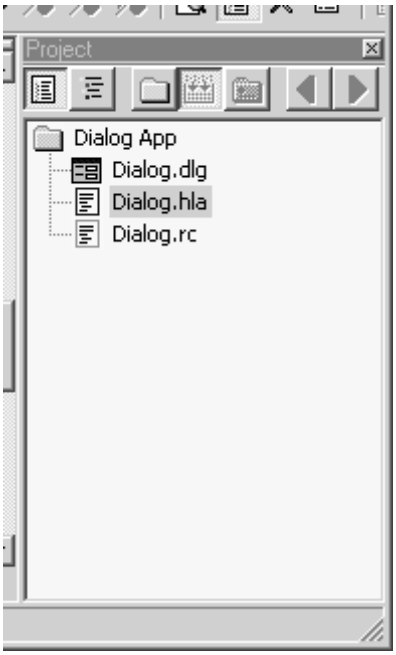
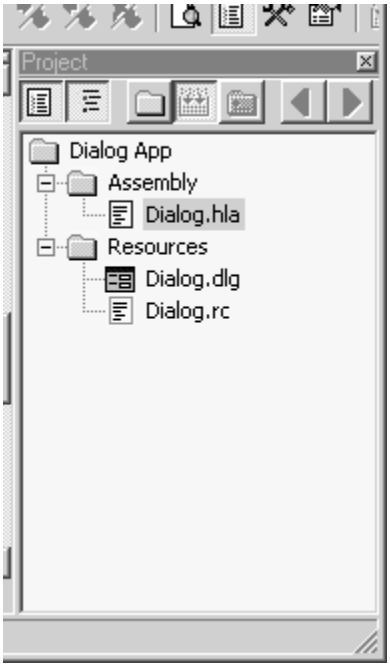


Figure 4-14: Effect of Pressing the "Project Groups" Button



When you ve got a project loaded, RadASM displays the project view by default. By pressing the File Browser and Project Browser buttons in the Project window pane, you can switch between these two views of your files (see Figure 4-15).

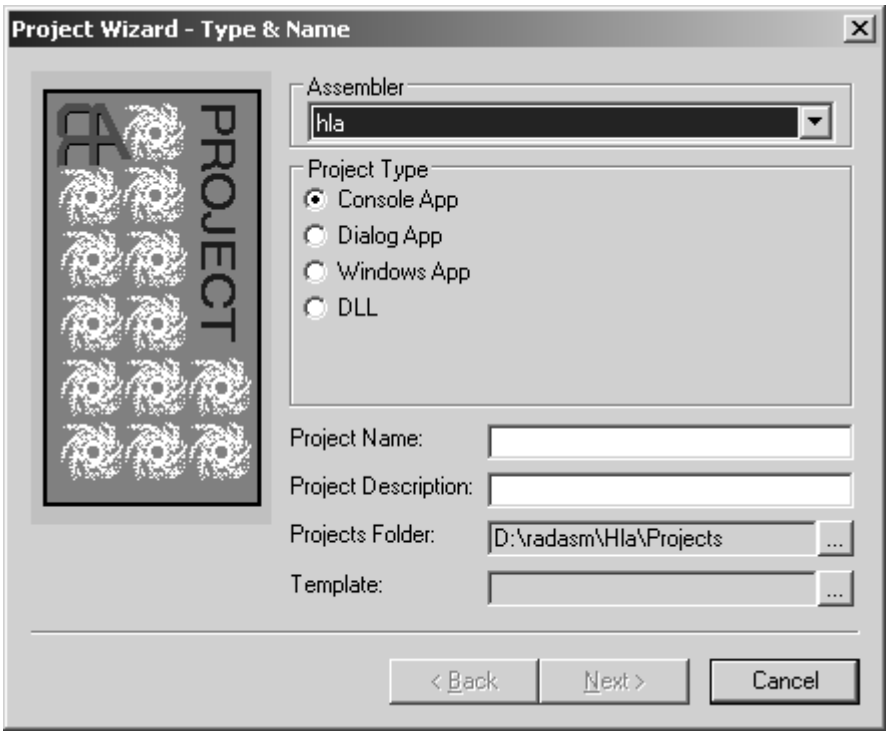
Figure 4-15: The Project Browser and File Browser Buttons



4.5.2: Creating a New Project in RadASM

While the two default projects that RadASM supplies are useful for demonstrating the RadASM Project window pane, you re probably far more interested in creating your own RadASM/HLA projects. Creating your own project is a relatively straight-forward process using RadASM s *project creation wizard*. To begin this process, select the File>New Project menu item. This opens the project wizard dialog box (see Figure 4-16).

Figure 4-16: RadASM Project Wizard Dialog Box



The Assembler pop-up menu list lets you select the assembler that you want to use for this project. Remember, RadASM supports a variety of different assemblers and the rules are different for each one. Because you re probably using HLA (if you re reading this book), you ll want to select the HLA assembler from this list. HLA should be the default assembler in this list assuming you ve placed hla at the front of the list in the `[Assembler]` section of the *radasm.ini* file. If you re not using the *radasm.ini* file supplied on the CD-ROM accompanying this book (or in the WPA\RadASM subdirectory in the HLA Examples download package on Webster), then you should make sure that HLA appears first in this list in the *radasm.ini* file.

The Project Type group is a set of radio buttons that let you select the type of project you re creating. RadASM populates this list of radio buttons from the `[Project]` section of the hla.ini file. The `Type=...` statement in this section specifies the valid projects that RadASM will create. RadASM creates the radio button items in the order the project type names appear in the `Type=...` list; the first item in the list is the one that will have the default selection. If you re going to be developing Windows GUI applications most of the time, you ll probably want to change this list so that Windows App appears first in the list. This will slightly streamline the use of the Project Wizard because you won t have to explicitly select Windows App every time you create a new Windows application.

The Project Name: text entry box is where you specify the name of the project you re creating. RadASM will create a folder by this name and any other default files it creates (within the project folder) will also have this name as their filename prefix. The text you enter at this point must be a valid Windows filename. Note that this

should be a simple file name (directory name), not a path. You ll supply the path to this file/directory in a moment.

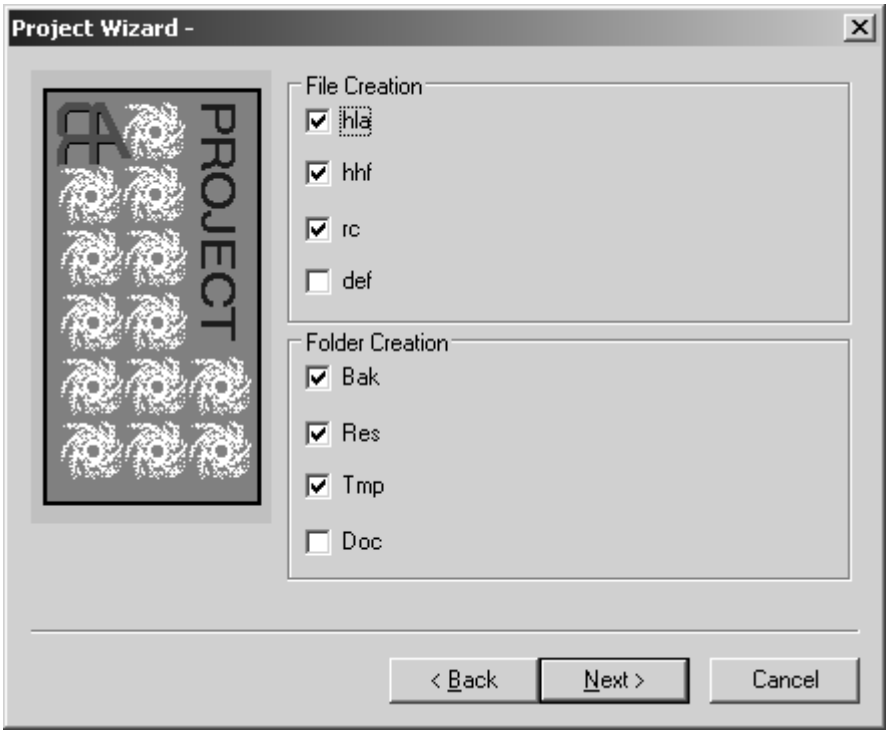
The Project Description: text entry box allows you to place a descriptive comment that describes the project. This is any arbitrary text you choose. It should be a brief (one-line) description of the project.

The Projects Folder: text entry box is where you select the path to the spot in the file system where RadASM will create the project folder. You can type the path in directly, or you can press the browse button to the right of this text entry box and use a Windows dialog box to select the subdirectory that will hold the project s folder.

The Template: text entry box and browse button lets you select a template for your project. If you don t select a template, then RadASM will create an empty project for you (i.e., the main .hla file will be empty). If you select one of the templates (e.g., the .tpl files found in the *RadASM\Hla\Templates* directory) then RadASM will create a skeletal project based on the project template you ve chosen. For example, the WPA\RadASM directory on the accompanying CD-ROM contains a *win32app.tpl file* that you can select. This automatically supplies a skeletal Win32 application from which you can build your Windows programs.

Once you ve selected the assembler type, project type, entered the project name and description, and optionally selected the folder and a template, press the Next> button to move on to the next window of the Project Wizard dialog. This dialog box appears in Figure 4-17. In this dialog box you select the initial set of files and folders that RadASM will create in the project s folder for you. At the very least, you re going to want a .hla file and a Tmp subdirectory. It s probably a good idea to create a BAK subdirectory as well (RadASM will maintain backup files in that subdirectory, if it is present). More complex Windows applications will probably need a header file (.HHF) and a resource file (.RC) as well. If you re creating a dynamically linked library (DLL), you ll probably want a definition file (.DEF) as well. If you plan on writing documentation, you might want to create a DOC subdirectory - the choice is yours. Check the files and folders you want to create and press the Next > button in the dialog box.

Figure 4-17: Project Wizard Dialog Box #2



The last dialog box of the Project Wizard lets you specify the options present in the Make menu and the commands each of these options executes (see Figure 4-18). RadASM initializes each of the items appearing in this dialog box from the values appearing in the *hla.ini* file. For the most part, you shouldn t have to change any of these options. RadASM properly initializes each item based upon the project type you ve chosen (Windows App, Console App, Dialog App, DLL, etc.). However, it is quite possible to customize how RadASM works on a project by project basis. This dialog box is where you do that customization. Because we ll be using the makefile scheme to build executables, most of the customization you ll do will actually take place in the *makefile*, not in the RadASM Project Wizard Dialog. Therefore, most of the time you ll just press the Finish > button to conclude the construction your new RadASM/HLA project. Figure 4-19 shows what the RadASM window looks like after create a sample Windows App application based on the *win32app* template found on the CD-ROM (this project was given the name MyApp).

Figure 4-18: Project Wizard Dialog Box #3

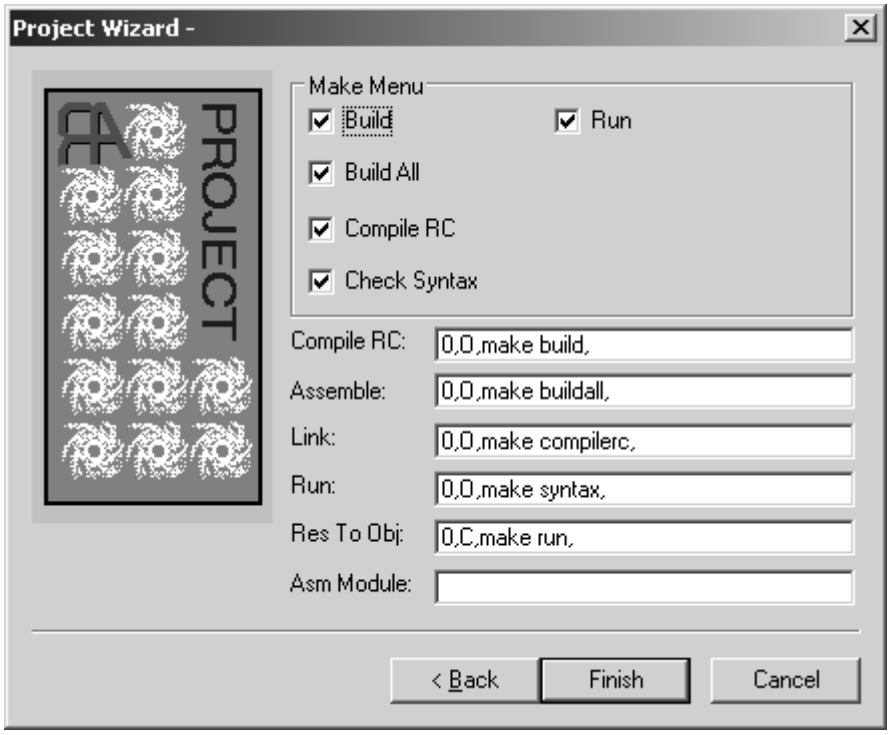
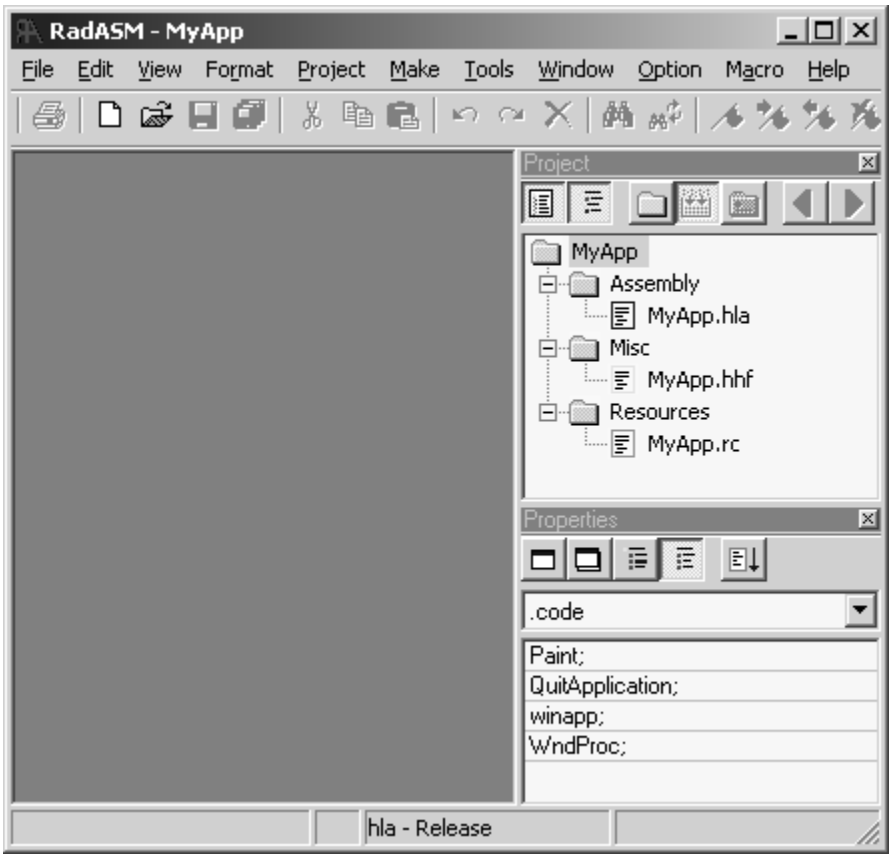


Figure 4-19: Typical RadASM Window After Project Creation



4.5.3: Working With RadASM Projects

Of course, once you ve created a RadASM project, you can open up that project and continue work on it at some later point. RadASM saves all the project information in a .rap (RadAsm Project) file. This .rap file keeps track of all the files associated with the project, project-specific options, and so on. These project files are actually text files, you can load them into a text editor (e.g., RadASM s editor) if you want to see their format. As a general rule, however, you should not modify this file directly. Instead, let RadASM take care of this file s maintenance.

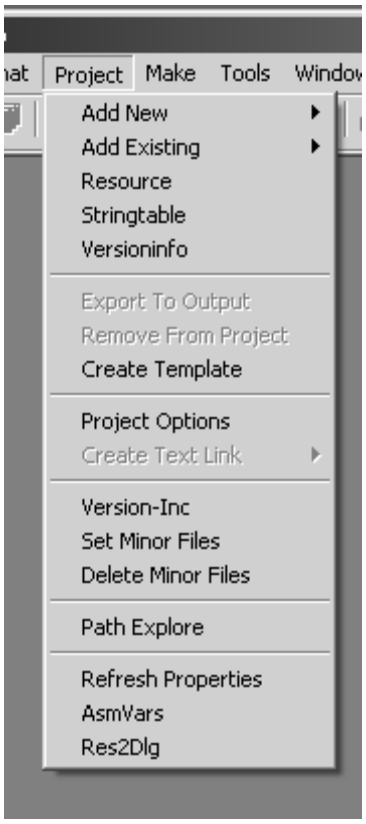
There are several ways to open an existing RadASM project file - you can double-click on the file s icon within Windows and RadASM will begin running and automatically load that project. Another way to open a RadASM project is to select the File>Open Project menu item and open some .rap file via this open command. A third way to open a RadASM project is to use the File Browser to find a .rap file in one of your project directories and double-click on the project file s icon (the .rap file) that appears in the project browser. Any one of these schemes will open the project file you ve specified.

RadASM only allows one open project at a time. If you have a currently open project and you open a second project, RadASM will first close the original project. You can also explicitly close a project, without concurrently opening another project, by selecting the File>Close Project menu item.

Once you ve opened a RadASM project, RadASM s Project menu becomes a lot more interesting. When you create a project, RadASM gives you the option of adding certain stock files to the project (either empty

files, or files with data if you select a template when creating the project). All of the files that RadASM creates bear the project s name (with differing suffixes). As a result, you can only create one .hla file (and likewise, only one .hhf file, only one .rc file, etc.). For smaller assembly projects, this is all you ll probably need. However, as you begin writing more complex applications, you ll probably want additional assembly source files (.hla files), additional header files (.hhf), and so on. RadASM s Project menu is where you ll handle these tasks (and many others). Figure 4-20 shows the entries that are present in the Project menu.

Figure 4-20: The RadASM Project Menu



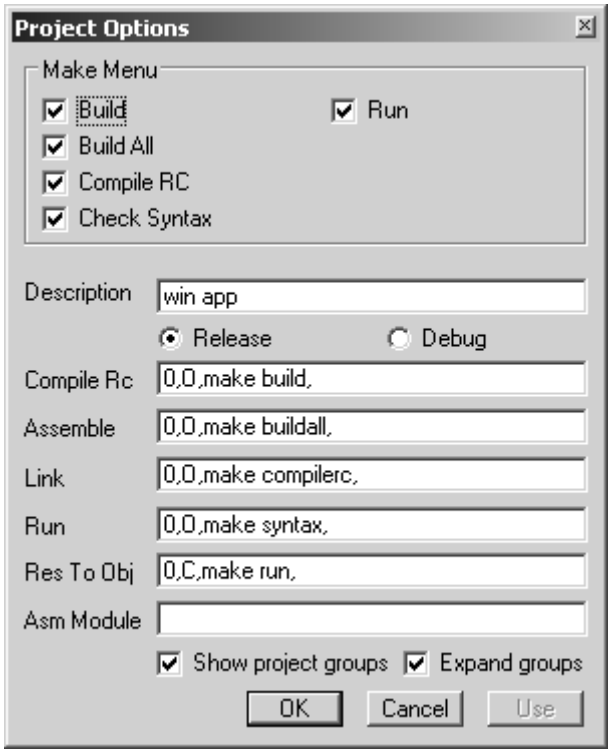
To add new, empty, files to a RadASM project, you use the Project > Add New menu item. This opens up a new submenu that lets you select an assembly file (.hla file), an include file (.hhf), a resource compiler file (.rc), a text file, and so on. Selecting one of these submenu items opens up an Add New File dialog box that lets you specify the filename for the file. Enter the filename and RadASM will create an empty text file with the name you ve specified. Later on, you can edit this source file with RadASM and add whatever text is necessary to that file. Note that RadASM will automatically add that file to the appropriate group based on the file s type (i.e., it s suffix).

The Project > Add Existing sub-menu lets you add a pre-existing file to a project. This is a useful option for creating a RadASM project out of an existing HLA (non-RadASM) project or adding files from some other project (e.g., library routines) into the current project. Note that this option does not create a copy of the files you specify, it simply notes (in the .rad file) that the current project includes that file. To avoid problems, you should make a copy of the actual source file to the current project s folder before adding it to the project; then add the version you ve just copied to your project. It s generally unwise to add the same source file to several different projects. If you change that source file in one project, the changes are automatically reflected in every other project that links this file in. Sometimes this is desirable, but most of the time programmers expect changes to a source file to be localized to the current project. That s why it s always best to make a copy of a source file

when adding that file to a new project. In those cases where you do want the changes reflect to every application that includes the file, it s better to build a library module project and link the resulting .lib file with your project rather than recompile the source file in.

The Project > Project options menu item opens up a Project Options dialog box that lets you modify certain project options (see Figure 4-21). This dialog box lets you change certain options that were set up when you first created the project using the File > New Project Project Wizard dialogs. Most of the items in this dialog box should have been described earlier, but a few of the items do need a short explanation.

Figure 4-21: "Project > Project Options" Dialog Box



The Project Options dialog box provides two radio buttons that let you select whether RadASM will do a debug build or a release build. As you may recall from earlier in this chapter, the choice of release versus debug build controls which set of commands RadASM executes when you select an item from the Make menu. We ll discuss debug builds in a later chapter. Until then, be sure that the Release radio button is selected.

Most of the other options in the Project menu we ll get around to discussing in later chapters as we discuss the Windows features to which they pertain.

4.5.4: Editing HLA Source Files Within RadASM

The RadASM text editor is quite similar to most Windows based text editors you ve used in the past (i.e., RadASM generally adheres to the Microsoft Common User Access (CUA) conventions. So the cursor keys, the mouse, and various control-key combinations (e.g., ctrl-Z, ctrl-X, and ctrl-C) behave exactly as you would expect in a Windows application. Because this is an advanced programming book, this chapter will assume that you ve used a CUA-compliant editor (e.g., Visual Studio) in the past and we ll not waste time discussing mundane things like how to select text, cutting and pasting, and other stuff like that. Instead, this section will concentrate on the novel features you ll find in the RadASM editor.

Of course, the first file navigation aid to consider is the Project Browser pane. We ve already discussed this RadASM feature in earlier sections of this chapter, but it s worth repeating that the Project Browser pane lets you quickly switch between the files you re editing in a RadASM project. Just double-click on the icon of the file you want to edit and that file will appear in the RadASM editor window pane.

Immediately below the Project Browser pane is the Properties pane (if this pane is not present, you can bring it up by selecting View > Properties from the RadASM View menu). This pane contains two main components: a pull down-down menu item that lets you select the information that RadASM displays in the lower half of this window. If not already selected, you should select the .code item from this list. The .code item tells RadASM to list all the sections of code that it recognizes as procedures (or the main program) in an HLA source file (see Figure 4-22).

Figure 4-22: The HLA Properties Window Pane

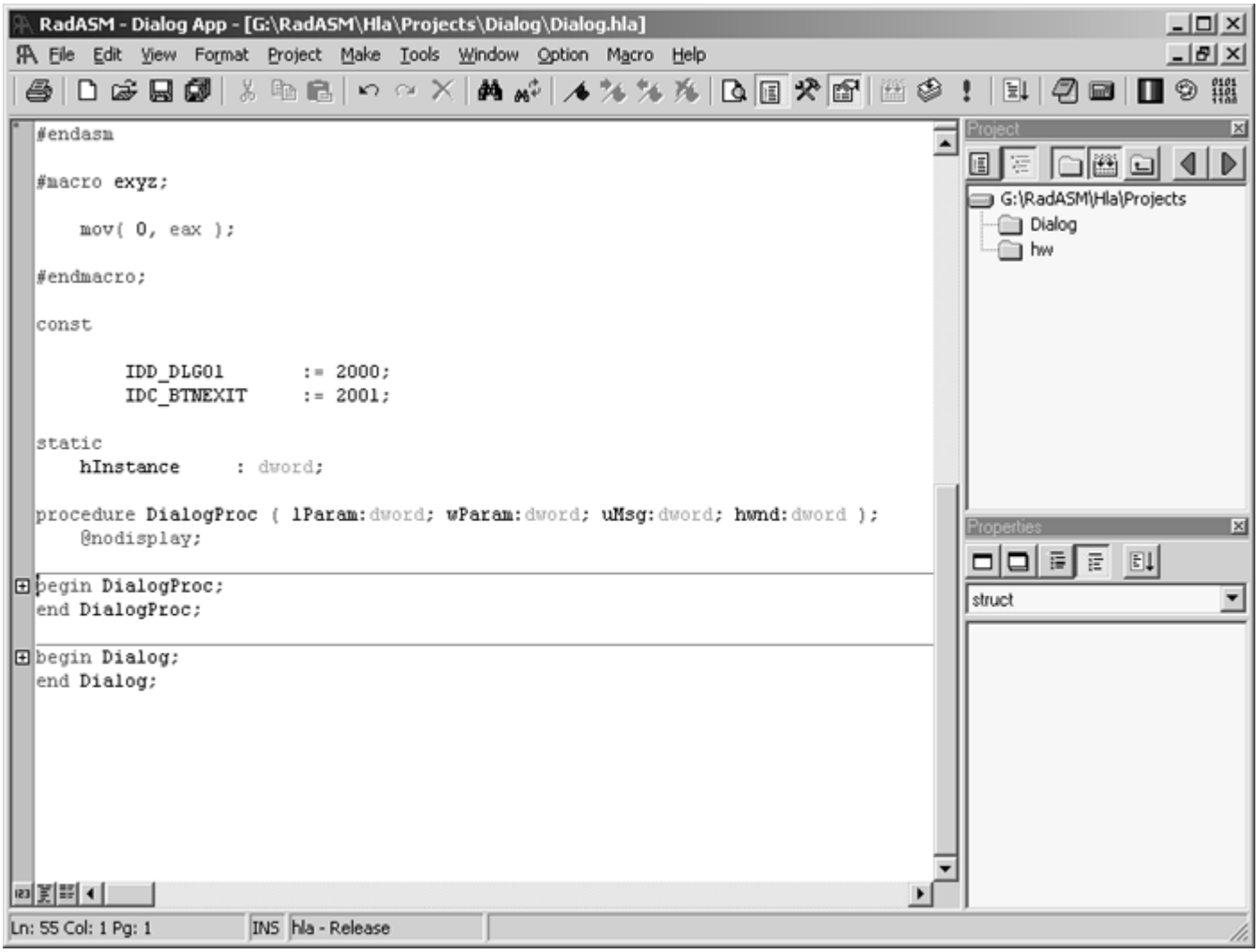
One very useful RadASM feature is that you can quickly jump to the start of a procedure s body (at the *begin* statement) by simply double-clicking on that procedure s name in the Properties Window pane. In the example appearing in Figure 4-22 (this is the Dialog project supplied with RadASM for HLA), double-clicking on the Dialog; and DialogProc; lines in this list box automatically navigates to the start of the code for the selected procedure.

The pull-down menu in the Properties window lets you select the type of objects the assembler provides. For example, by selecting .const you can take a look at constant declarations in HLA. The macro selection lets you view the macro definitions that appear in the source file. As this chapter was first being written, the other property items weren t 100% functional; hopefully by the time you read this RadASM will have additional support for other types of HLA declarations.

Another neat feature that RadASM provides is an outline view of the source file. Looking back at Figure 4-22 you ll notice that *begin DialogProc;* statement has a rectangle with a minus sign in it just to the left of the source code line. Clicking on this box closes up all the code between the *begin* and the corresponding *end* in the source file. Figure 4-23 shows what the source file looks like when the *Dialog* and *DialogProc* procedures are collapsed in outline mode. The neat thing about outline mode is that it lets you view the big picture without out the mind-numbing details of the source code for each procedure in the program. In outline view, you can quickly skim through the source file looking for important code and drill down to a greater level of detail by opening up the code for a procedure you re interested in looking at. You can also rapidly collapse or expand all procedure

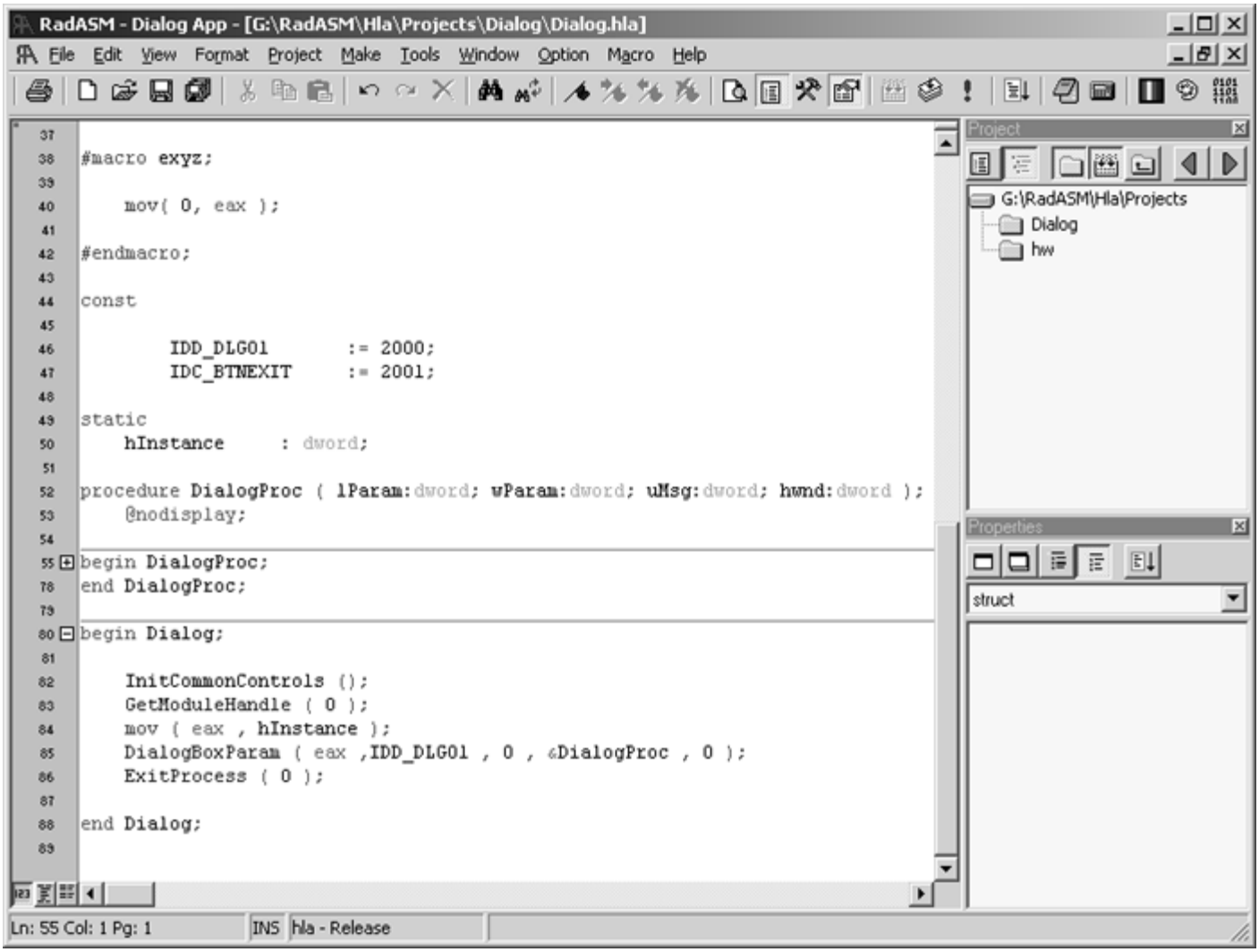
levels by pressing the expand or collapse buttons appearing on the lower left hand corner of the text editor window (see Figure 4-23).

Figure 4-23: RadASM Outline View (with Collapsed Procedures)



Another useful feature RadASM provides is the ability to display line numbers with each line of source code. Pressing on the line number icon in the lower-left hand corner of the text editor window (the icon with the 123 in it) toggles the display of line numbers in the editor s window. See Figure 4-24 to see what the source file looks like with line numbers displayed. The line number display mode is quite useful when searching for a line containing a syntax error (as reported by HLA). Note that you can also navigate to a given line number by pressing ctrl-G and entering the line number (you can also select Edit > Goto line from the Edit menu).

Figure 4-24: Displaying Line Numbers in RadASM's Editor



Another useful navigation feature in RadASM is support for *bookmarks*. A bookmark is just a point in the source file that you can mark. You can create a bookmark by selecting a line of text (by clicking the mouse on the gray bar next to the line) and selecting Edit > Toggle BookMark or by pressing shift-F8. You can navigate between the bookmarks by pressing F8 or ctrl-F8 (these move to the next or previous bookmarks in the source file). RadASM (by default) provides several icons on it s toolbar to toggle bookmarks, navigate to the previous or next bookmark, or clear all the bookmarks. Which method (edit menu, function keys, or toolbar) is most convenient simply depends on where your hands and the mouse cursor currently sits.

The RadASM Format menu also provides some useful features for editing HLA programs. The Format > Indent and Format > Outdent items (also accessible by pressing F9 and ctrl-F9) move a selected block of text in or out four spaces (so you can indent text between an `if` and `endif`, for example). You can also convert tabs in a document to spaces (or vice versa) from the Format > Convert > Spaces To Tab and Format > Convert > Tab To Spaces menu selections.

You ll notice that RadASM provides syntax coloring in the editor window (that is, it sets the text color for various classes of reserved words and symbols to different colors, making them easy to identify with a quick

glance in the editor window). The *hla.ini* file appearing on the CD-ROM accompanying this book contains a set of reasonable color definitions for HLA s different reserved word types. However, if you don t particularly agree with this color scheme, it s really easy to change the colors that RadASM uses for syntax highlighting. Just select the Options > Colors & Keywords menu item and select an item from the Syntax/Group list box (Figure 4-25 shows what this dialog box looks like with the Group #00 item selected). By double-clicking on an item within the Group list box, you can change the color for all the items in that particular group (e.g., see Figure 4-26). RadASM automatically updates the *hla.ini* file to remember your choice of colors the next time you run RadASM.

Figure 4-25: Option>Colors & Keywords Dialog Box with Group#00 Selected

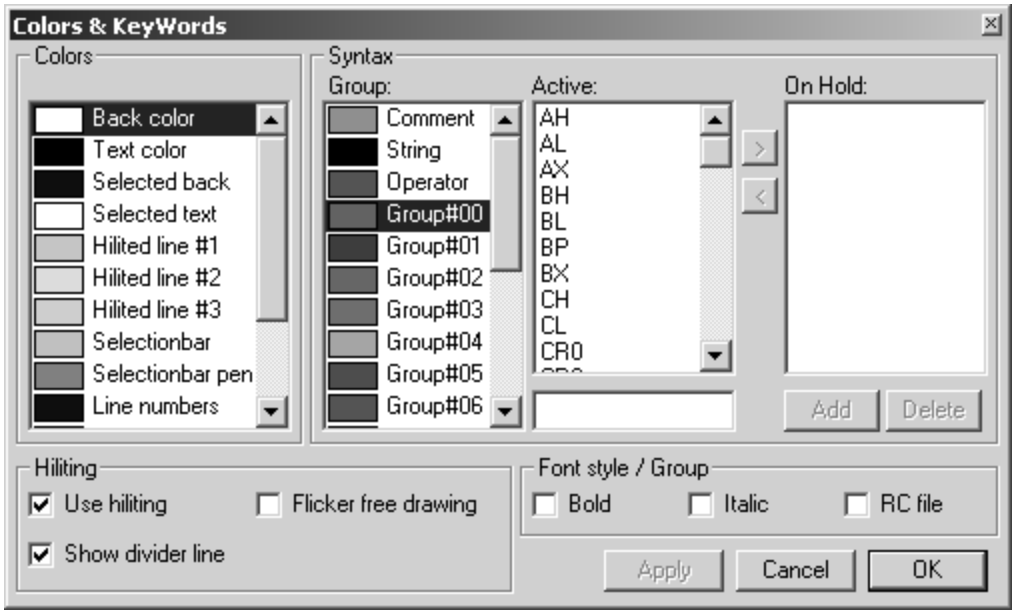
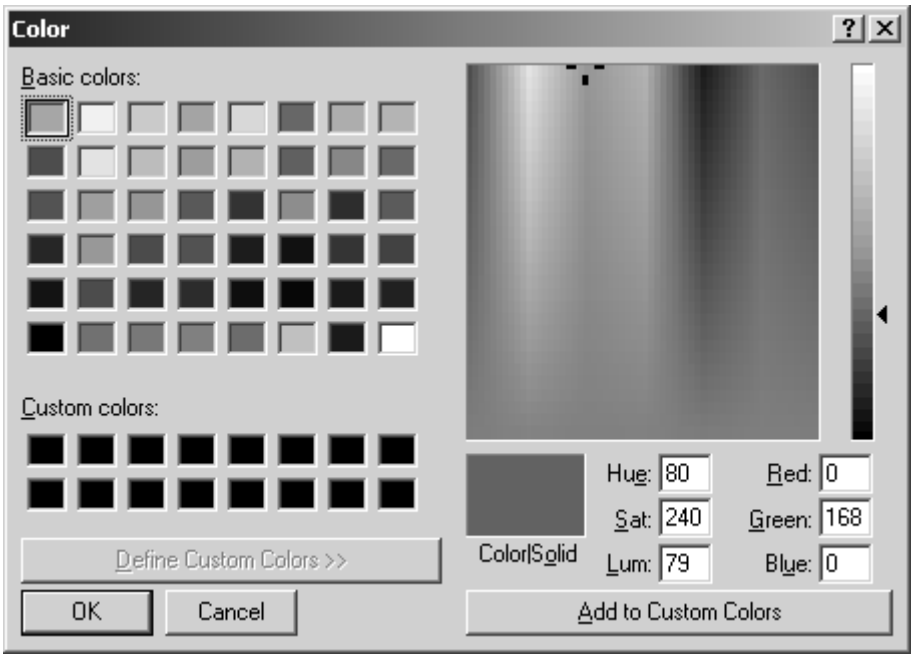


Figure 4-26: Color Selection Dialog Box



You can also set the display fonts to something you re happier with if the default font (Courier New, typically) isn t to your liking. This is also achievable from the RadASM Option menu.

4.6: Creating and Compiling HLA Projects With RadASM

There are two ways to create a new RadASM/HLA project - from scratch or via a *template*. Creating a new project from scratch is great when you ve got some HLA (or other) files that you want to incorporate into a new project; using a template is a great solution when you want to begin a brand-new project in RadASM. We ll explore both of these approaches here.

To create a new project, either from scratch or from a template, you select the File > New Project menu item. This brings up the Project Wizard dialog box (see Figure 4-16). By default, assuming you ve set up your *radasm.ini* file properly, the HLA assembler should be selected. If not, select that assembler from the Assembler pull down menu (and, by all means, fix your *radasm.ini* file and make HLA the default assembler, as described earlier in this chapter).

Next, select the type of project you ll want to create. For the most part, we ll want to create Windows Applications so you d normally select the Windows App radio button in the Project Type group box. However, because we ve not discussed how to write a Windows application yet, you should select Console App for the time being.

The next step is to give this project a name by typing a valid file name into the Project Name: text edit box. This must be a valid Windows file name without a path prefix and without any suffix (RadASM will supply appropriate suffixes). In the examples that follow, this chapter will use the name myconsoleProject for the project name. If you supply a different name, RadASM will use that name in place of myconsoleProject but everything will work as you expect. For the sake of example, you should use myconsoleProject for this exercise and then repeat this process with a different base filename later, just to see how RadASM works.

After entering the project s name, type an English (or other natural language) description of the project into the Project Description: text edit box. RadASM will display this description in the Project pane when you open the project.

By default, RadASM assumes that you want to create a new project directory in the ....*RadASM\Hla\Projects* subdirectory. If this is fine, you can ignore the Projects Folder: box. If you want to place the project folder somewhere else, then click on the browse button to the right of the Projects Folder: text entry box and select the folder where you want RadASM to put your project directory (you can also type the path in if you prefer). Note that you do not include the project s directory name in this path; RadASM will create the project folder in the subdirectory you specify.

The last entry in the (first) Project Wizard dialog box is the Template: entry. You ll specify a template name here if you want to create a new project based on an existing template. Because we re starting from scratch at this point, we ll go ahead and select a template. To do this, press the Browse button that appears just to the right of the Template: text entry box. This opens up the Template Selection Dialog box (see Figure 4-27). For the time being, select the consApp.tpl template (which is a template for a Windows console application). Note that the consApp.tpl template is only available if you ve copied the template files from the *WPA\RadASM\Templates* subdirectory to your *RadASM\Hla\Templates* subdirectory. The consApp.tpl template file is not a standard part

of the RadASM distribution. Once you ve selected the consApp.tpl template, the Project Wizard dialog box should look something like that appearing in

Figure 4-27: Template Selection Dialog Box

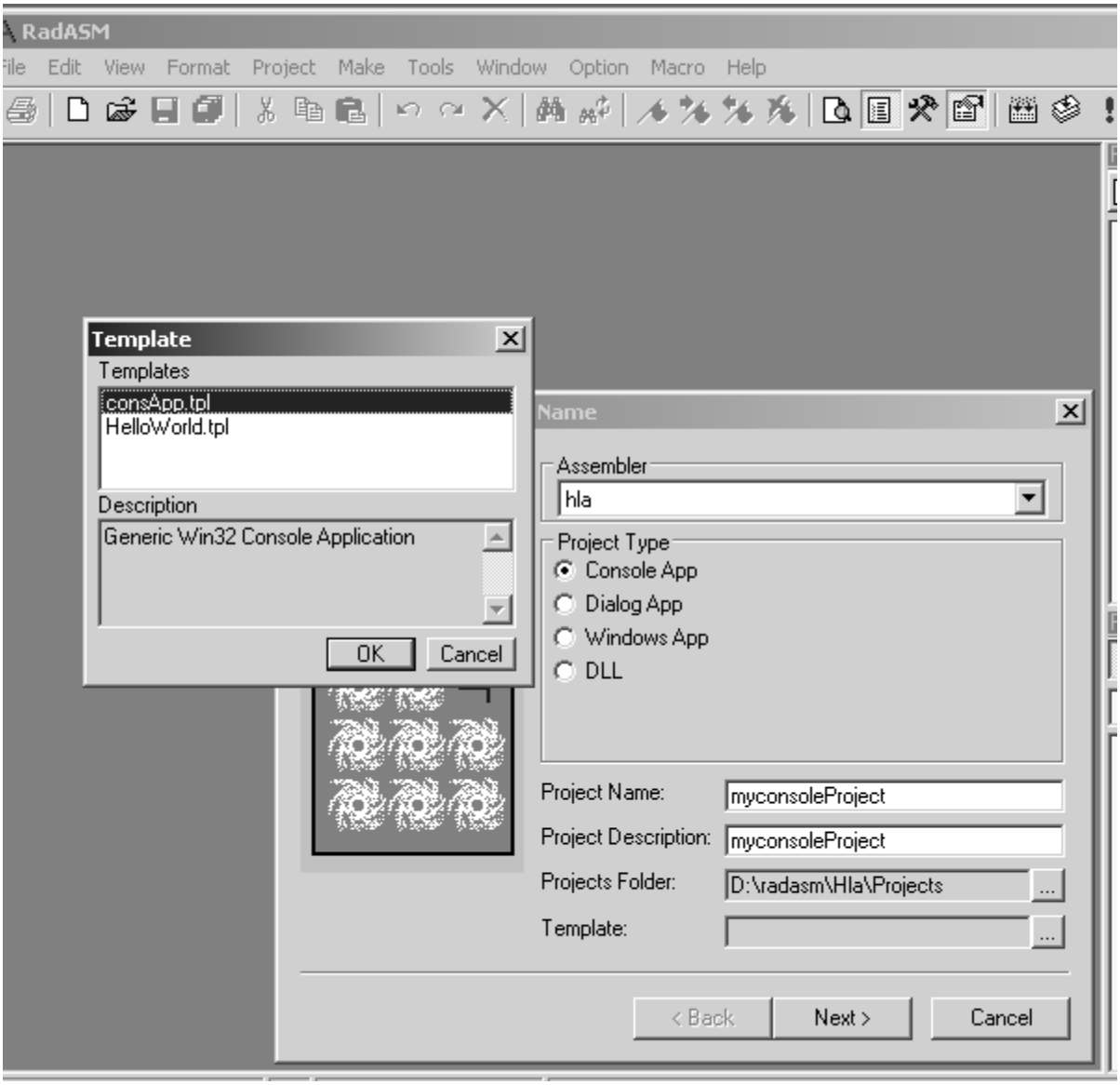
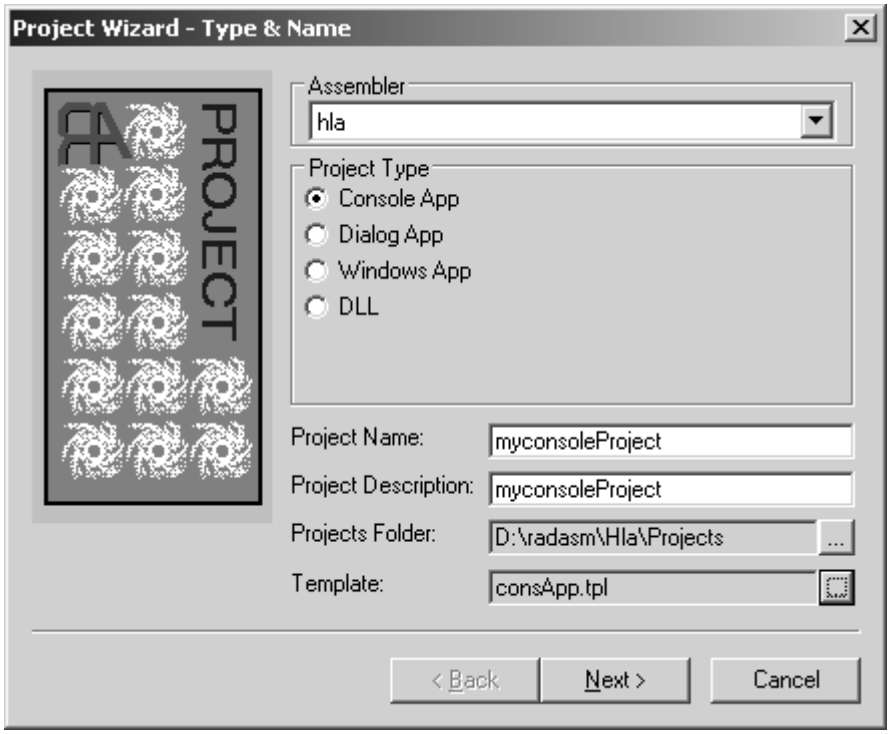


Figure 4-28: Project Wizard Dialog Box - consApp.tpl Template

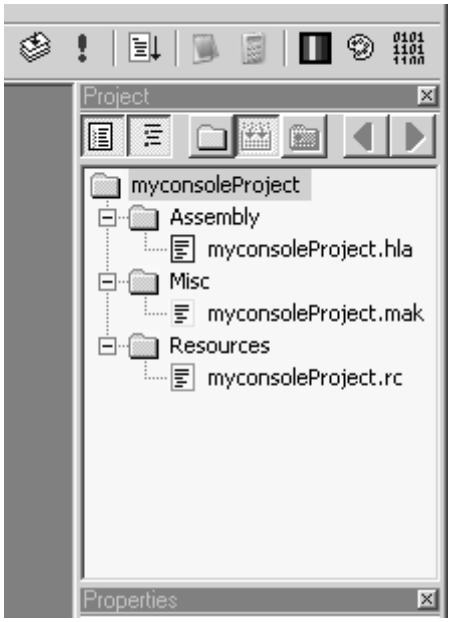


At this point, press the Next button to continue with the project creation. This brings up the second Project Wizard dialog box (see Figure 4-17). We only need an HLA file and BAK and TMP subdirectories, so make sure these checkboxes contain checks and all the other checkboxes are empty. Once you ve done this, press the Next button to continue.

At this point you should be looking at the third Project Wizard dialog box (see Figure 4-18). Because we re not working with any .rc files in this application, you can uncheck the check box next to Compile RC . All the other checkboxes should be checked and the text entry boxes should all be filled in except for Asm Mod-

ule: . At this point, press the Finish button to complete the construction of the project. Figure 4-29 shows you what the RadASM project pane looks like after you ve completed the construction of this project.

Figure 4-29: RadASM Project Pane After Creating a Console Project From a Template



The first thing you ve got to do is edit the makefile that the template has created and save the result as make-file . Note (in Figure 4-29) that the template created a file called myconsoleProject.mak . Because make really prefers you to name this file makefile , plus the fact that the generic makefile needs some editing, our first step is to edit this file. So double-click on the *myconsoleProject.mak* file icon to bring this file up into the editor.

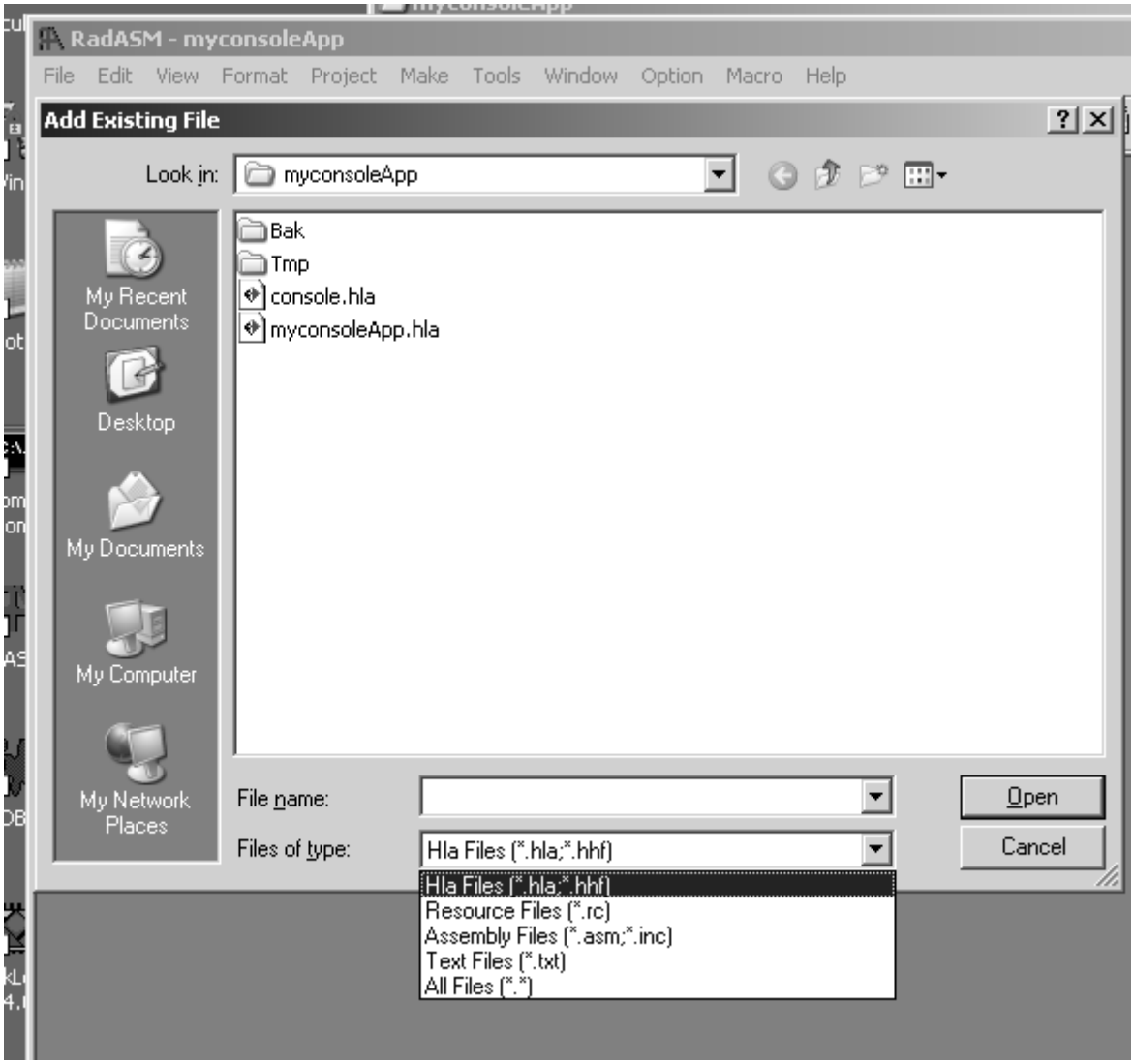
Throughout the makefile, you ll see filenames of the form <<file>> . You need to replace these stubs with the name of your project (myconsoleApp in this particular case). So press ctrl-R to enter search and replace mode and enter <<file>> for the Find What: string and myconsoleApp for the replace: string. Hit the Replace All button to quickly replace each occurrence of <<file>> by myconsoleApp throughout the makefile.

The next step is to save the file. Select File > Save to save the changes over the top of *myconsoleApp.mak*. Note that we also need to save this as *makefile* in the current project directory in order for RadASM s make menu to work properly. Select the File > Save File As menu item and browse the *myconsoleApp* project directory (unless you ve specified a different path, this will be in the *.....\RadASM\Hla\Projects* folder). Type makefile. into the File Name: text edit box. **Note the period at the end of makefile. !** This is important. Without it, RadASM will automatically tack a .hla suffix to the end of the filename, which is not what we want. Also, be sure to select All Files from the Save As Type: pull-down menu. Again, unless you select this, RadASM will add a suffix to the end of the filename, which is not something we want to happen.

To clean things up a little bit, let s remove *myconsoleApp.mak* from the project and add *makefile* to the project (to avoid confusion). To remove *myconsoleApp.mak* from the project, right-click on the *myconsoleApp.mak* icon in the Project window pane and select Remove From Project in the pop-up menu that appears. To add makefile to the project, select the Project > Add Existing > File menu item. This brings up the Add Existing File dialog box (seeFigure 4-30). Note that *makefile* doesn t appear in this list. This is because RadASM is applying file filtering here and is only displaying .hla and .hhf files. To see the make file, select All Files from the Files of type: pull down menu. Once you ve done this (and assuming you ve saved *make-file* into the project s directory), you should be able to select *makefile* and then press the Open button to add it

to the project. RadASM adds the *makefile* to the Misc file group. Don t get the impression that it s unnecessary to add the *makefile* to your project. Through RadASM will find the *makefile* just fine when building the application (assuming the *makefile* is in your project directory), as your projects become more complex you ll need to edit the *makefile* whenever you add new files to the project. Therefore, it s wise to make the *makefile* a part of the RadASM project so that it s easy to edit.

Figure 4-30: Add Existing File Dialog Box

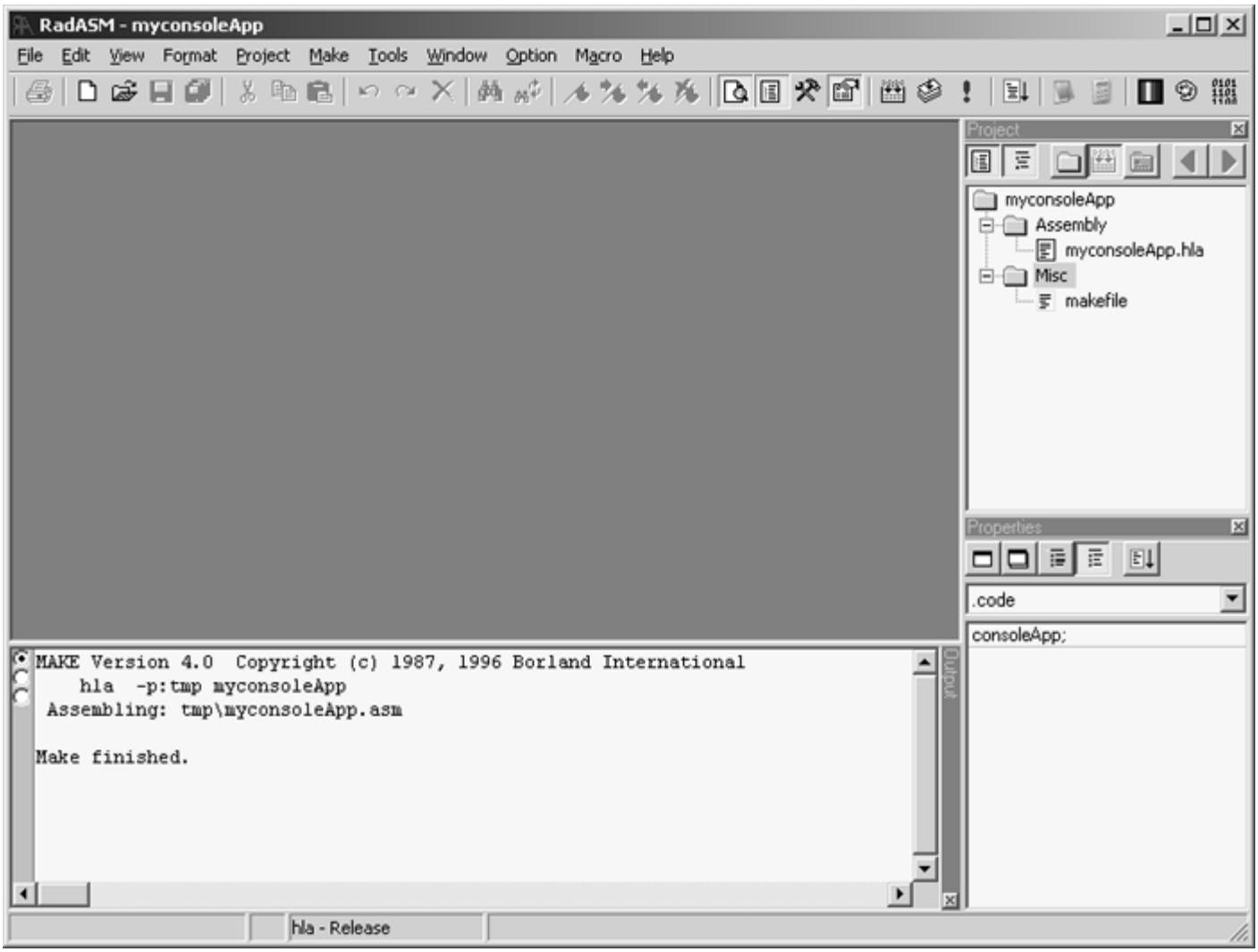


Once you ve edited the makefile and added it to the project, you should be able to build the application by selecting the RadASM Make > Build menu item. This should produce output in the RadASM output window like that appearing in Figure 4-31. Of course, the template we ve copied does produce a useful program (it s just an empty HLA program that simply returns to Windows). To make this program do something useful, click on the *myconsoleApp.hla* icon in the Project window pane and edit the file. Add the following two lines to this program:

```
stdout.put( "A Console App" nl );
stdin.readLn();
```

Save the file (by selecting the File > Save menu item or by pressing the little disk icon on the toolbar). Then build the application by selecting the Make > Build All menu item. Finally, run the application by selecting the Make > Run menu item. When the application runs, press the Enter key to close the window that opens up. Congratulations! You ve just created, edited, and run your first application from within RadASM.

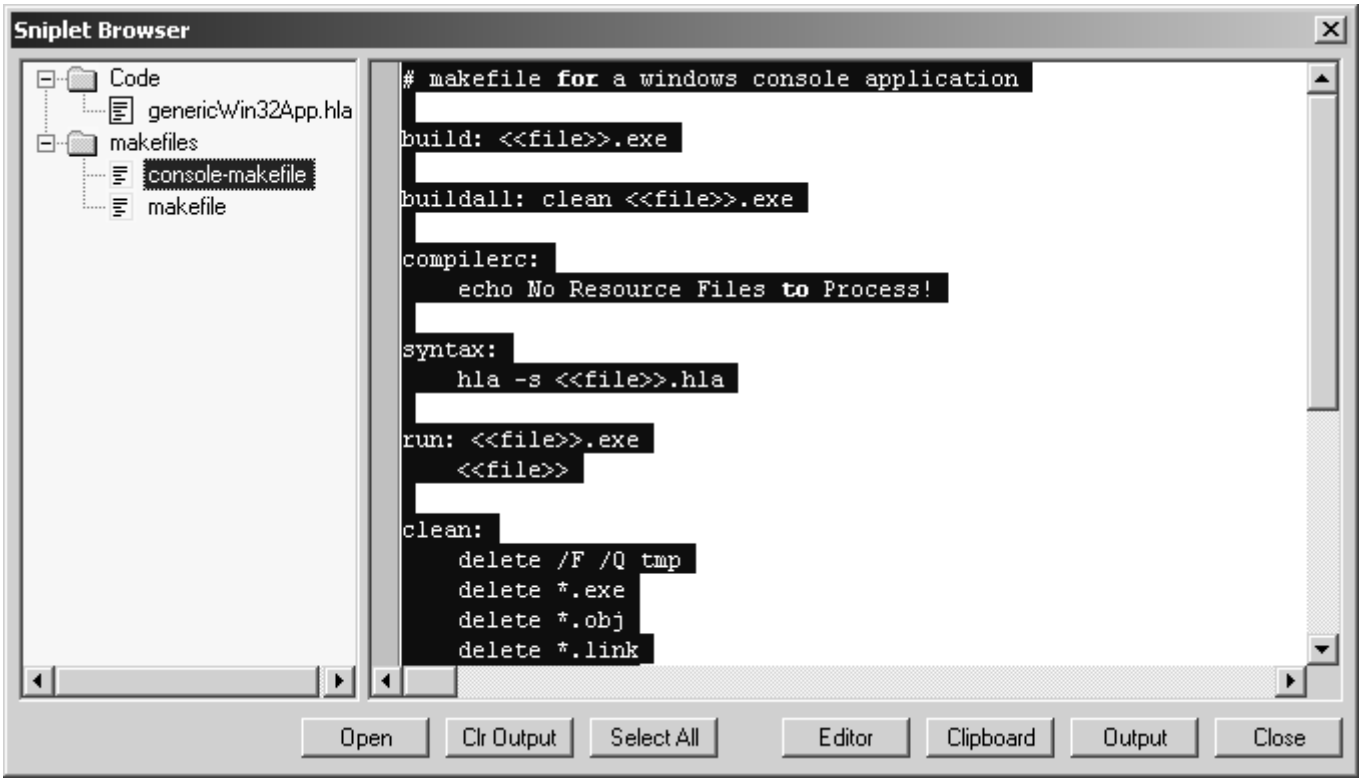
Figure 4-31: Building myconsoleApp



Creating a new project from scratch follows this same process, except you don t select a template file in the first dialog box of the Project Wizard. Rather than create HLA and make files that contain generic information, this process will create empty files (based on your selections in the second Project Wizard dialog box, see Figure 4-17). You ll edit these empty files to create the programs you want. If you want to add new files to the project, you can do so by selecting Project > Add New > File and specifying the filename. For example, creating a new project without a template does not create a *makefile* for you, so it s probably a good idea to create a *makefile* right away for this project. Rather than type in the entire makefile from scratch, you can take advantage of another neat RadASM feature - *sniplets*. A sniplet is simply a small text file containing a short sequence of code (or other data) that you often use. You can open up a sniplet and cut and paste it into a file in your current project. For example, if your makefile is currently open for editing, Select the Tools > Sniplets menu item. In the left-most pane of the window that opens up, you see a selection of files you can copy data from. Select the console-makefile item and this will display the contents of that sniplet file in the right-most pane of the sniplet window

(see Figure 4-32). Press the Select All button and then the clipboard button to copy all the selected text to the clipboard and then press the Close button to close the Sniplets Browser window. Now select Edit > Paste to paste this text into your *makefile* (which, presumably, is the open file in the RadASM editor window). Of course, this is a generic console makefile, requiring the same editing as before, so make those changes and then save your *makefile* to disk.

Figure 4-32: The Sniplet Browser Window



4.7: Developing Small Projects with RadASM

RadASM, by itself, supports the ability to drive HLA. If you select the Project > Project Options menu item and edit the make commands (or edit these corresponding commands in the *hla.ini* file), you can tell RadASM to call HLA directly to assemble your files. The only problem with this approach is that RadASM will assemble all files rather than just the ones that require changing. For small HLA-based applications, recompiling all the source files on each build is no big deal. But as the project gets larger and more complex, the compile times will start to become annoying if you recompile the entire application on every build. As you ve seen, by using a make file, you can recompile only those modules that need to be recompiled. This is one of the primary reasons for running make from RadASM rather than running HLA directly. Another advantage of having the make files is that it s easy to recompile from the command line (e.g., by typing make build) and produce the same results you get inside RadASM.

However, if you re developing small Windows applications, you may find the use of make files a bit too complex for your tastes. In particular, RadASM can automatically maintain projects for you so that you don t have to edit a make file every time you want to add or remove a file from the project. This can solve some consistency problems you might run into if you re not careful about properly maintaining your make files. This book assumes the use of, and always provides, make files. However, for your own projects it s quite possible to edit

the *hla.ini* file to run HLA directly rather than run make to run HLA. This book will not spend any more time describing how to do this (though this chapter does provide the basic information you ll need); please see the RadASM documentation for details on this technique.

4.8: Plus More!

This chapter barely begins to scratch the surface with respect to the features that RadASM supports. Some things (like Rapid Application Development features) will have to wait for a later chapter when we ve covered some Windows programming. Other features are simply to numerous to describe here. Your best bet is to sit down with RadASM and play around with it. Note that some features are not applicable to HLA users, so don t be disappointed if some feature doesn t work for you. If you ve got questions about features in RadASM or you re unsure how to do something, you should

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.

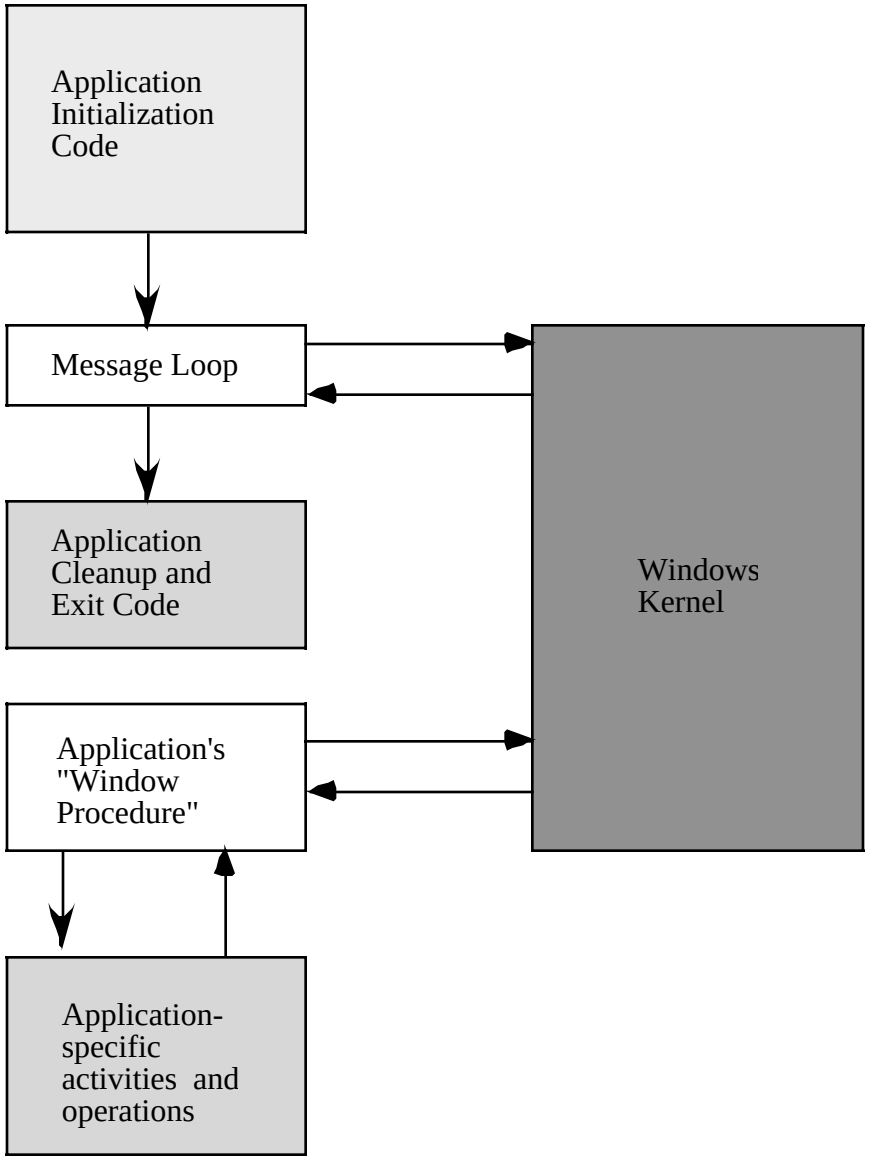
> If you've ever written a 16-bit DOS application in assembly language, you've done some message passing. The INT 21h instruction that "calls" DOS is equivalent to calling DOS' "window procedure". The values you pass in the 80x86 registers correspond to the message and, in particular, the value in the 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



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 appli-

cation 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.

Chapter 6: Text in a GUI World

6.1: Text Display Under Windows

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

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

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

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

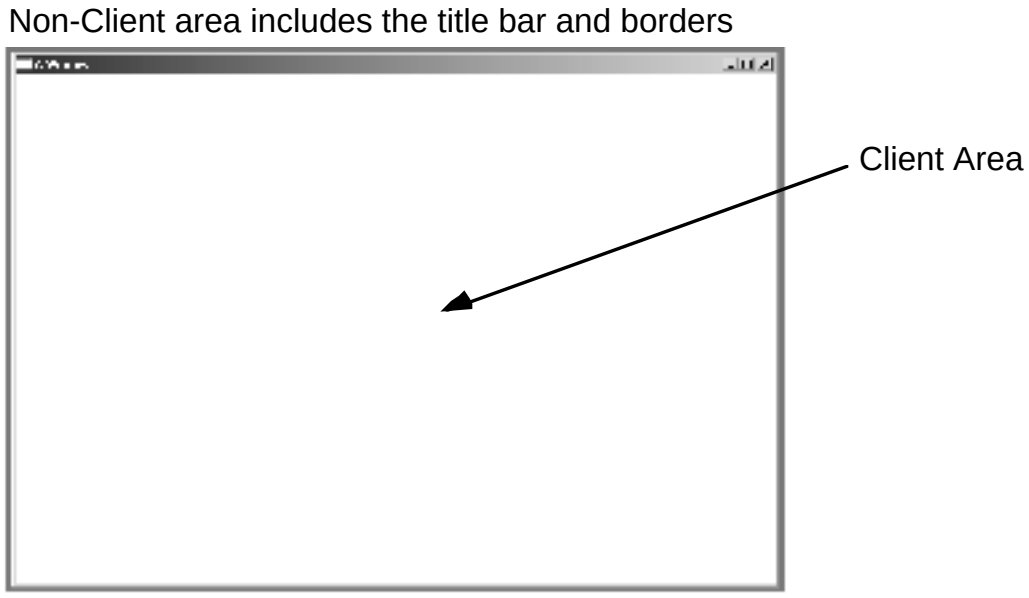
Because of the simplifying nature of text output in a console app, writing GUI apps is going to involve more complexity than writing console apps. The purpose of this chapter is to explain that additional complexity.

6.2: Painting

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

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

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



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

Calling the `w.UpdateWindow` function isn t the only way that the contents of some window could become invalid. An application s window could become invalid because the user drags (or opens) some other window over the top of the window and then closes that other window. Because the area covered by the second window in the original application s window now contains the image drawn for that second window, the client area of the original window is *invalid* - it does not contain a valid image for the original window. When this happens,

Microsoft Windows will invalidate the original window and post a *w.WM_PAINT* message to that window so it will redraw itself and repair the damage caused by overlaying that second window over the first.

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

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

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

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

6.2.1: Device Contexts

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.2.2: Device Context Attributes

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

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

6.2.3: Painting Text in the Client Area

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

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

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

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

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

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

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

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

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

Of course, the standard way to dynamically allocate storage for a string in HLA is via the `stralloc` function. The only problem with `stralloc` is that it ultimately winds up calling Windows memory management API. While there is nothing fundamentally wrong with doing this, keep in mind two facts: calling Win32 API functions are somewhat expensive (and the memory management calls can be expensive) and we re briefly allocating

storage for this string. Once we ve converted our output to string form and displayed it in the window, we don t really need the string data anymore. True, we can use the same string variable for several output operations, but the bottom line is that when our Paint procedure (or whomever is handling the w.WM_PAINT message) returns, that string data is no longer needed and we ll have to deallocate the storage (i.e., another expensive call to the Win32 API via the *strfree* invocation). A more efficient solution is to allocate the string object as local storage within the activation record of the procedure that contains the call to *str.put*. HLA provides two functions to help you do this efficiently: *str.init* and *tstralloc*.

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

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

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

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

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

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

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

```
begin demoTStrAlloc;

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

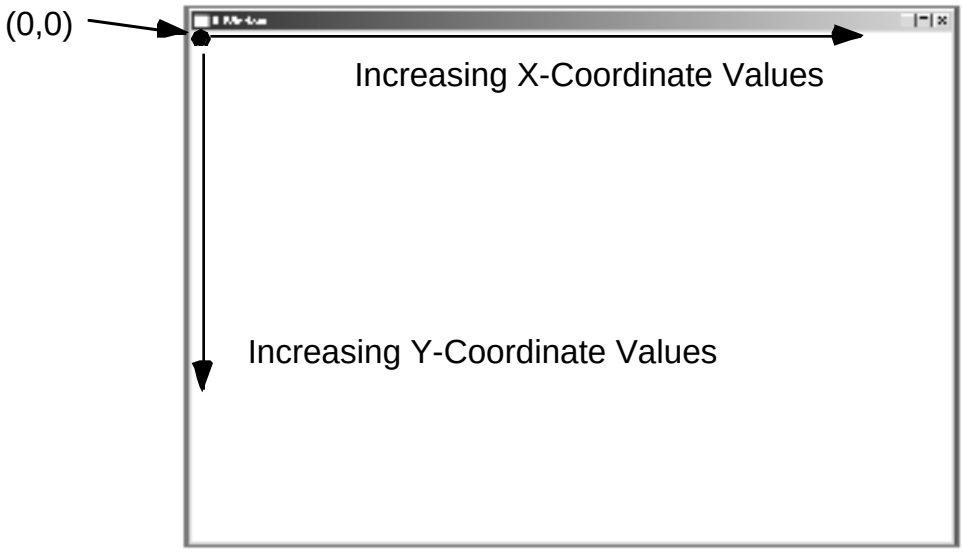
Note that the call to *tstralloc* drops the stack down by as many as 12 bytes beyond the number you specify. In general, you won t know exactly how many bytes by which *tstralloc* will drop the stack. Therefore, it s not a good idea to call this function if you ve got stuff sitting on the stack that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

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

```
static

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

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

Figure 6-2: Windows' Default Coordinate System



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

- w.DT_CENTER: centers the text within the rectangle
- w.DT_EXPANDTABS: expands tab characters within the string
- w.DT_LEFT: left-aligns the string in the rectangle
- w.DT_RIGHT: right-aligns the string in the rectangle
- w.DT_SINGLELINE: only allows a single line of text in the rectangle, turns off the processing of carriage returns and line feeds
- w.DT_VCENTER: vertically centers the text in the rectangle
- w.DT_WORDBREAK: turns on word break - lines are broken on word boundaries when you use this option

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

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

```
static
```

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

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

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

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

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

The *hDC, X, Y, lpString*, and *nCount* parameters have the same meaning as the *w.TextOut* parameters (*nCount* is the number of characters in *lpString*). The *nTabPositions* parameter specifies the number of tab positions appearing in the *lpnTabStopPositions* (array) parameter. The *lpnTabStopPositions* parameter is a pointer to the first element of an array of 32-bit unsigned integer values. This array should contain at least *nTabPositions* elements. The array contains the number of pixels to skip to (relative to the value of the last parameter) for each tab stop that *w.TabbedTextOut* encounters in the string. For example, if *nTabPositions* contains 4

and the *lpnTabStopPositions* array contains 10, 30, 40, and 60, then the tab positions will be at pixels *nTabOrigin*+10, *nTabOrigin*+30, *nTabOrigin*+40, and *nTabOrigin*+60. As this discussion suggests, the last parameter, *nTabOrigin*, specifies the pixel position from the start of the string where the tab positions begin.

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

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

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

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

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

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


readonly

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

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

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

    MsgProcPtr_t:
        record

            MessageValue:   dword;
            MessageHndlr:   MsgProc_t;

        endrecord;



// The dispatch table:
//
```

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

readonly

    Dispatch:   MsgProcPtr_t; @nostorage;

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

            // Insert new message handler records here.

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



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

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


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

    w.PostQuitMessage( 0 );

end QuitApplication;


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

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

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

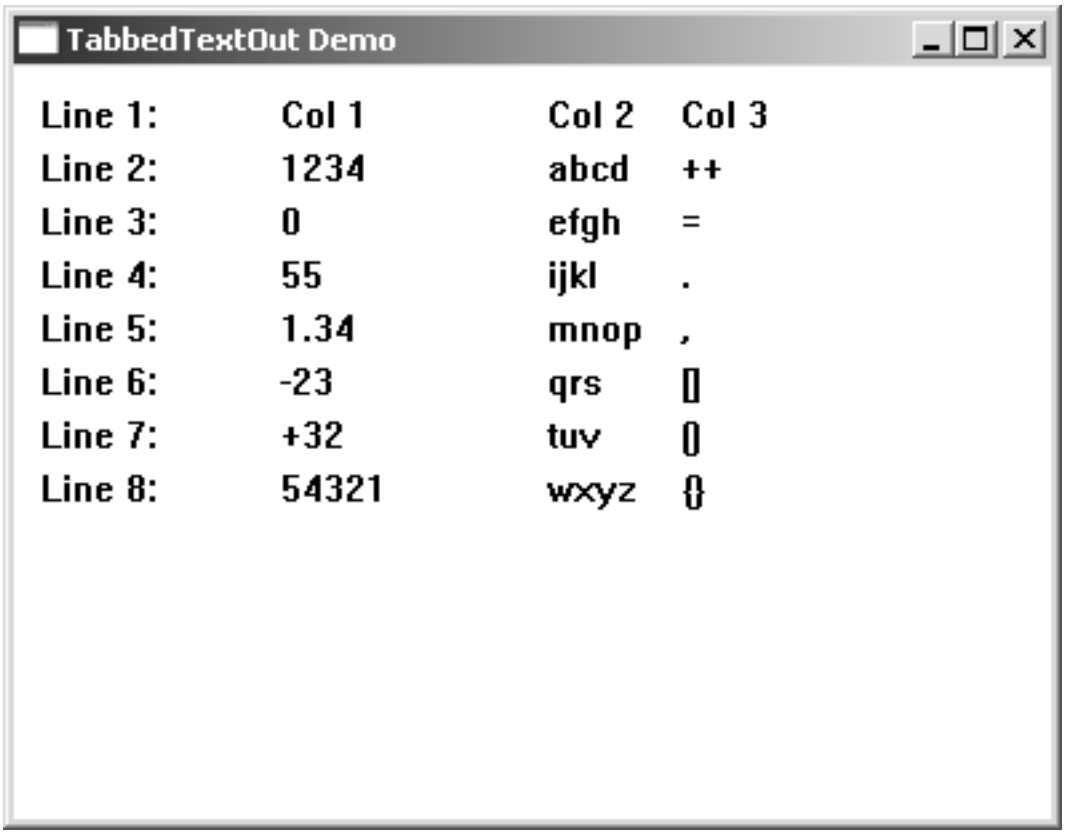
```
readonly
    TabStops        :dword[  NumTabs ]  := [ 50, 100, 200, 250];
    TextToDisplay   :string[  LinesOfText ]  :=
        [
            "Line 1:" tab "Col 1" tab "Col 2" tab "Col 3",
            "Line 2:" tab "1234"  tab "abcd"  tab "++",
            "Line 3:" tab "0"     tab "efgh"  tab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

        ];


begin Paint;

    push( ebx );

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

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

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

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

    endfor;


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

end Paint;

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

```
// The window procedure.  Since this gets called directly from
// windows we need to explicitly reverse the parameters (compared
// to the standard STDCALL declaration) in order to make HLA's
// Pascal calling convention compatible with Windows.
//
// This is actually a function that returns a return result in
// EAX.  If this function returns zero in EAX, then the event
// loop terminates program execution.

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

begin WndProc;

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

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

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

            // If an unhandled message comes along,
            // let the default window handler process the
            // message.  Whatever (non-zero) value this function
            // returns is the return result passed on to the
            // event loop.

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


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

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

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

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

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

    endfor;

end WndProc;



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

begin TabbedText;


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

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

    // Get this process' handle:

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

    // Get the icons and cursor for this application:

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

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

    // Okay, register this window with Windows so it
    // will start passing messages our way.  Once this
    // is accomplished, create the window and display it.

    w.RegisterClassEx( wc );

    w.CreateWindowEx
    (
        NULL,
        ClassName,
        AppCaption,
        w.WS_OVERLAPPEDWINDOW,
        w.CW_USEDEFAULT,
        w.CW_USEDEFAULT,
        w.CW_USEDEFAULT,
        w.CW_USEDEFAULT,
        NULL,
```

```
        NULL,
        hInstance,
        NULL
    );
    mov( eax, hwnd );

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

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

    forever

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

    endfor;

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

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

end TabbedText;
```

Figure 6-3: TabbedText.HLA Output



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

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

The *hdc, x, y, lpString*, and *cbCount* parameters are compatible to the parameters you supply to the *w.TextOut* function. The *fuOptions* parameter specifies how to use the application-defined *lprc* rectangle. The possible options include clipping the text to the rectangle, specifying an opaque background using the rectangle,

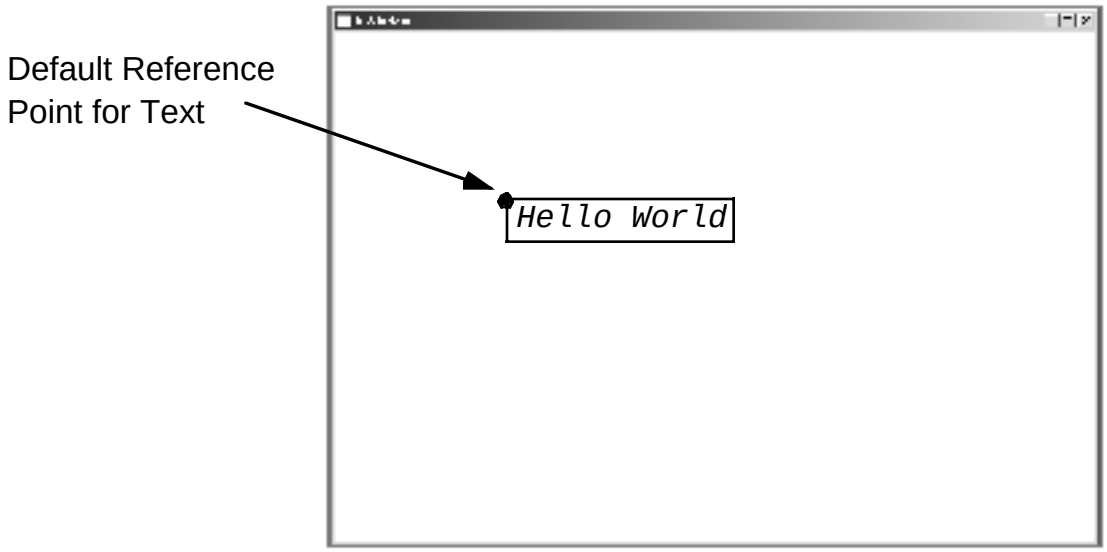
and certain non-western alphabet options. Please consult the GDI32 documentation on the accompanying CD-ROM for more details concerning the *fuOptions* and *lprc* parameters. The *lpDx* array is option. If you pass NULL as this parameter s value, then the *w.ExtTextOut* function will use the default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

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

- w.TA_BASELINE - The reference point will be on the base line of the text
- w.TA_BOTTOM - The reference point will be on the bottom line of the bounding rectangle surrounding the text
- w.TA_TOP - The reference point will be on the top line of the bounding rectangle surrounding the text
- w.TA_CENTER - The reference point will be aligned with the horizontal center of the bounding rectangle surrounding the text
- w.TA_LEFT - The reference point will be on the left edge of the bounding rectangle surrounding the text
- w.TA_RIGHT - The reference point will be on the right edge of the bounding rectangle surrounding the text
- w.TA_NOUPDATECP - The current position is not updated after outputting text to the window
- w.TA_RTLREADING - Used only for right-to-left reading text (e.g., middle eastern text)
- w.TA_UPDATECP - The output routines update the current position after outputting the text.

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

Figure 6-4: Default Reference Point for Textual Output



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

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

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

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

You pass the *w.GetTextAlign* function a device context handle (that you obtain from *w.BeginPaint*) and it returns the attribute bit map value in the EAX register. You pass the *w.SetTextAlign* function the device context handle and the new bitmap value (in the *fMode* parameter). This *fMode* parameter can be any combination of the

w.TA_XXXX constants listed earlier; you may combine these constants with the HLA bitwise-OR operator (|), e.g.,

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

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

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

```
static

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

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

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

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

   SetBkMode:
      procedure
      (
```

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

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

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

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

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

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

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

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

The *w.SetBkColor* function sets the background that Windows draws when rendering text to the device. Windows draws this background color to a rectangle immediately surrounding the output text. Obviously, there

should be a fair amount of contrast between the background color you select and the foreground (text) color or the text will be difficult to read.

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

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

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

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

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

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


readonly

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

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

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

    MsgProcPtr_t:
        record

            MessageValue:   dword;
            MessageHndlr:   MsgProc_t;

        endrecord;



// The dispatch table:
//
//  This table is where you add new messages and message handlers
```

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

readonly

    Dispatch:   MsgProcPtr_t; @nostorage;

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

            // Insert new message handler records here.

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



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

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


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

    w.PostQuitMessage( 0 );

end QuitApplication;


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

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

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

    fgShades :w.RGBTRIPLE[  LinesOfText ]  :=
        [
            w.RGBTRIPLE:[  0, 0, 0],
```

```
            w.RGBTRIPLE:[  $10, $10, $10 ],
            w.RGBTRIPLE:[  $20, $20, $20 ],
            w.RGBTRIPLE:[  $30, $30, $30 ],
            w.RGBTRIPLE:[  $40, $40, $40 ],
            w.RGBTRIPLE:[  $50, $50, $50 ],
            w.RGBTRIPLE:[  $60, $60, $60 ],
            w.RGBTRIPLE:[  $70, $70, $70 ]
        ];

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


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

begin Paint;

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

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

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

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

        w.SetBkColor
        (
            hdc,
            RGB
            (
                bgShades[ i] .rgbtRed,
```

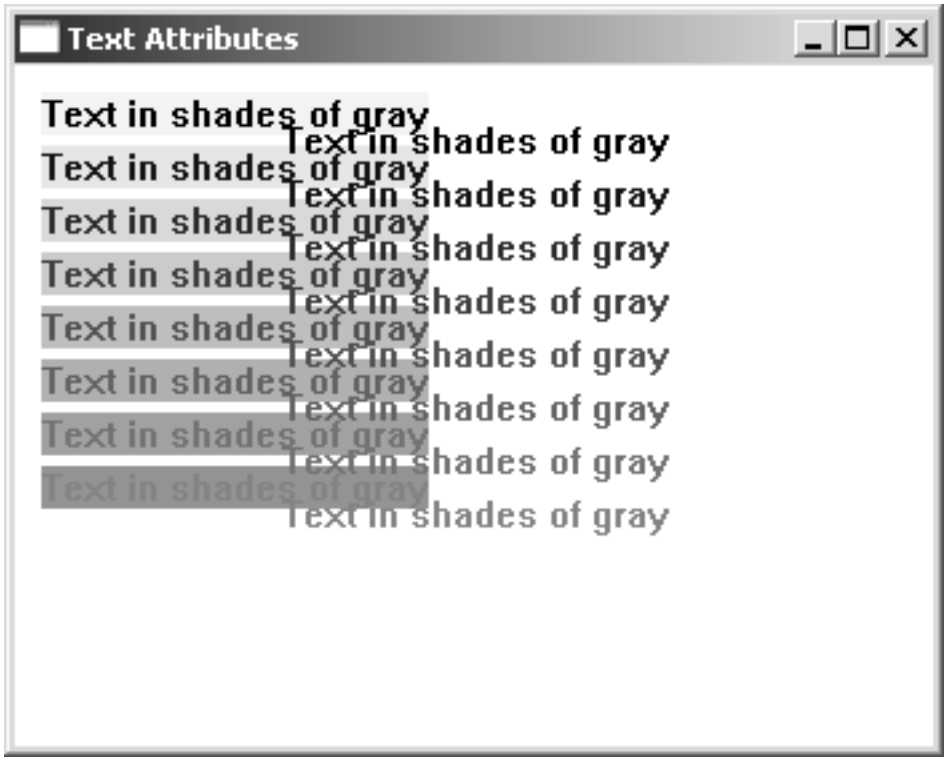
```
                bgShades[ i].rgbtGreen,
                bgShades[ i].rgbtBlue
            )
        );
        w.TextOut( hdc, 10, i*20+10, TextToDisplay, @length( TextToDisplay ));

    #endfor

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

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

    #endfor



    w.EndPaint( hwnd, ps );

end Paint;

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




// The window procedure.  Since this gets called directly from
// windows we need to explicitly reverse the parameters (compared
// to the standard STDCALL declaration) in order to make HLA's
// Pascal calling convention compatible with Windows.
//
// This is actually a function that returns a return result in
// EAX.  If this function returns zero in EAX, then the event
// loop terminates program execution.

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

begin WndProc;

    // uMsg contains the current message Windows is passing along to
    // us.  Scan through the "Dispatch" table searching for a handler
    // for this message.  If we find one, then call the associated
```

```
    // handler procedure.  If we don't have a specific handler for this
    // message, then call the default window procedure handler function.

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

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

            // If an unhandled message comes along,
            // let the default window handler process the
            // message.  Whatever (non-zero) value this function
            // returns is the return result passed on to the
            // event loop.

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


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

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

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

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

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

    endfor;

end WndProc;



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

begin TextATtr;


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

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

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

// Get this process' handle:

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

// Get the icons and cursor for this application:

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

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

// Okay, register this window with Windows so it
// will start passing messages our way.  Once this
// is accomplished, create the window and display it.

w.RegisterClassEx( wc );

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

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

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

forever

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

endfor;
```

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

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

end TextATtr;
```

Figure 6-5: Output From TextAttr Application



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

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

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

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


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

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

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

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

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

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

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

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

#endmacro
```

Now you may invoke BeginPaint and EndPaint as follows:

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

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

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

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

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

#endmacro;
```

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

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

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

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

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

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

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

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

```
    #endmacro;
```

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

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

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

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

        ];


begin Paint;

    push( ebx );

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

    BeginPaint( hwnd, ps, hdc );

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

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

```
                0
            );

        endfor;


    EndPaint;
    pop( ebx );

end Paint;
```

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

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

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

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

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

EndOfProc:
```

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

Because of their safety and convenience, this book will use these context-free macros through the remaining example programs in this book. You should consider using them in your applications for these same reasons. Of course, if you need to write one of these rare applications that needs to call these API functions in a different

order (within your source file), there is nothing stopping you from calling the original Windows functions to handle this task.

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

6.3: Device Capabilities

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

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

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

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

Table 6-1: Common GetDeviceCaps nIndex Values

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

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

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

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

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

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

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

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

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

(this application reads a subset of the *w.GetDeviceCaps* values into a record and then displays those values in a Window when the application receives a w.WM_PAINT message, see Figure 6-6 for the output).

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

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

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

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

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

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

```
        dct_t:[  "ColorRes",     "Color resolution" ]
    ];

    DCfields_c := @elements( dcTemplate_c );



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

type
    deviceCapRecord_t :record

        #for( i in dcTemplate_c )

            @text( i.fieldName ) :int32;

        #endfor

    endrecord;

    deviceCapabilities_t :union

        fields      :deviceCapRecord_t;
        elements    :int32[  DCfields_c ];

    endunion;




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

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

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

    appDevCaps: deviceCapabilities_t;

readonly

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




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

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

    MsgProcPtr_t:
        record

            MessageValue:   dword;
            MessageHndlr:   MsgProc_t;

        endrecord;



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

readonly

    Dispatch:   MsgProcPtr_t; @nostorage;

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

            // Insert new message handler records here.

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




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

procedure Create;
var
    hdc :dword;

begin Create;

    GetDC( hwnd, hdc );

        // Generate a sequence of calls to GetDeviceCaps that
        // take the form:
        //
        //  GetDeviceCaps( w.FIELDNAMEINUPPERCASE );
        //  mov( eax, appDevCaps.FieldName );
```

```
        //
        // Where the field names come from the deviceCapabilities_t type.

        #for( field in dcTemplate_c )

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

        #endfor

    ReleaseDC;

end Create;




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


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

    w.PostQuitMessage( 0 );

end QuitApplication;


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

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

type
    dclbl_t:    record

        theName     :string;
        desc        :string;

    endrecord;


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

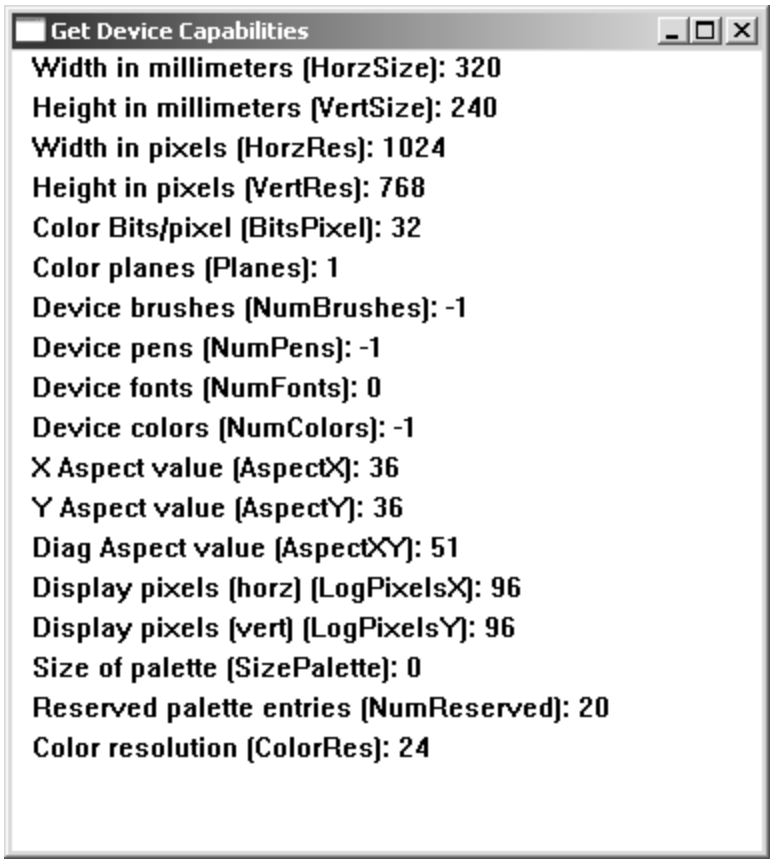
```
begin Paint;

    push( ebx );
    push( edi );

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

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

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

    BeginPaint( hwnd, ps, hdc );

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

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

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

        endfor;

    EndPaint;

    pop( edi );
    pop( ebx );

end Paint;




// The window procedure.  Since this gets called directly from
// windows we need to explicitly reverse the parameters (compared
// to the standard STDCALL declaration) in order to make HLA's
// Pascal calling convention compatible with Windows.
//
// This is actually a function that returns a return result in
// EAX.  If this function returns zero in EAX, then the event
```

```
// loop terminates program execution.

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

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

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

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

            // If an unhandled message comes along,
            // let the default window handler process the
            // message.  Whatever (non-zero) value this function
            // returns is the return result passed on to the
            // event loop.

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


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

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

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

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

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

    endfor;

end WndProc;



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

begin GDCaps;
```

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

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

// Get this process' handle:

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

// Get the icons and cursor for this application:

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

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

// Okay, register this window with Windows so it
// will start passing messages our way.  Once this
// is accomplished, create the window and display it.

w.RegisterClassEx( wc );

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

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

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

```
    forever

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

    endfor;

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

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

end GDCaps;
```

Figure 6-6: GDCaps Output



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

Device capability information tends to be static. That is, the capabilities of a device are usually fixed by the manufacturer of that device and these capabilities generally don t change while your application is running. For

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

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

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

```
type
   deviceCapRecord_t :record
      HorzSize       :int32;
      VertSize       :int32;
      HorzRes        :int32;
      BitsPixel      :int32;
      Planes         :int32;
      NumBrushes     :int32;
      NumPens        :int32;
      NumFonts       :int32;
      NumColors      :int32;
      AspectX        :int32;
```

```
      AspectY        :int32;
      AspectXY       :int32;
      LogPixelsX     :int32;
      LogPixelsY     :int32;
      SizePalette    :int32;
      NumReserved    :int32;
      ColorRes       :int32;
   endrecord;
```

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

```
type
    deviceCapRecord_t :record

        #for( i in dcTemplate_c )

            @text( i.fieldName ) :int32;

        #endfor

    endrecord;
```

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

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

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

const
    dcTemplate_c :dct_t[ ] :=
    [
        dct_t:[ "HorzSize",     "Width in millimeters" ],
```

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

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

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

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

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

```
GetDC( hwnd, hdc );

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

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

```
ReleaseDC;
```

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

```
#for( field in dcTemplate_c )

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

#endfor
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

To print the data to the window, the *Paint* procedure converts the numeric information in the corresponding appDevCaps array element to a string and then concatenates the two strings in the *DClabels* array with this

numeric string and calls *w.TextOut* to display the whole string. This sounds like a bit of work, but the HLA Standard Library comes to the rescue with the *str.put* function that lets you do all this work with a single function call:

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

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

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

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

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

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

6.4: Typefaces and Fonts

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

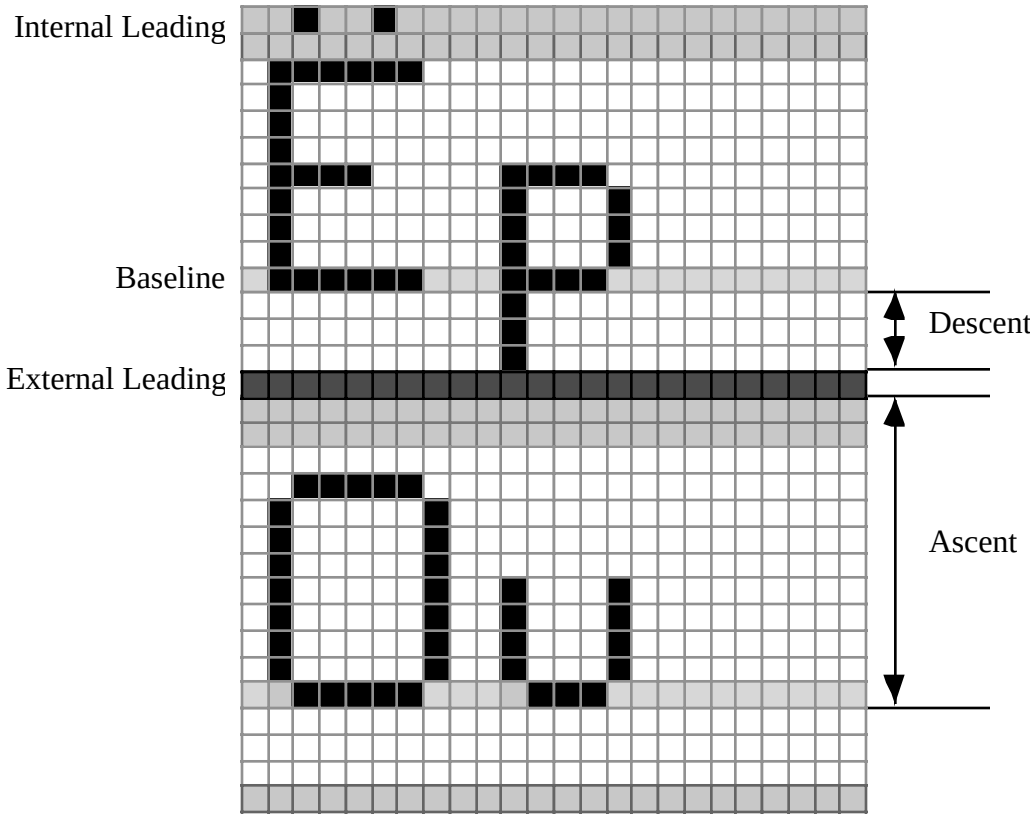
Perhaps the first place to start this discussion is with the definition of a *font*. This side trip is necessary because many people confuse a font with a *typeface*. A *font* is a collection of attributes associated with printed or displayed text. The attributes that define a font include the typeface,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

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

Another attribute of a font is the size of that font. Font sizes are usually specified in *points*. A point is approximately $^{1}/_{72}$ inches. Therefore, a font with a 72 point size consists of characters that are approximately one inch

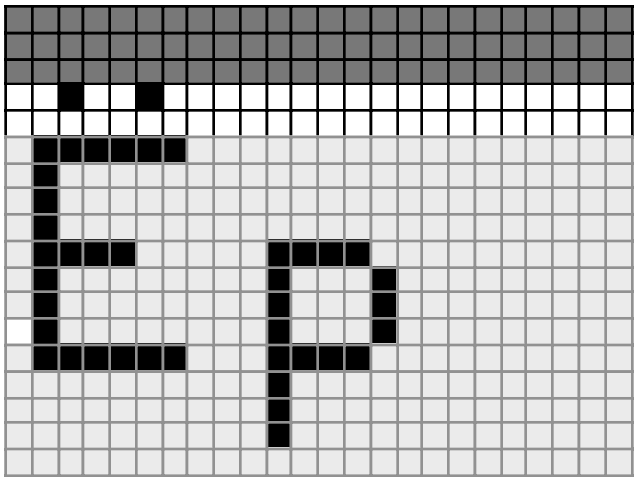
tall. An important thing to realize is that 72 point characters are not all approximately one inch tall. The size of a font specifies a set of boundaries between which typical characters in that font actually fall. A 72-point lowercase a , for example, is smaller than a 72-point uppercase A even though they both have the same size . To understand how graphic artists measure font sizes, consider Figure 6-7. This figure specifies four terms that graphic artists typically use when discussing the size of a font: leading, baseline, descent, and ascent.

Figure 6-7: Font Sizes



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

Figure 6-8: Internal Versus External Leading



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

Figure 6-9: The Ascent of a Font

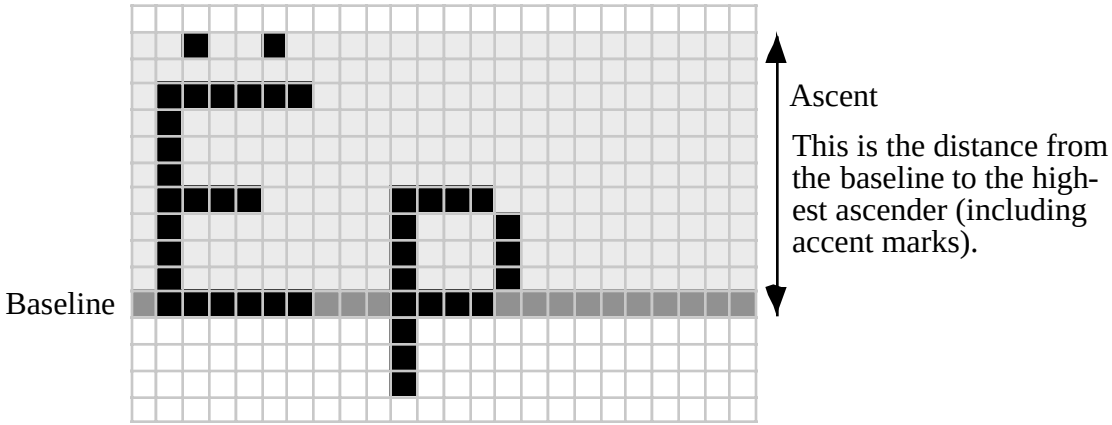


Figure 6-10: The Descent of a Font

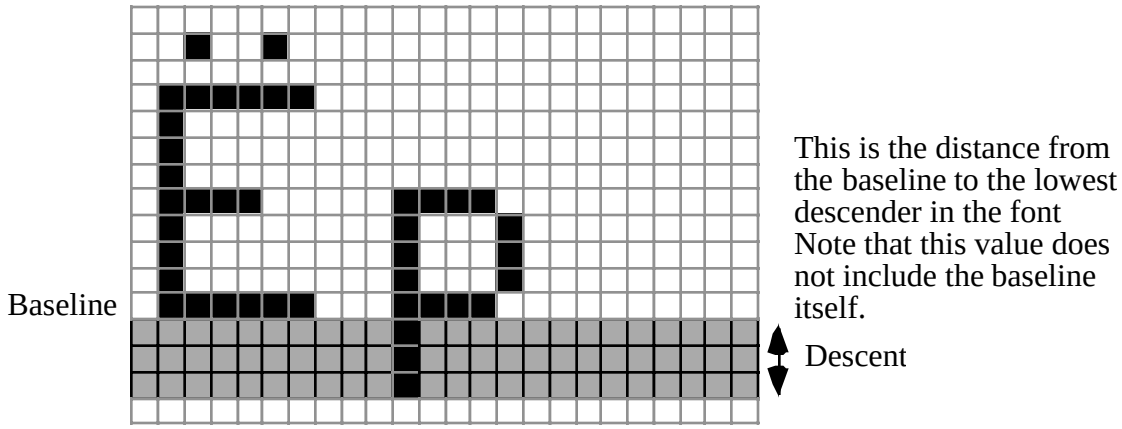
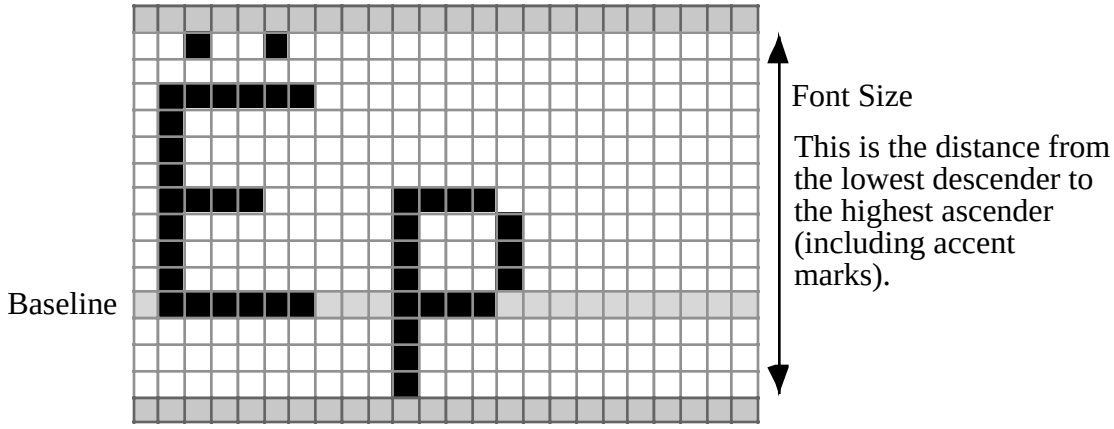


Figure 6-11: The Size of a Font



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

The exact width of characters within a font is typically a function of the typeface. However, we can classify fonts into two categories based upon the widths of the characters within the font: proportional fonts and non-proportional (or monospaced) fonts. In a proportional font, the widths of the characters vary according to the particular character and the whim of the font designer. Certain characters, like W require more horizontal space than other characters (e.g., i) in order to look proper within a line of text. As a result, you cannot determine the amount of horizontal space a string of characters will consume by simply counting the number of characters in

the string. Instead, you ll have to sum up the widths of each of the individual characters in the string in order to determine the width of the whole string. This makes it difficult to predict the number of characters you can display within a given horizontal distance without actually having the characters to work with.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- w.FW_DONTCARE (0)
- w.FW_THIN (100)
- w.FW_EXTRALIGHT (200)
- w.FW_ULTRALIGHT (200)
- w.FW_LIGHT (300)
- w.FW_NORMAL (400)
- w.FW_MEDIUM (500)
- w.FW_SEMIBOLD (600)
- w.FW_DEMIBOLD (600)
- w.FW_BOLD (700)
- w.FW_EXTRABOLD (800)
- w.FW_ULTRABOLD (800)
- w.FW_HEAVY (900)
- w.FW_BLACK (900)

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

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

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

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

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

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

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

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

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

- w.FF_DECORATIVE - use a decorative font (like Old English)
- w.FF_DONTCARE - use an arbitrary font (Windows chooses)
- w.FF_MODERN - generally specified for monospaced fonts
- w.FF_ROMAN - generally specified for proportional, serifed, fonts
- w.FF_SCRIPT - specifies a font that uses a cursive (handwritten) style
- w.FF_SWISS - specifies a proportional, sans-serifed font

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

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

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

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

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

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

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

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

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

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

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

EndPaint;
```

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

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

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

```
type
        TEXTMETRIC: record

            tmHeight            :dword;
            tmAscent            :dword;
            tmDescent           :dword;
            tmInternalLeading   :dword;
```

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

    endrecord;
```

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

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

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

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

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

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

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

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

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


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

The *w.EnumFontFamilies* function requires these parameters:

- *hdc* - The handle for the device context whose fonts you wish to enumerate. Note that *w.EnumFontFamilies* always iterates over the fonts for a specific device context.
- *lpszFamily* - the address of a zero-terminated (e.g., HLA) string. This string must contain the name of the font family you wish to enumerate, or NULL if you want the function to enumerate all font families in the system.
- *lpEnumFontFamProc* - the address of a call back function. Windows will call this function once for each font it enumerates. Your application must supply this function and Windows will pass information about the font to this callback function. Note that the call function s declaration must exactly match the *w.FONTENUMPROC* definition.

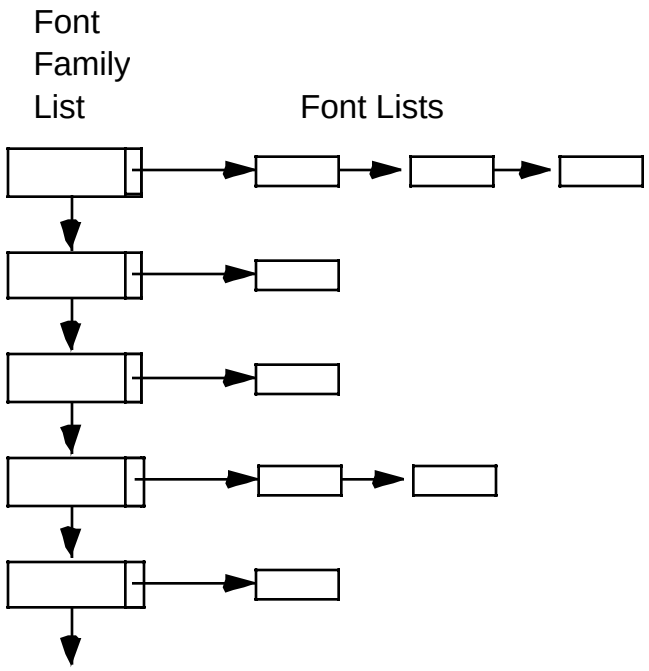
- ¥ *lParam* - this is a 32-bit application-specific piece of data that the font numeration code passes on to the callback routine (in the lParam2 parameter). Generally, you will pass some application-specific data to the callback routine (such as the address where the callback routine can store the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

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

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

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

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



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

```
type
    fFamily_t:
```

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

            override procedure create;
            override method destroy;

        endclass;
```

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

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

            override procedure create;
            override method destroy;

        endclass;
```

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

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

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

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

begin Create;

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

        // Enumerate the families:

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

        // Enumerate the fonts appearing in each family:

        foreach fontFamList.itemInList() do
```

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

        endfor;

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

end Create;
```

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

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

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

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

begin FontFamilyCallback;

    push( esi );
    push( edi );

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

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

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

    fontFamList.append( curFam );

    // Initialize the font family object we just created:

    mov( curFam, esi );

    // Initialize the string containing the font family name:

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

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

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

    // Return success

    mov( 1, eax );

    pop( edi );
    pop( esi );

end FontFamilyCallback;
```

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

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

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

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

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

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

var
    curFont :pointer to font_t;

begin EnumSingleFamily;

    push( esi );
    push( edi );

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

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

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

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

    // Initialize the string containing the font family name:

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

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

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

    mov( curFont, esi );
    lea( edi, (type font_t [ esi]).lf );
```

```
    mov( lplf, esi );
    mov( @size( w.LOGFONT ), ecx );
    rep.movsb();

    mov( 1, eax ); // Return success

    pop( edi );
    pop( esi );

end EnumSingleFamily;
```

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

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

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

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

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

begin Paint;

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

    push( edi );
```

```
push( ebx );

BeginPaint( hwnd, ps, hdc );

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

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

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

    mov( -10, yCoordinate );

    // Okay, output a sample of each font:

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

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

        add( 10, yCoordinate );

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

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

        if( eax == 1 ) then

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

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

        else

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

            str.put
            (
                outputMsg,
```

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

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

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

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

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

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

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

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

    // Select the new font into the device context:

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

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

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

    // Output the font info:

    str.put
    (
        outputMsg,
        (type font_t [ esi] ).fontName,
```

```
                    " (Size in points: ",
                    (type uns32 eax),
                    ')'
                );
                w.TextOut
                (
                    hdc,
                    20,
                    yCoordinate,
                    outputMsg,
                    str.length( outputMsg )
                );

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

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

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

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

            endfor;

        endfor;
        pop( esi );

    EndPaint;

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

end Paint;
```

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

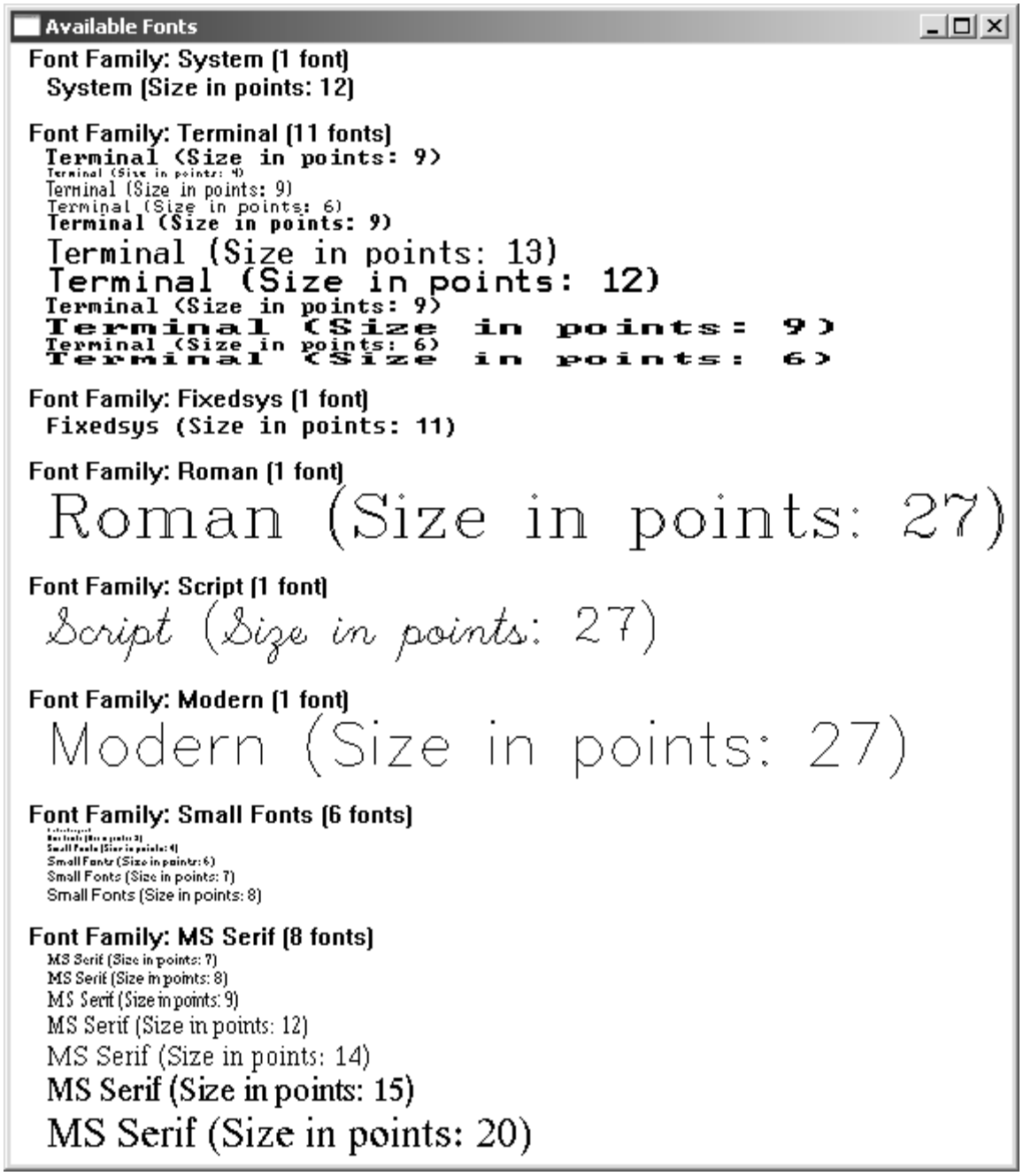
For each font family, the *Paint* procedure draws a line of text (in the system font) specifying the font family s name. *Paint* tracks the output position (y-coordinate) using the *yCoordinate* local variable. For each line of text that this procedure outputs, it adds the height of the font (plus external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

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

ing the height of the font (in pixels) by the number of points/inch (72) and then divides this product by the number of pixels/inch in the device context. After all this work, `foreach` loop displays the name of that font (in that font) along with the size of the font.

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

Figure 6-13: Fonts Output



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

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

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

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

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

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

            override procedure create;
            override method destroy;

        endclass;


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

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

            override procedure create;
            override method destroy;

        endclass;


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

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

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


readonly
```

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

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

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

    MsgProcPtr_t:
        record

            MessageValue:   dword;
            MessageHndlr:   MsgProc_t;

        endrecord;



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

readonly

    Dispatch:   MsgProcPtr_t; @nostorage;

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

            // Insert new message handler records here.

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



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


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

```
procedure font_t.create;
begin create;

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

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

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

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

    // Zero out the tm and lf data structures:

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

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

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

end create;

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

method font_t.destroy;
begin destroy;

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

    if( this.fontName <> NULL ) then

        if( strIsInHeap( this.fontName )) then

            strfree( this.fontName );
```

```
        endif;

    endif;

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

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

        free( esi );

    endif;

end destroy;







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

procedure fFamily_t.create;
begin create;

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

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

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

    endif;

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

    mov( NULL, this.familyName );

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

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

    pop( eax );
```

```
end create;


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

method fFamily_t.destroy;
begin destroy;

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

    if( this.familyName <> NULL ) then

        if( strIsInHeap( this.familyName )) then

            strfree( this.familyName );

        endif;

    endif;

    // Free up the font list:

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

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

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

        free( esi );

    endif;

end destroy;




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

```
// end of the "fontFamList" list.

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

begin FontFamilyCallback;

    push( esi );
    push( edi );

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

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

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

    fontFamList.append( curFam );

    // Initialize the font family object we just created:

    mov( curFam, esi );

    // Initialize the string containing the font family name:

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

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

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

    // Return success

    mov( 1, eax );

    pop( edi );
    pop( esi );

end FontFamilyCallback;



// Font callback function that enumerates a single font.
```

```
// On entry, lparam points at a fFamily_t element whose
// fonts list we append the information to.

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

var
    curFont :pointer to font_t;

begin EnumSingleFamily;

    push( esi );
    push( edi );

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

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

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

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

    // Initialize the string containing the font family name:

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

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

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

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

    mov( 1, eax ); // Return success

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

```
end EnumSingleFamily;



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

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


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

    w.PostQuitMessage( 0 );

end QuitApplication;


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

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

begin Create;

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

        // Enumerate the families:

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

        // Enumerate the fonts appearing in each family:

        foreach fontFamList.itemInList() do

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

```
        endfor;

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

end Create;


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

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

begin Paint;

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

    push( edi );
    push( ebx );

    BeginPaint( hwnd, ps, hdc );

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

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

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

        mov( -10, yCoordinate );

        // Okay, output a sample of each font:

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

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

            add( 10, yCoordinate );
```

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

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

if( eax == 1 ) then

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

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

else

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

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

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

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

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

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

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

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

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

    // Select the new font into the device context:

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

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

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

    // Output the font info:

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

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

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

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

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

endfor;
```

```
        endfor;
        pop( esi );

    EndPaint;

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

end Paint;

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




// The window procedure.  Since this gets called directly from
// windows we need to explicitly reverse the parameters (compared
// to the standard STDCALL declaration) in order to make HLA's
// Pascal calling convention compatible with Windows.
//
// This is actually a function that returns a return result in
// EAX.  If this function returns zero in EAX, then the event
// loop terminates program execution.

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

begin WndProc;

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

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

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

            // If an unhandled message comes along,
            // let the default window handler process the
            // message.  Whatever (non-zero) value this function
            // returns is the return result passed on to the
            // event loop.

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

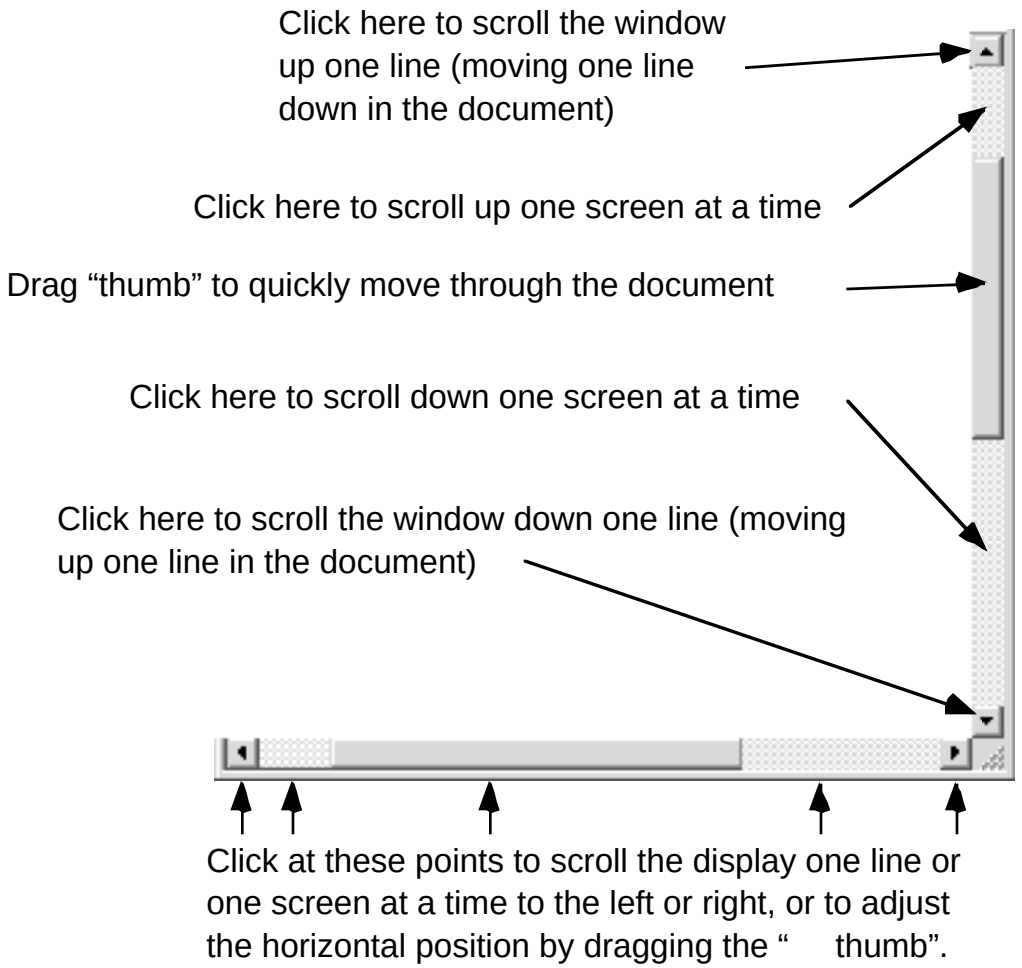
```
        elseif( eax = (type MsgProcPtr_t [ edx] ).MessageValue ) then

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

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

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

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

    endfor;

end WndProc;



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

begin Fonts;

    // Create the font family list here:

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

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

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

    // Get this process' handle:

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

    // Get the icons and cursor for this application:

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

```
    mov( eax, wc.hIconSm );

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

    // Okay, register this window with Windows so it
    // will start passing messages our way.  Once this
    // is accomplished, create the window and display it.

    w.RegisterClassEx( wc );

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

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

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

    forever

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

    endfor;

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

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

end Fonts;
```

6.5: Scroll Bars

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

Figure 6-14: Scroll Bar Actions



From the user s perspective, the scroll bars move the document around within the window. From an application s perspective, however, what is really going on is that the scroll bars are repositioning the window over the document. If we think of coordinate (0,0) as being the upper-left hand pixel in the document, then adjust the view using the scroll bars simply defines the coordinate in the document that corresponds to the upper-left hand corner of the window within the document. This is why clicking on the up arrow on the scroll bar actually cause the

contents of the window to scroll down. What's really happening is that the user is moving the "starting coordinate" of the window up on line in the document. Because the top of the window is now displaying one line earlier in the document, the contents of the window shifts down one line. A similar explanation applies to scrolling data left or right in the window via the scroll bars.

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

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

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

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

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

    GetScrollRange: procedure
    (
            hWnd        :dword;
            nBar        :dword;
        var lpMinPos    :dword;
        var lpMaxPos    :dword
```

```
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__GetScrollRange@16" );
```

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

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

- *w.SB_CTL* - use this constant if you re reading the value of a custom scroll bar control. When you supply this constant, the *hWnd* parameter must be the handle of the scroll bar control whose value(s) you wish to retrieve.
- *w.SB_HORZ* - use this constant if you want to retrieve the position or range values for the horizontal scroll bar in a standard window with the *w.WS_HSCROLL* or *w.WS_VSCROLL* styles.
- *w.SB_VERT* - use this constant if you want to retrieve the position or range values for the vertical scroll bar in a standard window with the *w.WS_HSCROLL* or *w.WS_VSCROLL* styles.

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

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

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

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

bar; setting *bRedraw* to false prevents Windows from redrawing the scroll bar twice (which slows down your application and make cause the scroll bar region to flash momentarily).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- *w.SM_CXBORDER* - width, in pixels, of a window border.
- *w.SM_CYBORDER* - height, in pixels, of a window border.
- *w.SM_CMOUSEBUTTONS* - number of buttons on the mouse.
- *w.SM_CXFULLSCREEN* - width, in pixels, of the client area for a full-screen window on the primary display.
- *w.SM_CYFULLSCREEN* - height, in pixels, of the client area for a full-screen window on the primary display.
- *w.SM_CXICON* - the width, in pixels, of an icon.
- *w.SM_CYICON* - the height, in pixels, of an icon.
- *w.SM_CXSCREEN* - the width, in pixels, of the primary display.
- *w.SM_CYSCREEN* - the height, in pixels, of the primary display.
- etc. See the Microsoft documentation for all the possible constants and their meaning.

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

```
MetricRec_t:
        record

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

        endrecord;
```

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

```
readonly

    MetricData: MetricRec_t[] :=
```

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

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

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

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

```
readonly

    Dispatch    :MsgProcPtr_t; @nostorage;
```

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

            // Insert new message handler records here.

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

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

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

```
static

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

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

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

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

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

    GetDC( hwnd, hdc );

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

        GetTextMetrics( tm );

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

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

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

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

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

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

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

end Create;
```

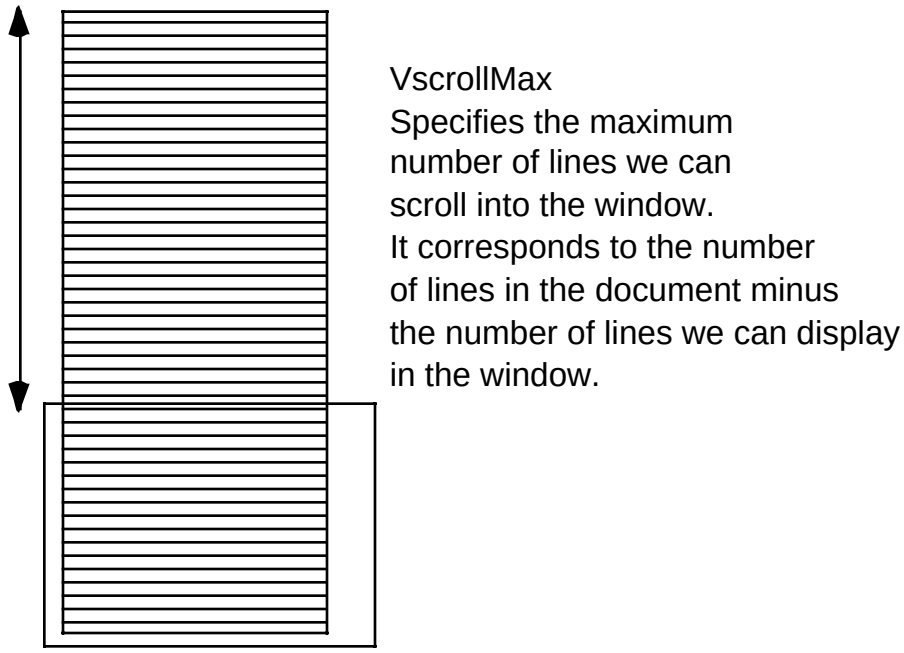
Whenever windows first creates a window, or whenever the user resizes the window, Windows will send a *w.WM_SIZE* message to the window procedure. Programs we ve written in the past have simply ignored this message. However, once you add scroll bars to your window you need to intercept this message so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

the current *VscrollPos, VscrollMax, HscrollPos,* and *HscrollMax* variables. These variables have the following declarations in the global data area:

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

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

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

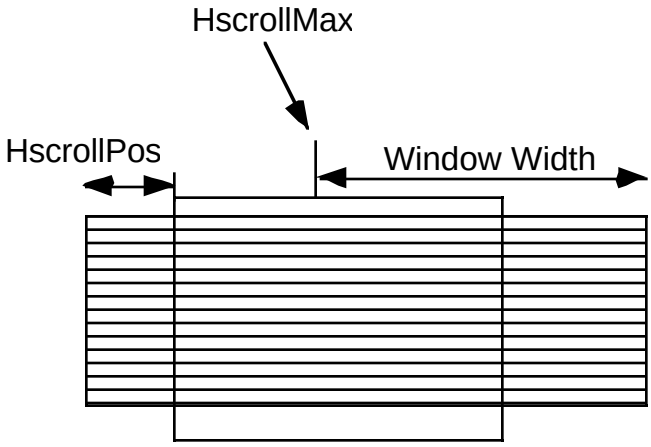
Figure 6-15: VscrollMax Value



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

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

Figure 6-16: HScrollMax and HscrollPos Values



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

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

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

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

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

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

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

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

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

        xor( ecx, ecx );

    endif;
    mov( ecx, VscrollMax );

    // VscrollPos = min( VscrollPos, VscrollMax )
```

```
    if( ecx > VscrollPos ) then

        mov( VscrollPos, ecx );

    endif;
    mov( ecx, VscrollPos );

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

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

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

        xor( eax, eax );

    endif;
    mov( eax, HscrollMax );

    // HscrollPos = min( HscrollMax, HscrollPos )

    if( eax > HscrollPos ) then

        mov( HscrollPos, eax );

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


end Size;
```

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

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

of 25% to 50% of the window s width would be a decent amount to scroll. Such a modification to the *sysmet* program is rather trivial; feel free to do it as an experiment with this program.

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

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

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

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

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

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

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

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

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

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

```
// switch macro:

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

     case( w.SB_LINEUP )

         mov( -1, eax );

     case( w.SB_LINEDOWN )

         mov( 1, eax );

     case( w.SB_PAGEUP )

         mov( -8, eax );

     case( w.SB_PAGEDOWN )

         mov( 8, eax );

     case( w.SB_THUMBPOSITION )

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

     default

         xor( eax, eax );

 endswitch;

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

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

     mov( ecx, eax );

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

     mov( edx, eax );

 endif;
 if( eax <> 0 ) then

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

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

end HScroll;
```

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

Here s the complete *VScroll* procedure:

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

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

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

        case( w.SB_TOP )

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

        case( w.SB_BOTTOM )

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

        case( w.SB_LINEUP )

            mov( -1, eax );

        case( w.SB_LINEDOWN )

            mov( 1, eax );
```

```
    case( w.SB_PAGEUP )

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

            mov( -1, eax );

        endif;

    case( w.SB_PAGEDOWN )

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

            mov( 1, eax );

        endif;

    case( w.SB_THUMBTRACK )

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

    default

        xor( eax, eax );

endswitch;

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

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

    mov( ecx, eax );

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

    mov( edx, eax );

endif;

if( eax <> 0 ) then

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

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

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


end VScroll;
```

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

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

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

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

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

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

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

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


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

    MetricRec_t:
        record

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

        endrecord;


    // Message and dispatch table related definitions:

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

    MsgProcPtr_t:
        record

            MessageValue    :dword;
            MessageHndlr    :MsgProc_t;

        endrecord;




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

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

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

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


readonly

    ClassName   :string := "SMWinClass";                // Window Class Name
```

```
    AppCaption  :string := "System Metrics Program";    // Caption for Window


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


    Dispatch    :MsgProcPtr_t; @nostorage;

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

            // Insert new message handler records here.

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



readonly

    MetricData: MetricRec_t[ ] :=
    [
        MetricRec_t:[  w.SM_CXSCREEN, "w.SM_CXSCREEN", "Screen width" ],
        MetricRec_t:[  w.SM_CYSCREEN, "w.SM_CYSCREEN", "Screen height" ],
        MetricRec_t:[  w.SM_CXVSCROLL, "w.SM_CXVSCROLL", "Vert scroll arrow width" ],
        MetricRec_t:[  w.SM_CYVSCROLL, "w.SM_CYVSCROLL", "Vert scroll arrow ht" ],
        MetricRec_t:[  w.SM_CXHSCROLL, "w.SM_CXHSCROLL", "Horz scroll arrow width" ],
        MetricRec_t:[  w.SM_CYHSCROLL, "w.SM_CYHSCROLL", "Horz scroll arrow ht" ],
        MetricRec_t:[  w.SM_CYCAPTION, "w.SM_CYCAPTION", "Caption bar ht" ],
        MetricRec_t:[  w.SM_CXBORDER, "w.SM_CXBORDER", "Window border width" ],
        MetricRec_t:[  w.SM_CYBORDER, "w.SM_CYBORDER", "Window border height" ],
        MetricRec_t:[  w.SM_CXDLGFRAME, "w.SM_CXDLGFRAME", "Dialog frame width" ],
        MetricRec_t:[  w.SM_CYDLGFRAME, "w.SM_CYDLGFRAME", "Dialog frame height" ],
        MetricRec_t:[  w.SM_CXHTHUMB, "w.SM_CXHTHUMB", "Horz scroll thumb width" ],
        MetricRec_t:[  w.SM_CYVTHUMB, "w.SM_CYVTHUMB", "Vert scroll thumb width" ],
        MetricRec_t:[  w.SM_CXICON, "w.SM_CXICON", "Icon width" ],
        MetricRec_t:[  w.SM_CYICON, "w.SM_CYICON", "Icon height" ],
        MetricRec_t:[  w.SM_CXCURSOR, "w.SM_CXCURSOR", "Cursor width" ],
        MetricRec_t:[  w.SM_CYCURSOR, "w.SM_CYCURSOR", "Cursor height" ],
        MetricRec_t:[  w.SM_CYMENU,  "w.SM_CYMENU", "Menu bar height" ],
        MetricRec_t:[  w.SM_CXFULLSCREEN, "w.SM_CXFULLSCREEN", "Largest client width" ],
        MetricRec_t:[  w.SM_CYFULLSCREEN, "w.SM_CYFULLSCREEN", "Largets client ht" ],
        MetricRec_t:[  w.SM_DEBUG, "w.SM_CDEBUG", "Debug version flag" ],
        MetricRec_t:[  w.SM_SWAPBUTTON, "w.SM_CSWAPBUTTON", "Mouse buttons swapped" ],
        MetricRec_t:[  w.SM_CXMIN, "w.SM_CXMIN", "Minimum window width" ],
        MetricRec_t:[  w.SM_CYMIN, "w.SM_CYMIN", "Minimum window height" ],
```

```
        MetricRec_t:[  w.SM_CXSIZE, "w.SM_CXSIZE", "Minimize/maximize icon width" ],
        MetricRec_t:[  w.SM_CYSIZE, "w.SM_CYSIZE", "Minimize/maximize icon height" ],
        MetricRec_t:[  w.SM_CXFRAME, "w.SM_CXFRAME", "Window frame width" ],
        MetricRec_t:[  w.SM_CYFRAME, "w.SM_CYFRAME", "Window frame height" ],
        MetricRec_t:[  w.SM_CXMINTRACK,  "w.SM_CXMINTRACK", "Minimum tracking width" ],
        MetricRec_t:[  w.SM_CXMAXTRACK,  "w.SM_CXMAXTRACK", "Maximum tracking width" ],
        MetricRec_t:[  w.SM_CYMINTRACK,  "w.SM_CYMINTRACK", "Minimum tracking ht" ],
        MetricRec_t:[  w.SM_CYMAXTRACK,  "w.SM_CYMAXTRACK", "Maximum tracking ht" ],
        MetricRec_t:[  w.SM_CXDOUBLECLK, "w.SM_CXDOUBLECLK", "Dbl-click X tolerance" ],
        MetricRec_t:[  w.SM_CYDOUBLECLK, "w.SM_CYDOUBLECLK", "Dbl-click Y tolerance" ],
        MetricRec_t:[  w.SM_CXICONSPACING, "w.SM_CXICONSPACING", "Horz icon spacing" ],
        MetricRec_t:[  w.SM_CYICONSPACING, "w.SM_CYICONSPACING", "Vert icon spacing" ],
        MetricRec_t:[  w.SM_CMOUSEBUTTONS, "w.SM_CMOUSEBUTTONS", " # of mouse btns" ]
    ];

const
    NumMetrics := @elements( MetricData );


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

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

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

    w.PostQuitMessage( 0 );

end QuitApplication;


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

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

    GetDC( hwnd, hdc );

        // Initialization:
        //
        //  Get the text metric information so we can compute
```

```
        //  the average character heights and widths.

        GetTextMetrics( tm );

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

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

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

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

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

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

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

end Create;


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

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

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

    CurDesc     :string;        // Current system metrics description.
    CDlen       :string;        // Length of the above.
    CDx         :int32;         // X position for CurDesc string.
```

```
    value       :string;
    valData     :char[ 32];
    CVx         :int32;         // X position for value string.
    vallen      :uns32;         // Length of value string.

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

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

begin Paint;

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

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


    // Initialize the value->valData string object:

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

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

    BeginPaint( hwnd, ps, hdc );

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

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

            xor( eax, eax );

        endif;
        mov( eax, firstMet );


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

        mov( ps.rcPaint.bottom, eax );
```

```
cdq();
idiv( AverageCharHeight );
add( VscrollPos, eax );
if( (type int32 eax) > NumMetrics ) then

    mov( NumMetrics, eax );

endif;
mov( eax, lastMet );


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

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

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

    // Compute index into MetricData:

    intmul( @size( MetricRec_t ), edi, esi );

    // Grab the string from the current MetricData element:

    mov( MetricData.MetStr[ esi ], eax );
    mov( eax, CurVar );
    mov( (type str.strRec [eax]).length, eax );
    mov( eax, CVlen );

    mov( MetricData.MetDesc[ esi ], eax );
    mov( eax, CurDesc );
    mov( (type str.strRec [eax]).length, eax );
    mov( eax, CDlen );

    // Column one begins at X-position AverageCharWidth (ACW).
    // Col 2 begins at ACW + 25*AverageCapsWidth.
    // Col 3 begins at ACW + 25*AverageCapsWidth + 40*ACW.
    // Compute the Col 2 and Col 3 values here.

    mov( 1, eax );
    sub( HscrollPos, eax );
    intmul( AverageCharWidth, eax );
    mov( eax, x );
```

```
    intmul( 25, AverageCapsWidth, eax );
    add( x, eax );
    mov( eax, CDx );

    intmul( 40, AverageCharWidth, ecx );
    add( ecx, eax );
    mov( eax, CVx );

    // The Y-coordinate for the line of text we're writing
    // is computed as AverageCharHeight * (1-VscrollPos+edi).
    // Compute that value here:

    mov( 1, eax );
    sub( VscrollPos, eax );
    add( edi, eax );
    intmul( AverageCharHeight, eax );
    mov( eax, y );


    // Now generate the string we're going to print
    // as the value for the current metric variable:

    w.GetSystemMetrics( MetricData.MetConst[ esi ] );
    conv.i32ToStr( eax, 0, ' ', value );
    mov( str.length( value ), vallen );


    // First two columns have left-aligned text:

    SetTextAlign( w.TA_LEFT | w.TA_TOP );

    // Output the name of the metric variable:

    TextOut( x, y, CurVar, CVlen );

    // Output the description of the metric variable:

    TextOut( CDx, y, CurDesc, CDlen );

    // Output the metric's value in the third column.  This is
    // a numeric value, so we'll right align this data.

    SetTextAlign( w.TA_RIGHT | w.TA_TOP );
    TextOut( CVx, y, value, vallen );

    // Although not strictly necessary for this program,
    // it's a good idea to always restore the alignment
    // back to the default (top/left) after you done using
    // some other alignment.

    SetTextAlign( w.TA_LEFT | w.TA_TOP );


endfor;
```

```
    EndPaint;

    pop( edi );
    pop( esi );
    pop( ebx );

end Paint;



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

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

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

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

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

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

        xor( ecx, ecx );

    endif;
    mov( ecx, VscrollMax );

    // VscrollPos = min( VscrollPos, VscrollMax )

    if( ecx > VscrollPos ) then

        mov( VscrollPos, ecx );

    endif;
    mov( ecx, VscrollPos );

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

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

    mov( MaxWidth, eax );
    sub( ClientSizeX, eax );
    cdq();
```

```
    idiv( AverageCharWidth );
    add( 2, eax );
    if( @s ) then

        xor( eax, eax );

    endif;
    mov( eax, HscrollMax );

    // HscrollPos = min( HscrollMax, HscrollPos )

    if( eax > HscrollPos ) then

        mov( HscrollPos, eax );

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


end Size;



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

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

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

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

        case( w.SB_LINEUP )

            mov( -1, eax );

        case( w.SB_LINEDOWN )

            mov( 1, eax );

        case( w.SB_PAGEUP )

            mov( -8, eax );

        case( w.SB_PAGEDOWN )

            mov( 8, eax );

        case( w.SB_THUMBPOSITION )
```

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

        default

            xor( eax, eax );

    endswitch;

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

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

        mov( ecx, eax );

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

        mov( edx, eax );

    endif;
    if( eax <> 0 ) then

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

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

end HScroll;



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

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

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

        case( w.SB_TOP )
```

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

    case( w.SB_BOTTOM )

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

    case( w.SB_LINEUP )

        mov( -1, eax );

    case( w.SB_LINEDOWN )

        mov( 1, eax );

    case( w.SB_PAGEUP )

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

            mov( -1, eax );

        endif;

    case( w.SB_PAGEDOWN )

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

            mov( 1, eax );

        endif;

    case( w.SB_THUMBTRACK )

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

    default

        xor( eax, eax );

endswitch;

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

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

```
        mov( ecx, eax );

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

        mov( edx, eax );

    endif;

    if( eax <> 0 ) then

        add( eax, VscrollPos );
        intmul( AverageCharHeight, eax );
        neg( eax );
        w.ScrollWindow( hwnd, 0, eax, NULL, NULL );
        w.SetScrollPos( hwnd, w.SB_VERT, VscrollPos, true );
        w.UpdateWindow( hwnd );

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


end VScroll;



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




// The window procedure.
// This is actually a function that returns a return result in
// EAX.  If this function returns zero in EAX, then the event
// loop terminates program execution.

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

begin WndProc;

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

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

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

```
            // If an unhandled message comes along,
            // let the default window handler process the
            // message.  Whatever (non-zero) value this function
            // returns is the return result passed on to the
            // event loop.

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


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

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

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

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

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

    endfor;

end WndProc;



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

begin systemMetrics;

    // Get this process' handle:

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


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

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

    // Get the icons and cursor for this application:
```

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

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


// Okay, register this window with Windows so it
// will start passing messages our way.  Once this
// is accomplished, create the window and display it.

w.RegisterClassEx( wc );

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

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

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

forever

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

endfor;

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

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

```
end systemMetrics;
```

6.6: The DebugWindow Application

When moving from console applications to GUI applications, an important feature is lost to the application programmer - the ability to print a string of text from an arbitrary point in the program that displays the program's status and other debugging information. In this section we will explore how to write a simple terminal emulator application that displays text much like console applications process text. This simple application will demonstrate how to create console-like applications within a GUI environment. One feature of this particular application, however, is that it displays textual data sent to it from other applications. This allows you to write a GUI application that can send text messages to this simple console emulator application, allowing you to display debug (and other status messages) from a GUI application that doesn't normally support console-like output.

Writing a *DebugWindow* application that accepts messages from other applications makes use of some advanced *message passing* features in Windows. Few books on Windows programming would present this topic so early in the book. This book makes an exception to this rule for a couple of reasons. First, though the whole concept of message passing and multitasking applications is a bit advanced, you don't have to learn everything there is to know about this subject area to create the *DebugWindow* application. This section will only present a few of the API calls that you will need to implement this advanced form of message passing, so this shouldn't be a problem. Another reason for presenting this application here is that *DebugWindow* is, intrinsically, a text-based application. So describing the operation of this application in a chapter on text processing in Windows seems like a natural fit. Finally, one really good reason for presenting this application here is because you'll find it extremely useful and the sooner you have this application available, the sooner you'll be able to employ it in your own Windows projects. So the sooner the better...

6.6.1: Message Passing Under Windows

Windows communicates with your applications by passing them *messages*. Your window procedure is the code that Windows calls when it sends your application a message. Window procedures in code we've written up to this point have processed messages like `w.WM_PAINT`, `w.WM_CREATE`, and `w.WM_DESTROY` that Windows has passed to these applications. Although Windows is the most common source of messages to your applications, it's also possible for your application to send messages to itself or send messages to other processes in the system. For example, it's perfectly possible for you to send a `w.WM_PAINT` message to your program to force the window to repaint itself. Indeed, it's reasonable to manually send almost any message Windows sends to your window procedure. Messages provide a form of "deferred procedure call" that you can use to call the message handling procedures that your window procedure calls.

In addition to the stock messages that Windows defines, Windows also predefines a couple of sets of user-definable messages. These user-definable messages let your application create its own set of procedures that can be called via the message passing mechanism. Windows defines three ranges of user-definable messages. The first set of user-definable messages have values in the range `w.WM_USER`..$7FFF. Windows reserves messages in this range for private use by a window procedure. Note, however, that Window messages in this range are only meaningful within a single window class, they are not unique throughout an application. For example, some stock controls that Windows provides will use message values in this range. However, if you are sending a message to a specific window (which is a member of some specific window class), then you can define private messages that your window procedure will respond to using the values in this range.

If you need a range of messages that are unique throughout an application (e.g., you're going to broadcast a message to all active windows within a given application), then you'll want to use the message values in the

range *w.WM_APP*..$BFFF. Windows guarantees that no system messages use these values, so if you broadcast a message whose message number is in this range, then none of the Windows system window procedures (e.g., button, text edit boxes, list boxes, etc.) will inadvertently respond to messages in this range.

If two different processes want to pass messages between themselves, allocating hard-coded message numbers is not a good idea (because it s possible that some third process could accidentally intercept such messages if it also uses the same message number for inter-process communication). Therefore, Windows defines a final range of message values ($C000..$FFFF) that Windows explicitly manages via the *w.RegisterWindowMessage* API call. Here s the prototype for this API function:

```
static
    RegisterWindowMessage: procedure
    (
        lpString        :string
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__RegisterWindowMessageA@4" );
```

The parameter is an HLA string that uniquely identifies the message value to create. Windows will search for this string in an internal database and return whatever value is associated with that string. If no such string exists internal to Windows, then Windows will allocate a unique message number in the range $C000..$FFFF and return that value (*w.RegisterWindowMessage* returns the message value in the EAX register). The message number remains active until all programs that have registered it terminate execution. Note that unlike the other message values, that you can represent via constants, you must store message values that *w.RegisterWindowMessage* returns in a variable so you can test for these message values in your window procedure. The coding examples up to this point have always put their *dispatch* table in a *readonly* section. If you want to be able to process a message number that you obtain from *w.RegisterWindowMessage*, you ll have to put the *dispatch* table in a static section or modify the window procedure to test the message number that w.RegisterWindowMessage returns outside the normal loop that it uses to process the dispatch table entries. E.g.,

```
static

    Dispatch:   MsgProcPtr_t; @nostorage;

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

    MyMsg: MsgProcPtr_t; @nostorage;
            MsgProcPtr_t:[  -1, &MyMsgHandler ],  // -1 is a dummy value.

            // Insert new message handler records here.

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


      .
      .
      .
    // Get a unique message value from Windows and overwrite the dummy "-1" value
    // we're currently using as MyMsg's message value:
```

```
    w.RegisterWindowMessage( "MyMsg_Message_Value" );
    mov( eax, MyMsg.MessageValue );
```

Sending a message is really nothing more than a fancy way of calling a window procedure. Windows provides many different ways to send a message to a window procedure, but a very common API function that does this is *w.SendMessage*:

```
static
    SendMessage: procedure
    (
        hWnd            :dword;
        _Msg            :dword;
        _wParam         :dword;
        _lParam         :dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__SendMessageA@16" );
```

The *hWnd* parameter specifies the handle of the window whose window procedure will receive the message. The *_Msg* parameter specifies the value of the message you want to pass to the window procedure (e.g., *w.WM_PAINT* or *w.WM_USER*). The *_wParam* and *_lParam* parameters are arbitrary 32-bit values that Windows passes on through to the window procedure; these parameters usually supply parameter data for the specific message. As you ll recall, the declaration of a window procedure takes the following form:

procedure WndProc(hwnd:dword; uMsg:uns32; wParam:dword; lParam:dword);

As you can probably tell, Windows just passes the four *w.SendMessage* parameters on through to the appropriate Window procedure.

When an application is send a message to its own window procedure, knowing the window procedures handle is no big deal; Windows returns this handle when the application calls *w.RegisterWindowEx* from the main program. The application can save this value and use it when calling the window procedure via *w.SendMessage*. However, if an application wants to send a message to some other process in the system, the application will need to know the handle value for that other process window procedure. To obtain the window handle for a window belonging to some other process in the system, an application can call the Windows *w.FindWindow* function:

```
static
    FindWindow: procedure
    (
        lpClassName     :string;
        lpWindowName    :string
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__FindWindowA@8" );
```

The *lpClassName* parameter is a string containing the window procedure s class name, that is, the string that was registered with Windows in the *w.RegisterClassEx* call[3]. If you only have one instance of a given window

3. Windows also allows you to pass a small integer value, known as an *atom*, as this parameter s value. We will not consider atoms here.

class active in the system, then you can pass NULL as the value of the second parameter. However, if you've instantiated several different copies of a window, you can use the second parameter (which is the window title string) to further differentiate the windows. If you're going to be passing messages between different processes in the system, it's a real good idea to create a unique window class for the receiving process and only run one instance of that class at a time. That's the approach this book will take; doing so allows us to simply pass NULL as the value of the second parameter to *w.FindWindow*.

The *w.FindWindow* function returns the handle of the window you specify in the EAX register, assuming that window is presently active in the system. If the window class you specify is not available in the system, then *w.FindWindow* will return NULL (zero) in the EAX register. You can use this NULL return value to determine that there is no process waiting to receive the messages you want to send it, and take corrective action, as appropriate.

As a short example, suppose that you have the System Metrics program from the previous section running on your system. You can send this application a message from a different process and tell it to resize its window using a code sequence like the following:

```
w.FindWindow( "SMWinClass", NULL );  // Get Sysmet's window handle in eax.
w.SendMessage( eax, w.WM_SIZE, 0, 480 << 16 + 640 );  // 640X480 resize operation.
```

The *w.SendMessage* API function is *synchronous*. This means that it doesn't return to the caller until the window procedure it invokes finishes whatever it does with the message and returns control back to Windows. The only time *w.SendMessage* returns without the other process completing its task is when you pass *w.SendMessage* an illegal handle number. For example, if the process containing the window procedure you're invoking has quit (or was not executing in the first place), then the handle you're passing *w.SendMessage* is invalid. Otherwise, it's up to the message destination's window procedure to retrieve and process the message before whomever calls *w.SendMessage* will continue execution. If the target of the message never retrieves the message, or hangs up while processing the message, then the application that sent the message will hang up as well.

Synchronous behavior is perfect semantics for an intra-process message (that is, a message that an application sends to itself). This is very similar to a procedure call, which most programmers are comfortable using. Unfortunately, this behavior is not entirely appropriate for inter-process calls because the activities of that other process (e.g., whether or not that other process has "hung up") have a direct impact on the execution of the current process. To write more robust message passing applications, we need to relax the semantics of a synchronous call somewhat to avoid problems.

Windows provides several different API functions you can call to send a message through the message-handling system. Some of these additional functions address the problems with *w.SendMessage* hanging up if the receiving window procedure doesn't properly process the message. Examples of these API functions include *w.PostMessage, w.SendMessageCallback*, and *w.SendMessageTimeout*. These particular API functions have the following prototypes:

```
static
    PostMessage: procedure
    (
        hWnd            :dword;
        _Msg            :dword;
        wParam          :dword;
        lParam          :dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__PostMessageA@16" );
```

```
SendMessageCallback: procedure
(
    hWnd            :dword;
    _Msg            :dword;
    _wParam         :dword;
    _lParam         :dword;
    lpCallBack      :SENDASYNCPROC;
    dwData          :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SendMessageCallbackA@24" );

SendMessageTimeout: procedure
(
        hWnd        :dword;
        _Msg        :dword;
        _wParam     :dword;
        _lParam     :dword;
        fuFlags     :dword;
        uTimeout    :dword;
    var lpdwResult  :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SendMessageTimeoutA@28" );
```

The *w.PostMessage* function sends a message to some window procedure and immediately returns to the caller without waiting for the receiver to process the message. The parameters are identical to those for *w.SendMessage*. Although *w.PostMessage* avoids the problem with an application hanging up because the recipient of the message doesn t respond, there is a major drawback to using *w.PostMessage* - the caller doesn t know whether the target window procedure has actually received and processed the message. Therefore, if the caller requires some sort of proof of completion of the message, *w.PostMessage* is not an appropriate function to call. On the other hand, if the loss of the message (or the activity that the message causes) is no big deal, then *w.PostMessage* is a safe way to communicate with another window procedure[4].

To receive confirmation that the target window procedure has processed (or has not processed) a message, without hanging up the calling process, Windows provides the *w.SendMessageCallback* function and the *w.SendMessageTimeout* function. These two functions take completely different approaches to dealing with the problems of synchronous message passing.

The *w.SendMessageCallback* API function behaves like *w.PostMessage* insofar as it immediately returns after sending the message to the target window procedure (that is, it doesn t wait for the window procedure to actually process the message). To receive notification that message processing has taken place, you pass *w.SendMessageCallback* the address of a *callback function*. Windows will call this function when the target window procedure completes execution. In the meantime, your application has continued execution. Because the call to the callback function is asynchronous, there are many issues involving concurrent programming that you must consider when using this technique. The issues of concurrent programming are a bit beyond the scope of this chapter, so we ll ignore this API function for the time being. See the Windows documentation for more details concerning the use of this function.

4. Actually, we ll see another big problem with *w.PostMessage* in the next section. But for most purposes, *w.PostMessage* is just fine if the application calling w.PostMessage can live with lost messages.

The second mechanism is to use a timeout to return control back to whomever calls *w.SendMessageTimeout* if the send message operation is taking too long (implying that the target window procedure is hung or is otherwise not going to process the message we sent it). Under normal circumstances, *w.SendMessageTimeout* behaves just like *w.SendMessage* - that is, the caller blocks (halts) until the recipient of the message processes the message and returns. The *w.SendMessageTimeout* function, however, will go ahead and return if the message s receiver does not respond within some time period (that you specify).

The *w.SendMessageTimeout* API function has three additional parameters (above and beyond those you supply to *w.SendMessage*): *fuFlags, uTimeout*, and *lpdwResult*. The *fuFlags* parameter should be one of the following values:

- *w.SMTO_ABORTIFHUNG* - This tells *w.SendMessageTimeout* to immediately return (with a timeout error) if the receiving process appears to be hung. To Windows, a process is hung if it has not attempted to read a message from its message queue during the past five seconds.
- *w.SMTO_BLOCK* - This value tells *w.SendMessageTimeout* to wait until the message returns (or times out). Windows will not send any other messages to the current process window procedure when you use this value.
- *w.SMTO_NORMAL* - The caller can process other messages that are sent to the window procedure while waiting for the *w.SendMessageTimeout* call to return.
- *w.SMTO_NOTIMEOUTIFNOTHUNG* - (Windows 2000 and later) Does not return when the time-out period expires if the process receiving the message does not appear to be hung (that is, it is still processing messages on a regular basis).

Most of the time you ll want to pass *w.SMTO_BLOCK* or *w.SMTO_NORMAL* as the *fuFlags* parameter value.

The *uTimeout* parameter specifies the timeout period in milliseconds. Choosing a time-out period can be tricky. Too long a value may cause your application to pause for extended periods of time while waiting for a hung receiver, to short a timeout period may cause Windows to prematurely return a time-out error to your program because the message s receiver hasn t gotten around to processing the message. Because Windows defines a hung program as one that doesn t respond to any messages within five seconds, a five second timeout period is probably a good starting point unless you need much faster response from *w.SendMessageTimeout*. Presumably, if the system is functioning correctly and the target process is running, you should get a response within a couple of milliseconds.

6.6.2: Memory Protection and Messages

A Windows message provides only a 64-bit data payload (i.e., the values in the *wParam* and *lParam* parameters). If someone needs to pass more data via a message, the standard convention is to pass a pointer to a data structure containing that data through the *lParam* parameter. The window procedure then casts the *lParam* parameter as a pointer to that data structure and accesses the data indirectly. By limiting the data payload to 64 bits, Windows keeps the message passing mechanism lean and efficient. Unfortunately, this scheme creates some problems when we want to pass data from one process to another via a message.

The problem with passing information between two processes under (modern versions of) Windows is that each process runs in a separate *address space*. This means that Windows sets aside a four-gigabyte memory range for each running program that is independent of the four-gigabyte address space for other executing programs. The benefit of this scheme is that one process cannot inadvertently modify code or data in another process. The drawback is that two processes cannot easily share data in memory by passing pointers between themselves. Address $40_1200 in one process will not reference the same data as address $40_1200 in a second process. Therefore, passing data directly through memory from one process to another isn t going to work.

In order to copy a large amount of data (that is, anything beyond the 64-bit data payload that *wParam* and *lParam* provide), Windows has to make a copy of the data in the source process address space in the address space of the destination process. For messages that Windows knows about (e.g., a w.WM_GETTEXT message that returns text associated with certain controls, like a text edit box), Windows handles all the details of copying data to and from buffers (typically pointed at by *lParam*) between two different processes. For user-defined messages, however, Windows has no clue whether the bit pattern in the *lParam* parameter to *w.SendMessage* is a pointer to some block of data in memory or a simple integer value. To overcome this problem, Windows provides a special message specifically geared towards passing a block of data via the *w.SendMessage* API function: the *w.WM_COPYDATA* message. A call to *w.SendMessage* passing the *w.WM_COPYDATA* message value takes the following form:

```
w.SendMessage
(
   hDestWindow,   // Handle of destination window
   w.WM_COPYDATA, // The copy data message command
   hWnd,          // Handle of the window passing the data
   cds            // (address of) a w.COPYDATASTRUCT object
);
```

The *w.COPYDATASTRUCT* data structure takes the following form:

```
type
   COPYDATASTRUCT:
      record
         dwData    :dword;  // generic 32-bit value passed to receiver.
         cbData    :uns32;  // Number of bytes in block to transfer.
         lpData    :dword;  // Pointer to block of data to transfer.
      endrecord;
```

The *dwData* field is a generic 32-bit value that Windows will pass along to the application receiving the block of data. The primary purpose of this field is to pass along a user-defined message number (or command) to the receiver. Windows only provides a single *w.WM_COPYDATA* message; if you need to send multiple messages with large data payloads to some other process, you will have to send each of these messages using the single *w.WM_COPYDATA* message. To differentiate these messages, you can use the *cds.dwData* field to hold the actual message type. If your receiving application only needs to process a single user-defined message involving a block of data, it can switch off the *w.WM_COPYDATA* message directly and use the *dwData* field for any purpose it chooses.

The *cbData* field in the *w.COPYDATASTRUCT* data type specifies the size of the data block that is being transferred from the source application to the destination window. If you re passing a zero-terminated string from the source to the target application, don t forget to include the zero terminating byte in your length (i.e., don t simply pass the string length as the *cbData* value). If you re passing some other data structure, just take the size of that structure and pass it in the *cbData* field.

The *lpData* field in the *w.COPYDATASTRUCT* object is a pointer to the block of data to copy between the applications. This block of data must not contain any pointers to objects within the source application s address space (even if those pointers reference other objects in the data block you re copying). There is no guarantee that Windows will copy the block of data to the same address in the target address space and Windows will only copy data that is part of the data block (it will not copy data that pointers within the block reference, unless that data is also within the block). Therefore, any pointers appearing in the *lpData* area will be invalid once Windows copies this data into the address space of the target window procedure.

Given the complexity and overhead associated with passing a block of data from one application to another via `w.SendMessage`, you might question why we would want to use this scheme to create the *DebugWindow* application. After all, it s perfectly possible to open up a second window in an existing application and send all of your debugging output to that window. However, the problem with sending debug messages to an application s own debug window is that if the program crashes or hangs, that application s debug window may be destroyed (so you can t see the last output messages sent to the display just prior to the crash) or the program may freeze (so you can scroll through the messages sent to the debug window). Therefore, it s much better to have all the debug messages sent to a separate application whose browsing ability isn t tied to the execution of the application you re testing.

6.6.3: Coding the *DebugWindow* Application

Like any system involving inter-process communication, the *DebugWindow* application is one of two (or more) applications that have to work together. *DebugWindow* is responsible for displaying debug and status messages that other applications send. However, for *DebugWindow* to actually do something useful, you ve actually got to have some other application send a debug message to the *DebugWindow* program. In this section we ll discuss the receiver end of this partnership (that is, the *DebugWindow* application), in the next section we ll discuss the transmitter (that is, the modifications necessary to applications that transmit debug messages).

The *DebugWindow* application itself will be a relatively straight-forward *dumb terminal emulation*. It will display text sent to it and it will buffer up some number of lines of text allowing the user to review those lines. By default, the version of *DebugWindow* we will develop in this section will save up to 1,024 lines of text, each line up to 255 characters long (though these values are easy enough to change in the source code). *DebugWindow* will make use of the vertical and horizontal scroll bars to allow the user to view all lines and columns of text displayable in the window. Once the transmitting application(s) send more than the maximum number of lines of text, *DebugWindow* will begin throwing away the oldest lines of text in the system. This keeps the program from chewing up valuable system resources (memory) if an application produces a large amount of output. The number of lines of text that *DebugWindow* remembers is controlled by the `MaxLines_c` constant appearing at the beginning of the source file:

```
const
MaxLines_c  := 1024;                // Maximum number of lines we will
                                    //  save up for review.
```

If you want to allow fewer or more lines of saved text, feel free to change this value.

DebugWindow will be capable of processing a small set of control characters embedded in the strings sent to it. Specifically, it will handle *carriage returns, line feeds*, and *tab* characters as special characters. All other characters it will physically write to the display window. Though it s fairly easy to extend *DebugWindow* to handle other control characters, there really is no need to do so. Most other control characters that have a widely respected meaning (e.g., backspace and formfeed) have a destructive nature and we don t want garbage output to the debugging window to accidentally erase any existing data (that could be used to determine why we got the garbage output).

DebugWindow ignores carriage returns and treats linefeeds as a new line character. Normal console applications under Windows use carriage return to move the cursor to the beginning of the current line. However, keeping in mind that we don t want *DebugWindow* to wipe out any existing data, we ll choose to quietly ignore carriage returns that appear in the character stream. The reason for considering carriage returns at all is because HLA s `nl` constant (newline) is the character sequence <carriage return><line feed> so it s quite likely that carriage returns will appear in the debug output because programmers are used to using the `nl` constant. Whenever a

linefeed comes along, *DebugWindow* will emit the current line of text to the next output line in the output window and move the cursor to the beginning of the next line in the output window (*DebugWindow* will also save the line in an internal memory buffer so the user can review the line later, using the vertical scroll bar).

DebugWindow will interpret tab characters to adjust the output cursor so that it moves to the next tab stop on the output line. By default, *DebugWindow* sets tab stops every eight character positions. In order for tab positions to make any sense at all, *DebugWindow* uses a monospaced font (Fixedsys) for character output. If a tab character appears in the data stream, *DebugWindow* moves the cursor to the next column that is an even multiple of eight, plus one. I.e., columns 9, 17, 25, 33, 41, etc., are tab positions. A tab character moves the cursor (blank filling) to the closest tab stop whose column number is greater than the current column position. *DebugWindow* uses a default of eight-column tab positions (this is a standard output for terminal output). You may, however, change this by modifying the *tabCols* constant at the beginning of the source file:

```
const
    tabCols     := 8;                   // Columns per tabstop position.
```

Like many of the applications you ve seen in this chapter, *DebugWindow* uses the *w.WM_CREATE* message to determine when it can do program initialization of various global objects. The initialization that the Create procedure handles includes the following:

- Initialization of the array of strings that maintain the output lines for later review by the user (i.e., sets all of these strings to the NULL pointer).
- Selection of the Fixedsys font and determining font metrics so the application knows the height and width of characters in this font (so *DebugWindows* can determine how many characters will fit in the output window as well as deal with vertical and horizontal scrolling).
- Output of an initial debug message to inform the user that *DebugWindows* is active.

Here s the code for the Create procedure that handles these tasks:

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

procedure Create( hwnd: dword; wParam:dword; lParam:dword );
readonly
    s           :string :=
                    "Debug Windows begin execution:" nl nl;

    FixedsysName:byte; @nostorage;
                 byte "Fixedsys", 0;

static
    hDC         :dword;
    hOldFont    :dword;
    Fixedsys    :w.LOGFONT;
    tm          :w.TEXTMETRIC;
    cds         :w.COPYDATASTRUCT;

begin Create;
```

```
push( edi );


// Zero out our line index, count, and the array of pointers:

xor( eax, eax );
mov( eax, StartLine );
mov( eax, LineCnt );
mov( MaxLines_c, ecx );
mov( &Lines, edi );
rep.stosd();

// Zero out the FONT structure:

mov( @size( w.LOGFONT ), ecx );
mov( &Fixedsys, edi );
rep.stosb();

// Create the font using the system's "Fixedsys" font and select
// it into the device context:

str.zcpy( FixedsysName, Fixedsys.lfFaceName );
w.CreateFontIndirect( Fixedsys );
mov( eax, FixedsysFontHandle );     // Save, so we can free this later.
GetDC( hwnd, hDC );

    // Select in the fixed system font:

    SelectObject( FixedsysFontHandle );
    mov( eax, hOldFont );

    // Determine the sizes we need for the fixed system font:

    GetTextMetrics( tm );


    mov( tm.tmHeight, eax );
    add( tm.tmExternalLeading, eax );
    inc( eax );
    mov( eax, CharHeight );

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

    SelectObject( hOldFont );

ReleaseDC;

// Just for fun, let's send ourselves a message to print the
// first line in the debug window:

mov( DebugMsg_c, cds.dwData );
mov( s, eax );
mov( eax, cds.lpData );
mov( (type str.strRec [ eax]).length, eax );
inc( eax ); // Count zero byte, too!
```

```
    mov( eax, cds.cbData );
    w.SendMessage( hwnd, w.WM_COPYDATA, hwnd, &cds );

    pop( edi );

end Create;
```

The *DebugWindow* application will process several Windows messages and one user message. Here s the dispatch table for *DebugWindow*:

```
static
    Dispatch    :MsgProcPtr_t; @nostorage;

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

            // Insert new message handler records here.

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

The user message will actually be the *w.WM_COPYDATA* system message. This is because the one user-defined message that *DebugWindow* accepts is a print message that contains a zero-terminated string to display in the debug output window. The *RcvMsg* procedure (which handles the *w.WM_COPYDATA* message) defines the three *w.COPYDATASTRUCT* fields as follows:

- *dwData* - This field contains the four bytes dbug (d is in the H.O. byte). *DebugWindow* only processes a single user message, so there is no need to use this field to differentiate messages. However, *DebugWindow* does check the value of this field just as a sanity check to avoid display garbage data.
- *cbData* - This field contains the length of the string data (including the zero byte). *DebugWindow* actually ignores this information and simply processes the string data until it encounters a zero terminating byte.
- *lpData* - This is a pointer to a zero-terminated string (note: this is *not* an HLA string!) that contains the data to display in the output window.

Note that the string at which *lpData* points isn t necessarily a single line of text. This string may contain multiple new line sequences (<carriage return><line feed>) that result in the display of several lines of text in the output window. Therefore, *DebugWindow* cannot simply copy this string to an element of the array of strings that maintain the lines of text in the display window. Instead, the *RcvMsg* procedure has to process each character in the message sent to *DebugWindows* and deal with the control characters (tabs, carriage returns, and line feeds) and construct one or more strings from the message of the text to place in the internal string list. Here s the *RcvMsg* procedure that does this processing:

```
// RcvMsg-
//
//  Receives a message from some other process and prints
// the zstring passed as the data payload for that message.
// Note that lParam points at a w.COPYDATASTRUCT object. The
```

```
// dwData field of that structure must contain DebugMsg_c if
// we are to process this message.

procedure RcvMsg( hwnd: dword; wParam:dword; lParam:dword );
var
    tabPosn :uns32;
    line    :char[ 256];

   // Note: addLine procedure actually goes here...

begin RcvMsg;

    push( esi );
    push( edi );

    // Process the incoming zero-terminated string.
    // Break it into separate lines (based on newline
    // sequences found in the string) and expand tabs.

    mov( lParam, esi );
    if( (type w.COPYDATASTRUCT [ esi]).dwData = DebugMsg_c ) then

        // Okay, we've got the w.COPYDATASTRUCT type with a valid debug
        // message. Extract the data and print it.

        mov( (type w.COPYDATASTRUCT [ esi]).lpData, esi );
        mov( 0, tabPosn );
        lea( edi, line );
        while( (type char [ esi]) <> #0 ) do

            mov( [ esi], al );

            // Ignore carriage returns:

            if( al <> stdio.cr ) then

                if( al = stdio.tab ) then

                    // Compute the number of spaces until the
                    // next tab stop position:

                    mov( tabPosn, eax );
                    cdq();
                    div( tabCols, edx:eax );
                    neg( edx );
                    add( tabCols, edx );

                    // Emit spaces up to the next tab stop:

                    repeat

                        mov( ' ', (type char [ edi]) );
                        inc( edi );
                        inc( tabPosn );
                        if( tabPosn >= 255 ) then

                            dec( edi );
```

```
                endif;
                dec( edx );

            until( @z );

        elseif( al = stdio.lf ) then

            // We've just hit a new line character.
            // Emit the line up to this point:

            mov( #0, (type char [ edi]) ); // Zero terminate.
            lea( edi, line );       // Resets edi for next loop iteration.
            str.a_cpyz( [ edi] );   // Build HLA string from zstring.
            addLine( eax );         // Add to our list of lines.

            // Reset the column counter back to zero since
            // we're starting a new line:

            mov( 0, tabPosn );

        else

            // If it's not a special control character we process,
            // then add the character to the string:

            mov( al, [ edi] );
            inc( edi );
            inc( tabPosn );

            // Don't allow more than 255 characters:

            if( tabPosn >= 255 ) then

                dec( edi );

            endif;

        endif;

    endif;

    // Move on to next character in source string:

    inc( esi );

endwhile;

// If edi is not pointing at the beginning of "line", then we've
// got some characters in the "line" string that need to be added
// to our list of lines.

lea( esi, line );
if( edi <> esi ) then

    mov( #0, (type char [ edi]) );   // Zero terminate the string.
    str.a_cpyz( [ esi] );            // Make a copy of this zstring.
```

```
            addLine( eax );

        endif;

        // Because adding and removing lines can affect the
        // maximum line length, recompute the width and horizontal
        // scroll bar stuff here:

        ComputeWidth( hwnd );

        // Ditto for the vertical scroll bars:

        ComputeHeight( hwnd );

        // Tell Windows to tell us to repaint the screen:

        w.InvalidateRect( hwnd, NULL, true );
        w.UpdateWindow( hwnd );

    endif;

    pop( edi );
    pop( esi );

end RcvMsg;
```

This procedure begins by checking the *dwData* field of the *w.COPYDATASTRUCT* parameter passed by reference via the *lParam* parameter. If this field contains the constant *DebugMsg_c* (dbug) then this procedure continues processing the data, otherwise it completely ignores the message (i.e., this is the sanity check to see if the message is valid).

If the message passes the sanity check, then the *RcvMsg* procedure begins processing the characters from the message one at a time, copying these characters to the local *line* character array if the characters aren t carriage returns, tabs, line feeds, or the zero terminating byte. If the individual character turns out to be a tab character, then *RcvMsg* expands the tab by writing the appropriate number of spaces to the *line* array. If it s a carriage return, then *RcvMsg* quietly ignores the character. If it s a line feed, then *RcvMsg* converts the characters in the *line* array to an HLA string and calls its local procedure *addLine* to add the current line of text to the list of lines that *DebugWindow* displays. If it s any other character besides a zero-terminating byte, then *RcvMsg* just appends the character to the end of the *line* array.

When *RcvMsg* encounters a zero-terminating byte it checks to see if there are any characters in the line array. If so, it converts those characters to an HLA string and adds them to the list of strings that *DebugWindow* will display. If there are no (more) characters in the input string, then RcvMsg doesn t bother creating another string. This process ensures that we don t lose any strings that consist of a sequence of characters ending with a new line sequence (on the other hand, it also means that *RcvMsg* automatically appends a new line sequence to the end of the message text if it doesn t end with this character sequence).

After adding the string(s) to the string list, the *RcvMsg* procedure calls the ComputeWidth and ComputeHeight functions (see the source code a little later for details). These functions compute the new maximum width of the file (the width of the list of strings could have gotten wider or narrowing depending on the lines we ve added or delete) and the new height information (because we ve added lines). These functions also update the horizontal and vertical scroll bars in an appropriate fashion.

The last thing that *RcvMsg* does is call the *w.InvalidateRect* and *w.UpdateWindow* API functions to force Windows to send a *w.WM_PAINT* message to the application. This causes *DebugWindow* to redraw the screen and display the text appearing in the debug window.

The *RcvMsg* procedure has a short local procedure (*addLine*) that adds HLA strings to the list of strings it maintains. To understand how this procedure operates, we need to take a quick look at the data structures that *DebugWindows* uses to keep track of the strings it has displayed. Here are the pertinent variables:

```
static
    LineAtTopOfScrn     :uns32 := 0;    // Tracks where we are in the document
    DisplayLines        :uns32 := 2;    // # of lines we can display.

    StartLine   :uns32;                 // Starting index into "Lines" array.
    LineCnt     :uns32;                 // Number of valid lines in "Lines".
    Lines       :string[ MaxLines_c] ;  // Holds the text sent to us.
```

The fundamental data structure that keeps track of the text in the debug output window is an array of strings named *Lines*. This array holds up to *MaxLines_c* strings. Once this array fills up, *DisplayWindows* reuses the oldest strings in the array to hold incoming data. The *StartLine* and *LineCnt* variables maintain this circular queue of strings. *LineCnt* specifies how many strings are present in the *Lines* array. This variable starts with the value zero when *DebugWindow* first runs and increments by one with each incoming line of text until it reaches *MaxLines_c*. Once *LineCnt* reaches *MaxLines_c* the *DebugWindows* program stops adding new lines to the array and it starts reusing the existing entries in the *lines* array. The *StartLine* variable is what *DebugWindows* uses to maintain the circular buffer of strings. *StartLine* specifies the first array element of *Lines* to begin drawing in the debug window (assuming you re scrolled all the way to the top of the document). This variable will contain zero as long as there are fewer than *MaxLines_c* lines in the array. Once *LineCnt* hits *MaxLines_c*, however, *DisplayWindow* stops adding new lines to the *Lines* array and it no longer increments the *LineCnt* variable. Instead, it will first free the storage associated with the string at *Lines[StartLine]*, it will store the new (incoming) string into *Lines[StartLine]*, and then it will increment *StartLine* by one (wrapping back to zero whenever *StartLine* reaches *MaxLines_c*). This has the effect of reusing the oldest line still in the *Lines* array and setting the second oldest line to become the new oldest line.

The *LineAtTopOfScrn* is an integer index into the *Lines* array that specifies the line that the *Paint* procedure will begin drawing at the top of the window. *DisplayLines* is a variable that holds the maximum number of lines that the program can display in the window at one time. Once the value of *LineCnt* exceeds the value of *DisplayLines*, the program will increment *LineAtTopOfScrn* with each incoming line of text so that the screen will scroll up with each incoming line (as you d normally expect for a terminal).

In addition to these variables, there are a complementary set that control horizontal scrolling. The addLine procedure, however, doesn t use them so we ll ignore them for now.

Here s the actual *addLine* procedure (which is local to the *RcvMsg* procedure):

```
    procedure addLine( lineToAdd:string in eax ); @nodisplay; @noframe;
    begin addLine;

        if( LineCnt >= MaxLines_c ) then

            mov( StartLine, ecx );
            strfree( Lines[ ecx*4 ] ); // Free the oldest line of text.
            inc( StartLine );
            if( StartLine >= MaxLines_c ) then

                mov( 0, StartLine );
```

```
            endif;

        else

            mov( LineCnt, ecx );
            inc( LineCnt );

        endif;
        mov( eax, Lines[  ecx*4 ] );

        // If we've got more than "DisplayLines" lines in the
        // output, bump "LineAtTopOfScrn" to scroll the window up:

        mov( LineCnt, ecx );
        if( ecx >= DisplayLines ) then

            inc( LineAtTopOfScrn );

        endif;
        ret();

    end addLine;
```

The DebugWindow application also processes *w.WM_SIZE*, *w.WM_VSCROLL*, and *w.WM_HSCROLL* messages. However, the logic their corresponding message handling procedures use is nearly identical to that used by the *Sysmets* application given earlier in this chapter, so we won t rehash the description of these routines. The major differences between *Sysmets* and *DebugWindow* are mainly in the horizontal scrolling code. DebugWindow uses a monospaced font, so it can more easily handle horizontal scrolling as a function of some number of characters. Another difference is that horizontal scrolling in *DebugWindows* scrolls $^1/_4$ of the screen when you do a page left or page right operation.

The *Paint* procedure in *DebugWindow* is also fairly straight-forward. A global variable (*ColAtLeftOfScrn*) tracks the amount of horizontal scrolling that takes place; *DebugWindow* uses this as an index into each line of text that it displays.

Here s the complete code to the *DebugWindow* application:

```
// DebugWindow.hla-
//
//  This program accepts "Debug Print" statements from other processes
// and displays that data in a "console-like" text window.

program DebugWindow;
#include( "stdio.hhf" )
#include( "conv.hhf" )
#include( "strings.hhf" )
#include( "memory.hhf" )
#include( "hll.hhf" )
#include( "w.hhf" )
#include( "wpa.hhf" )
#include( "excepts.hhf" )

?@NoDisplay := true;
?@NoStackAlign := true;
```

```
const
    tabCols     := 8;                   // Columns per tabstop position.
    MaxLines_c  := 1024;                // Maximum number of lines we will
                                        //  save up for review.



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

    wc                  :w.WNDCLASSEX;  // Our "window class" data.
    msg                 :w.MSG;         // Windows messages go here.
    hwnd                :dword;         // Handle to our window.
    FixedsysFontHandle  :dword;         // Save this so we can free it later.

    WM_DebugPrint       :dword;         // Message number sent to us.

    CharWidth           :dword;
    CharHeight          :dword;

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

    LineAtTopOfScrn     :uns32 := 0;    // Tracks where we are in the document
    MaxLnAtTOS          :uns32 := 0;    // Max display position (vertical).
    DisplayLines        :uns32 := 2;    // # of lines we can display.

    ColAtLeftOfScrn     :uns32 := 0;    // Current Horz position.
    MaxColAtLeft        :uns32 := 0;    // Max Horz position.
    MaxWidth            :uns32 := 40;   // Maximum columns seen thus far


    StartLine   :uns32;                 // Starting index into "Lines" array.
    LineCnt     :uns32;                 // Number of valid lines in "Lines".
    Lines       :string[ MaxLines_c] ;  // Holds the text sent to us.


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

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

    MsgProcPtr_t:
        record

            MessageValue:   dword;
            MessageHndlr:   MsgProc_t;

        endrecord;


readonly

    ClassName   :string := "DebugWindowClass";  // Window Class Name
    AppCaption  :string := "Debug Window";      // Caption for Window
```

```
    // The dispatch table:
    //
    //  This table is where you add new messages and message handlers
    //  to the program.  Each entry in the table must be a MsgProcPtr_t
    //  record containing two entries: the message value (a constant,
    //  typically one of the w.WM_***** constants found in windows.hhf)
    //  and a pointer to a "MsgProcPtr_t" procedure that will handle the
    //  message.
    //
    // Note: the RcvMsg entry *must* be first as the Create code will
    // patch the entry for the message number (this is not a constant,
    // the message number gets assigned by the system).  The current
    // message number value for this entry is bogus; it must not, however,
    // be zero.


static
    Dispatch    :MsgProcPtr_t; @nostorage;

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

            // Insert new message handler records here.

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




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

// ComputeWidth-
//
//  This procedure scans through all the lines of text we've saved
// up and finds the longest line in the list.  From this, it computes
// the maximum width we have and sets up the horizontal scroll bar
// accordingly.  This function also sets up the global variables
// MaxWidth,
// MaxColAtLeft, and
// ColAtLeftOfScrn.

procedure ComputeWidth( hwnd:dword );
begin ComputeWidth;

    push( eax );
    push( ecx );
```

```
    push( edx );

    // Need to scan through all the lines we've saved up and
    // find the maximum width of all the lines:

    mov( 0, ecx );
    for( mov( 0, edx ); edx < LineCnt; inc( edx )) do

        mov( Lines[ edx*4 ], eax );
        mov( (type str.strRec [eax]).length, eax );
        if( eax > ecx ) then

            mov( eax, ecx );

        endif;

    endfor;
    mov( ecx, MaxWidth );

    // MaxColAtLeft =
    //  max( 0, MaxWidth+1 - ClientSizeX / CharWidth);

    mov( ClientSizeX, eax );
    cdq();
    idiv( CharWidth );
    neg( eax );
    add( MaxWidth, eax );
    inc( eax );
    if( @s ) then

        xor( eax, eax );

    endif;
    mov( eax, MaxColAtLeft );

    // ColAtLeftOfScrn = min( MaxColAtLeft, ColAtLeftOfScrn )

    if( eax > ColAtLeftOfScrn ) then

        mov( ColAtLeftOfScrn, eax );

    endif;
    mov( eax, ColAtLeftOfScrn );

    w.SetScrollRange( hwnd, w.SB_HORZ, 0, MaxColAtLeft, false );
    w.SetScrollPos( hwnd, w.SB_HORZ, ColAtLeftOfScrn, true );

    pop( edx );
    pop( ecx );
    pop( eax );

end ComputeWidth;


// ComputeHeight-
//
// Computes the values for the following global variables:
```

```
//
//  DisplayLines,
//  MaxLnAtTOS, and
//  LineAtTopOfScrn
//
//  This procedure also redraws the vertical scroll bars, as necessary.

procedure ComputeHeight( hwnd:dword );
begin ComputeHeight;

    push( eax );
    push( ebx );
    push( ecx );

    // DisplayLines = ClientSizeY/CharHeight:

    mov( ClientSizeY, eax );
    cdq();
    idiv( CharHeight );
    mov( eax, DisplayLines );

    // MaxLnAtTOS = max( 0, LineCnt - DisplayLines )

    mov( LineCnt, ecx );
    sub( eax, ecx );
    if( @s ) then

        xor( ecx, ecx );

    endif;
    mov( ecx, MaxLnAtTOS );

    if( edx <> 0 ) then // EDX is remainder from ClientSizeY/CharHeight

        // If we can display a partial line, bump up the
        // DisplayLine value by one to display the partial line.

        inc( DisplayLines );

    endif;


    // LineAtTopOfScrn = min( LineAtTopOfScrn, MaxLnAtTOS )

    if( ecx > LineAtTopOfScrn ) then

        mov( LineAtTopOfScrn, ecx );

    endif;
    mov( ecx, LineAtTopOfScrn );

    w.SetScrollRange( hwnd, w.SB_VERT, 0, MaxLnAtTOS, false );
    w.SetScrollPos( hwnd, w.SB_VERT, LineAtTopOfScrn, true );

    pop( ecx );
    pop( ebx );
    pop( eax );
```

```
end ComputeHeight;



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

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

    // Free the font we created in the Create procedure:

    w.DeleteObject( FixedsysFontHandle );

    w.PostQuitMessage( 0 );

end QuitApplication;


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

procedure Create( hwnd: dword; wParam:dword; lParam:dword );
readonly
    s           :string :=
                    "Debug Windows begin execution:" nl nl;

    FixedsysName:byte; @nostorage;
                 byte "Fixedsys", 0;

static
    hDC         :dword;
    hOldFont    :dword;
    Fixedsys    :w.LOGFONT;
    tm          :w.TEXTMETRIC;
    cds         :w.COPYDATASTRUCT;

begin Create;

    push( edi );


    // Zero out our line index, count, and the array of pointers:

    xor( eax, eax );
    mov( eax, StartLine );
    mov( eax, LineCnt );
```

```
    mov( MaxLines_c, ecx );
    mov( &Lines, edi );
    rep.stosd();

    // Zero out the FONT structure:

    mov( @size( w.LOGFONT ), ecx );
    mov( &Fixedsys, edi );
    rep.stosb();

    // Create the font using the system's "Fixedsys" font and select
    // it into the device context:

    str.zcpy( FixedsysName, Fixedsys.lfFaceName );
    w.CreateFontIndirect( Fixedsys );
    mov( eax, FixedsysFontHandle );     // Save, so we can free this later.
    GetDC( hwnd, hDC );

        // Select in the fixed system font:

        SelectObject( FixedsysFontHandle );
        mov( eax, hOldFont );

        // Determine the sizes we need for the fixed system font:

        GetTextMetrics( tm );


        mov( tm.tmHeight, eax );
        add( tm.tmExternalLeading, eax );
        inc( eax );
        mov( eax, CharHeight );

        mov( tm.tmAveCharWidth, eax );
        mov( eax, CharWidth );

        SelectObject( hOldFont );

    ReleaseDC;

    // Just for fun, let's send ourselves a message to print the
    // first line in the debug window:

    mov( DebugMsg_c, cds.dwData );
    mov( s, eax );
    mov( eax, cds.lpData );
    mov( (type str.strRec [ eax]).length, eax );
    inc( eax ); // Count zero byte, too!
    mov( eax, cds.cbData );
    w.SendMessage( hwnd, w.WM_COPYDATA, hwnd, &cds );

    pop( edi );

end Create;


// RcvMsg-
```

```
//
//  Receives a message from some other process and prints
// the zstring passed as the data payload for that message.
// Note that lParam points at a w.COPYDATASTRUCT object. The
// dwData field of that structure must contain DebugMsg_c if
// we are to process this message.

procedure RcvMsg( hwnd: dword; wParam:dword; lParam:dword );
var
    tabPosn :uns32;
    line    :char[ 256];

    procedure addLine( lineToAdd:string in eax ); @nodisplay; @noframe;
    begin addLine;

        if( LineCnt >= MaxLines_c ) then

            mov( StartLine, ecx );
            strfree( Lines[ ecx*4 ] ); // Free the oldest line of text.
            inc( StartLine );
            if( StartLine >= MaxLines_c ) then

                mov( 0, StartLine );

            endif;

        else

            mov( LineCnt, ecx );
            inc( LineCnt );

        endif;
        mov( eax, Lines[ ecx*4 ] );

        // If we've got more than "DisplayLines" lines in the
        // output, bump "LineAtTopOfScrn" to scroll the window up:

        mov( LineCnt, ecx );
        if( ecx >= DisplayLines ) then

            inc( LineAtTopOfScrn );

        endif;
        ret();

    end addLine;

begin RcvMsg;

    push( esi );
    push( edi );

    // Process the incoming zero-terminated string.
    // Break it into separate lines (based on newline
    // sequences found in the string) and expand tabs.

    mov( lParam, esi );
```

```
if( (type w.COPYDATASTRUCT [ esi]).dwData = DebugMsg_c ) then

    // Okay, we've got the w.COPYDATASTRUCT type with a valid debug
    // message. Extract the data and print it.

    mov( (type w.COPYDATASTRUCT [ esi]).lpData, esi );
    mov( 0, tabPosn );
    lea( edi, line );
    while( (type char [ esi]) <> #0 ) do

        mov( [ esi], al );

        // Ignore carriage returns:

        if( al <> stdio.cr ) then

            if( al = stdio.tab ) then

                // Compute the number of spaces until the
                // next tab stop position:

                mov( tabPosn, eax );
                cdq();
                div( tabCols, edx:eax );
                neg( edx );
                add( tabCols, edx );

                // Emit spaces up to the next tab stop:

                repeat

                    mov( ' ', (type char [ edi]) );
                    inc( edi );
                    inc( tabPosn );
                    if( tabPosn >= 255 ) then

                        dec( edi );

                    endif;
                    dec( edx );

                until( @z );

            elseif( al = stdio.lf ) then

                // We've just hit a new line character.
                // Emit the line up to this point:

                mov( #0, (type char [ edi]) ); // Zero terminate.
                lea( edi, line );       // Resets edi for next loop iteration.
                str.a_cpyz( [ edi] );    // Build HLA string from zstring.
                addLine( eax );         // Add to our list of lines.

                // Reset the column counter back to zero since
                // we're starting a new line:

                mov( 0, tabPosn );
```

```
            else

                // If it's not a special control character we process,
                // then add the character to the string:

                mov( al, [edi] );
                inc( edi );
                inc( tabPosn );

                // Don't allow more than 255 characters:

                if( tabPosn >= 255 ) then

                    dec( edi );

                endif;

            endif;

        endif;

        // Move on to next character in source string:

        inc( esi );

    endwhile;

    // If edi is not pointing at the beginning of "line", then we've
    // got some characters in the "line" string that need to be added
    // to our list of lines.

    lea( esi, line );
    if( edi <> esi ) then

        mov( #0, (type char [edi]) );   // Zero terminate the string.
        str.a_cpyz( [esi] );            // Make a copy of this zstring.
        addLine( eax );

    endif;

    // Because adding and removing lines can affect the
    // maximum line length, recompute the width and horizontal
    // scroll bar stuff here:

    ComputeWidth( hwnd );

    // Ditto for the vertical scroll bars:

    ComputeHeight( hwnd );

    // Tell Windows to tell us to repaint the screen:

    w.InvalidateRect( hwnd, NULL, true );
    w.UpdateWindow( hwnd );

endif;
```

```
    pop( edi );
    pop( esi );

end RcvMsg;


// Paint:
//
//  This procedure handles the w.WM_PAINT message.
//  For this program, the Paint procedure draws all the
// lines from the scroll position to the end of the window.

procedure Paint( hwnd: dword; wParam:dword; lParam:dword );
var
    hdc         :dword;         // Handle to video display device context
    hOldFont    :dword;         // Saves old font handle.
    ps          :w.PAINTSTRUCT; // Used while painting text.

begin Paint;

    // Message handlers must preserve EBX, ESI, and EDI.
    // (They've also got to preserve EBP, but HLA's procedure
    // entry code already does that.)

    push( ebx );
    push( esi );
    push( edi );

    // When Windows requests that we draw the window,
    // fill in the string in the center of the screen.
    // Note that all GDI calls (e.g., w.DrawText) must
    // appear within a BeginPaint..EndPaint pair.

    BeginPaint( hwnd, ps, hdc );

        SelectObject( FixedsysFontHandle );
        mov( eax, hOldFont );


        // Figure out how many lines to display;
        // It's either DisplayLines (the number of lines we can
        // physically put in the Window) or LineCnt (total number
        // of lines in the Lines array), whichever is less:

        mov( DisplayLines, esi );
        if( esi > LineCnt ) then

            mov( LineCnt, esi );

        endif;

        // Display each of the lines in the Window:

        for( mov( 0, ebx ); ebx < esi; inc( ebx )) do

            // Get the string to display (address to edi):
```

```
            mov( LineAtTopOfScrn, eax );
            add( ebx, eax );
            if( eax < LineCnt ) then

                // If we've got more than a buffer full of lines,
                // base our output at "StartLine" rather than at
                // index zero:

                add( StartLine, eax );
                if( eax >= LineCnt ) then

                    sub( LineCnt, eax );

                endif;

                // Get the current line to output:

                mov( Lines[ eax*4 ], edi );

                // Compute the y-coordinate in the window:

                mov( ebx, eax );
                intmul( CharHeight, eax );

                // Compute the starting column position into the
                // line and the length of the line (to handle
                // horizontal scrolling):

                mov( (type str.strRec [edi]).length, ecx );
                add( ColAtLeftOfScrn, edi );
                sub( ColAtLeftOfScrn, ecx );

                // Output the line of text:

                TextOut
                (
                    CharWidth,
                    eax,
                    edi,
                    ecx
                );

            endif;

        endfor;
        SelectObject( hOldFont );

    EndPaint;

    pop( edi );
    pop( esi );
    pop( ebx );

end Paint;
```

```
// Size-
//
//  This procedure handles the w.WM_SIZE message
//
//  L.O. word of lParam contains the new X Size
//  H.O. word of lParam contains the new Y Size

procedure Size( hwnd: dword; wParam:dword; lParam:dword );
begin Size;

    // Convert new X size to 32 bits and save:

    movzx( (type word lParam), eax );
    mov( eax, ClientSizeX );

    // Convert new Y size to 32 bits and save:

    movzx( (type word lParam[2]), eax );
    mov( eax, ClientSizeY );

    // Compute the new screen height info here:

    ComputeHeight( hwnd );


    // Compute screen width and MaxWidth values, and set up the
    // horizontal scroll bars:

    ComputeWidth( hwnd );

    xor( eax, eax ); // return success.

end Size;



// HScroll-
//
//  Handles w.WM_HSCROLL messages.
//  On entry, L.O. word of wParam contains the scroll bar activity.

procedure HScroll( hwnd: dword; wParam:dword; lParam:dword );
begin HScroll;

    movzx( (type word wParam), eax );
    mov( ColAtLeftOfScrn, ecx );
    if( eax = w.SB_LINELEFT ) then

        // Scrolling left means decrement ColAtLeftOfScrn by one:

        dec( ecx );

    elseif( eax = w.SB_LINERIGHT ) then

        // Scrolling right means increment ColAtLeftOfScrn by one:
```

```
    inc( ecx );

elseif( eax = w.SB_PAGELEFT ) then

    // Page Left means decrement ColAtLeftOfScrn by 25% of screen width:

    mov( ClientSizeX, eax );
    cdq();
    div( CharWidth, edx:eax );  // Computes screen width in chars.
    shr( 2, eax );              // 25% of screen width.
    adc( 0, eax );
    sub( eax, ecx );

elseif( eax = w.SB_PAGERIGHT ) then

    // Page Right means increment ColAtLeftOfScrn by 25% of screen width:

    mov( ClientSizeX, eax );
    cdq();
    div( CharWidth, edx:eax );  // Computes screen width in chars.
    shr( 2, eax );              // 25% of screen width.
    adc( 0, eax );
    add( eax, ecx );


elseif( eax = w.SB_THUMBTRACK ) then

    // H.O. word of wParam contains new scroll thumb position:

    movzx( (type word wParam[2]), ecx );

// else leave ColAtLeftOfScrn alone

endif;

// Make sure that the new ColAtLeftOfScrn value (in ecx) is within
// a reasonable range (0..MaxColAtLeft):

if( (type int32 ecx) < 0 ) then

    xor( ecx, ecx );

elseif( ecx > MaxColAtLeft ) then

    mov( MaxColAtLeft, ecx );

endif;
mov( ColAtLeftOfScrn, eax );
mov( ecx, ColAtLeftOfScrn );
sub( ecx, eax );

if( eax <> 0 ) then

    intmul( CharWidth, eax );
    w.ScrollWindow( hwnd, eax, 0, NULL, NULL );
    w.SetScrollPos( hwnd, w.SB_HORZ, ColAtLeftOfScrn, true );
    w.InvalidateRect( hwnd, NULL, true );
```

```
    endif;
    xor( eax, eax ); // return success

end HScroll;



// VScroll-
//
//  Handles the w.WM_VSCROLL messages from Windows.
//  The L.O. word of wParam contains the action/command to be taken.
//  The H.O. word of wParam contains a distance for the w.SB_THUMBTRACK
//  message.

procedure VScroll( hwnd: dword; wParam:dword; lParam:dword );
begin VScroll;

    movzx( (type word wParam), eax );
    mov( LineAtTopOfScrn, ecx );
    if( eax = w.SB_TOP ) then

        // Top of file means LATOS becomes zero:

        xor( ecx, ecx );

    elseif( eax = w.SB_BOTTOM ) then

        // Bottom of file means LATOS becomes MaxLnAtTOS:

        mov( MaxLnAtTOS, ecx );

    elseif( eax = w.SB_LINEUP ) then

        // LineUp - Decrement LATOS:

        dec( ecx );

    elseif( eax = w.SB_LINEDOWN ) then

        // LineDn - Increment LATOS:

        inc( ecx );

    elseif( eax = w.SB_PAGEUP ) then

        // PgUp - Subtract DisplayLines from LATOS:

        sub( DisplayLines, ecx );

    elseif( eax = w.SB_PAGEDOWN ) then

        // PgDn - Add DisplayLines to LATOS:

        add( DisplayLines, ecx );

    elseif( eax = w.SB_THUMBTRACK ) then
```

```
        // ThumbTrack - Set LATOS to L.O. word of wParam:

        movzx( (type word wParam[2]), ecx );

    // else - leave LATOS alone

    endif;


    // Make sure LATOS is in the range 0..MaxLnAtTOS-

    if( (type int32 ecx) < 0 ) then

        xor( ecx, ecx );

    elseif( ecx >= MaxLnAtTOS ) then

        mov( MaxLnAtTOS, ecx );

    endif;
    mov( LineAtTopOfScrn, eax );
    mov( ecx, LineAtTopOfScrn );
    sub( ecx, eax );

    if( eax <> 0 ) then

        intmul( CharHeight, eax );
        w.ScrollWindow( hwnd, 0, eax, NULL, NULL );
        w.SetScrollPos( hwnd, w.SB_VERT, LineAtTopOfScrn, true );
        w.InvalidateRect( hwnd, NULL, true );
        w.UpdateWindow( hwnd );

    endif;
    xor( eax, eax ); // return success.


end VScroll;



/*****************************************************************************/
/*                    End of Application Specific Code                       */
/*****************************************************************************/




// The window procedure.
// This is actually a function that returns a return result in
// EAX.  If this function returns zero in EAX, then the event
// loop terminates program execution.

procedure WndProc( hwnd:dword; uMsg:uns32; wParam:dword; lParam:dword );
    @stdcall;

begin WndProc;
```

```
    // uMsg contains the current message Windows is passing along to
    // us.  Scan through the "Dispatch" table searching for a handler
    // for this message.  If we find one, then call the associated
    // handler procedure.  If we don't have a specific handler for this
    // message, then call the default window procedure handler function.

    mov( uMsg, eax );
    mov( &Dispatch, edx );
    forever

        mov( (type MsgProcPtr_t [ edx] ).MessageHndlr, ecx );
        if( ecx = 0 ) then

            // If an unhandled message comes along,
            // let the default window handler process the
            // message.  Whatever (non-zero) value this function
            // returns is the return result passed on to the
            // event loop.

            w.DefWindowProc( hwnd, uMsg, wParam, lParam );
            exit WndProc;


        elseif( eax = (type MsgProcPtr_t [ edx] ).MessageValue ) then

            // If the current message matches one of the values
            // in the message dispatch table, then call the
            // appropriate routine.  Note that the routine address
            // is still in ECX from the test above.

            push( hwnd );   // (type tMsgProc ecx)(hwnd, wParam, lParam)
            push( wParam ); //  This calls the associated routine after
            push( lParam ); //  pushing the necessary parameters.
            call( ecx );

            sub( eax, eax ); // Return value for function is zero.
            break;

        endif;
        add( @size( MsgProcPtr_t ), edx );

    endfor;

end WndProc;



// Here's the main program for the application.

begin DebugWindow;

    // Get this process' handle:

    w.GetModuleHandle( NULL );
    mov( eax, hInstance );
```

```
// Set up the window class (wc) object:

mov( @size( w.WNDCLASSEX ), wc.cbSize );
mov( w.CS_HREDRAW | w.CS_VREDRAW, wc.style );
mov( &WndProc, wc.lpfnWndProc );
mov( NULL, wc.cbClsExtra );
mov( NULL, wc.cbWndExtra );
mov( w.COLOR_WINDOW+1, wc.hbrBackground );
mov( NULL, wc.lpszMenuName );
mov( ClassName, wc.lpszClassName );
mov( hInstance, wc.hInstance );

// Get the icons and cursor for this application:

w.LoadIcon( NULL, val w.IDI_APPLICATION );
mov( eax, wc.hIcon );
mov( eax, wc.hIconSm );

w.LoadCursor( NULL, val w.IDC_ARROW );
mov( eax, wc.hCursor );


// Okay, register this window with Windows so it
// will start passing messages our way.  Once this
// is accomplished, create the window and display it.

w.RegisterClassEx( wc );

w.CreateWindowEx
(
    NULL,
    ClassName,
    AppCaption,
    w.WS_OVERLAPPEDWINDOW | w.WS_VSCROLL |  w.WS_HSCROLL,
    w.CW_USEDEFAULT,
    w.CW_USEDEFAULT,
    w.CW_USEDEFAULT,
    w.CW_USEDEFAULT,
    NULL,
    NULL,
    hInstance,
    NULL
);
mov( eax, hwnd );

w.ShowWindow( hwnd, w.SW_SHOWNORMAL );
w.UpdateWindow( hwnd );

// Here's the event loop that processes messages
// sent to our window.  On return from GetMessage,
// break if EAX contains false and then quit the
// program.

forever

    w.GetMessage( msg, NULL, 0, 0 );
```

```
        breakif( !eax );
        w.TranslateMessage( msg );
        w.DispatchMessage( msg );

    endfor;

    // The message handling inside Windows has stored
    // the program's return code in the wParam field
    // of the message.  Extract this and return it
    // as the program's return code.

    mov( msg.wParam, eax );
    w.ExitProcess( eax );

end DebugWindow;
```

6.6.4: Using *DebugWindow*

By itself, the *DebugWindow* application isn t very interesting. When you run it, it displays an initial debug message and then waits for another message to arrive. In order to make practical use of this application, other applications must send messages to *DebugWindow* so it can display their debug output. In this section, we ll explore the additions you ll need to make to an application in order to send output to *DebugWindow*.

Of course, the obvious way to send output to the *DebugWindow* process is to explicitly build up `w.COPYDATASTRUCT` objects and make the `w.SendMessage` call yourself. There are two problems with this approach - first, it s a lot of work. Of course, it s easy enough to write a short procedure you can pass a string that will build the data structure and call `w.SendMessage`, so this problem is easily corrected. Another problem, not quite as easy to fix, is that HLA users (particularly those coming from a console application background) expect to be able to use a statement like dbg.put(hwnd, ...); [5] to print all kinds of data, not just string data. So this is what we want to shoot for, something like the `stdout.put` or `str.put` macros that will allow a *DebugWindow* user to easily print all sorts of data.

Another issue with the debug output statements that we place in our applications is the effect they have on the application s size and performance. Although debug statements are quite useful during the debugging and testing phases of program development, we don t want these statements taking up space or slowing down the program when we produce a production version of the software. That is, we want to easily enable all these statements during debugging, but we want to be able to immediately remove them all when compiling a production version of the software. Fortunately, HLA s compile-time language provides all the facilities we need to achieve this.

To control the emission of debug code in our applications, we ll use a tried and true mechanism: *conditional assembly*. HLA s `#if` statement allows use to easily activate or eliminate sequences of statements during a compile by defining certain symbols or setting certain constants true or false. For example, consider the following code sequence:

```
#if( @defined( debug ))

   dbg.put( hwnd, "Reached Point A" nl ); // hwnd is this process' window handle

#endif
```

5. The `hwnd` parameter is the application s window handle. `dbg.put` will need this value to pass on to the *DebugWindow* process via the `wParam` parameter of the `w.SendMessage` API function.

This sequence of statements checks to see if the symbol debug is defined prior to the *#if* statement. If so, then HLA will compile the *dbg.put* statement; if not, then HLA will treat everything between the *#if* and *#endif* as a comment (and will ignore it). You may control the definition of the *debug* symbol a couple of different ways. One way is to explicitly place a statement like the following near the beginning of your program:

?debug := true;

(Note that the type and value of this symbol are irrelevant; the *@defined* compile-time function simply checks the symbol to see if it is previously defined; the convention for such symbols is to declare them as boolean constants and initialize them with true.)

Another way to define a symbol is from the HLA command line. Consider the following Windows cmd.exe command:

hla -Ddebug AppUsesDbgWin.hla

This command line will predefine the symbol *debug* as a boolean constant that is set to the value true. Note that the generic makefiles we ve created for HLA applications include a debug option that defines this symbol when you specify debug from the command line.

The only problem with using the *@defined* compile-time function to control conditional assembly is that it s an all or nothing proposition. A symbol is either defined or it is not defined in HLA. You cannot arbitrarily undefine a symbol. This means that you cannot turn the debugging code on or off on a line-by-line basis in your files. A better solution is to use a boolean compile-time variable and set it to true or false. This allows you to activate or deactivate the debug code at will throughout your source file, e.g.,

```
?debug := true;
       .
       .
       .
   << During this section of the file, debugging is active >>
       .
       .
       .
?debug := false;
       .
       .
       .
   << During this section of the file, debugging is deactivated >>
       .
       .
       .
?debug := true;
       .
       .
       .
   << Here, the debugging code is once again active >>
       .
       .
       .
```

The only catch to this approach is that you must define *debug* before the first conditional assembly statement that tests the value of debug, or HLA will report an error. An easy way to ensure that *debug* is defined is to include

code like this in the header file that contains the other definitions for the calls that output data to the debug window:

```
#if( !@defined( debug ))

   // If the symbol is not defined, default to a value of false-

   ?debug := false;

#endif
```

The combination of the definition of a symbol like *debug* and the use of conditional assembly (also called *conditional compilation*) in the program provides exactly what we need - the ability to quickly activate or deactivate the debugging code by defining (or not defining) a single symbol. There is, however, a minor problem with this approach - all those *#if* and *#endif* statements tend to clutter up the program and make it harder to read. Fortunately, there is an easy solution to this problem - make the *dbg.put* invocation a macro rather than a procedure call[6]. The *dbg.put* macro would then look something like the following:

```
namespace dbg; // This is where the "dbg." comes from...

   #macro put( hwnd, operand );
      #if( debug )
         <<Code that passes the string data to DebugWindow>>
      #endif
   #endmacro

end dbg;
```

With a macro definition like this, you can write code like the following and the system will automatically make the code associated with *dbg.put* go away if you ve not set the *debug* symbol to true:

```
dbg.put( hwnd,  Some Message  nl );
```

Of course, we d like *dbg.put* to be able to handle multiple operands and automatically convert non-string data types to their string format for output (just like *stdout.put* and *str.put*). We could copy the code for the *stdout.put* or *str.put* macros into the *dbg* namespace with the appropriate modification to construct a string to send to the *DebugWindow* application. However, this is a lot of code to copy and modify (these macros are several hundred lines long each). It turns out, however, that all we really need to do is allocate storage for a string, pass the *dbg.put* parameters to *str.put* (that will convert the data to string format), and then pass the resulting string on to the *DebugWindow* application. That is, we could get by with the following two statements:

```
   str.put( someStr, <<dbg.put parameters>> );
   sendStrToDebugWindow( hwnd, someStr );
```

The only major complication here is the need to allocate storage for a string (*someStr*) before executing these two statements. A minor issue is the fact that you cannot easily pass all the parameters from one variable-length parameter macro to another. Fortunately, with the HLA compile-time language and an understanding of the HLA string format, it s easy enough to write a macro that automates these two statements for us. Consider the following code:

---

6. It turns out, we have to do this anyway, so this is a trivial modification to make to the program.

```
namespace dbg;

    #macro put( _hwnd_dbgput_, _dbgput_parms_[] ):
            _dbgput_parmlist_,
            _cur_dbgput_parm_;

        #if( debug & @elements( _dbgput_parms_ ) <> 0 )

            sub( 4096, esp );       // Make room for string on stack.
            push( edi );            // We need a register, EDI is as good as any.
            lea( edi, [esp+12]);    // Turn EDI into a string pointer.

            // Initialize the maxlength field of our string (at [edi-8]) with 4084
            // (this string has 12 bytes of overhead).

            mov( 4084, (type str.strRec [edi]).MaxStrLen );

            // Initialize the current length to zero (empty string ):

            mov( 0, (type str.strRec [edi]).length );

            // Zero terminate the string (needed even for empty strings):

            mov( #0, (type char [edi]) );

            // Okay, invoke the str.put macro to process all the dbg.put
            // parameters passed to us. Note that we have to feed our parameters
            // to str.put one at a time:

            ?_dbgput_parmlist_ := "";
            #for( _cur_dbgput_parm_ in _dbgput_parms_ )

                ?_dbgput_parmlist_ :=
                    _dbgput_parmlist_ + "," + _cur_dbgput_parm_;

            #endfor

            str.put
            (
                (type string edi)   // Address of our destination string
                @text( _dbgput_parmlist_ )
            );  // End of str.put macro invocation
            dbg.sendStrToDebugWindow( _hwnd_dbgput_, (type string edi) );

            // Clean up the stack now that we're done:

            pop( edi );
            add( 4096, esp );

        #endif

    #endmacro

   #if( @defined( debug ))

      procedure sendStrToDebugWindow( hwnd:dword; s:string );
```

```
    begin sendStrToDebugWindow;

       << Code that sends s to DebugWindow >>

    end sendStrToDebugWindow;

  #endif

end dbg;
```

This code creates an HLA string on the stack (maximum of 4084 characters, which is probably more than enough for just about any debug message you can imagine[7]) and initializes it appropriately. Then it invokes the *str.put* macro to convert *dbg.put's* parameters to string form. Note how this code uses a compile-time *#for* loop to process all the *dbg.put* parameters (variable parameters supplied to a macro are translated to an array of strings by HLA; the *#for* loop processes each element of this array). Also note the use of the *@text* function to convert each of these strings into text for use by *str.put* (and also note the sneaky placement of the comma inside the *#for* loop so that we get exactly enough commas, in all the right places, when the *#for* loop executes during compile time). Effectively, the *#for* loop is simple injecting all of the *dbg.put* parameters into the *str.put* macro invocation after the first parameter (the destination string). This implementation of *dbg.put* requires a whole lot less code than implement it directly (i.e., by writing the same code that the *str.put* and *stdout.put* macros require).

The only thing left to implement is the *sendStrToDebugWindow* procedure. Here s the code for that procedure:

```
    #if( @global:debug )

        procedure sendStrToDebugWindow( hwnd:dword; s:string );
            @nodisplay;
            @noalignstack;
        var
            cds :@global:w.COPYDATASTRUCT;

        static
            GoodHandle :boolean := @global:true;

        begin sendStrToDebugWindow;

            push( eax );
            push( ebx );
            push( ecx );
            push( edx );

            mov( s, ebx );
            mov( (type @global:str.strRec [ ebx] ).length, eax );
            inc( eax ); // Account for zero terminating byte
            mov( ebx, cds.lpData );
            mov( eax, cds.cbData );
            mov( @global:DebugMsg_c, cds.dwData );
            @global:w.FindWindow( "DebugWindowClass", NULL );

            // Logic to bring up a dialog box complaining that
```

7. If you attempt to print more than 4084 characters with a single dbg.put macro, HLA will raise a string overflow exception.

```
        // DebugWindow is not current running if we fail to
        // find the window.  Note that we don't want to display
        // this dialog box on each execution of a dbg.put call,
        // only on the first instance where we get a w.FindWindow
        // failure.

        if( GoodHandle ) then

            if( eax = NULL ) then

                // Whenever GoodHandle goes from true to false,
                // print the following error message in a dialog box:

                @global:w.MessageBox
                (
                    0,
                    "Debug Windows is not active!",
                    "dbg.put Error",
                    @global:w.MB_OK
                );
                mov( @global:false, GoodHandle ); // Only report this once.
                xor( eax, eax );    // Set back to NULL

            endif;

        else

            // If the handle becomes good after it was bad,
            // reset the GoodHandle variable.  That way if
            // DebugWindow dies sometime later, we can once again
            // bring up the dialog box.

            if( eax <> NULL ) then

                mov( @global:true, GoodHandle );

            endif;

        endif;
        lea( ebx, cds );
        @global:w.SendMessage( eax, @global:w.WM_COPYDATA, hwnd, ebx );

        pop( edx );
        pop( ecx );
        pop( ebx );
        pop( eax );

    end sendStrToDebugWindow;

#endif
```

The *@global:* prefixes before several of the identifiers exist because these symbols are not local to the *dbg* namespace and HLA requires the *@global:* prefix when referencing symbols that are not local to the name space (this was not necessary in the macro *put* because HLA doesn t process the macro body until you actually expand the *put* macro, and usually this is outside the namespace). You ll also notice a quick check to see if *w.FindWindow* returns a valid handle. If this is the first call to *sendStrToDebugWindow* or the previous call to

w.FindWindow returned a valid handle, and the current call returns an invalid handle, then this procedure brings up a dialog box to warn the user that the *DebugWindow* application is not active. To display this dialog box, *sendStrToDebugWindow* calls the *w.MessageBox* API function:

```
static
    MessageBox: procedure
    (
        hWnd            :dword;
        lpText          :string;
        lpCaption       :string;
        uType           :dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__MessageBoxA@16" );
```

The *hWnd* parameter is the window handle for the calling procedure, the *lpText* parameter is a string that Windows displays within the dialog box, the *lpCaption* parameter is a string that Windows displays on the title bar of the dialog box, and the last parameter specifies the type of buttons to display in the dialog box (e.g., the *w.MB_OK* constant tells *w.MessageBox* to display an OK button). Note that this code doesn t display the message box on each failure by *w.FindWindow*. If the *DebugWindow* application is not running, it would be annoying to have this dialog box appear every time the application tried to send a debug message to *DebugWindow*. If the user starts up the *DebugWindow* application after the dialog box appears, then *sendStrToDebugWindow* repairs itself and begins sending the messages again.

The astute reader (and expert assembly language programmer) might notice a whole bunch of inefficiencies present in this code. For example, this code calls *w.FindWindow* for every debug message it processes. It wouldn t be too difficult to allocate a static variable inside *sendStrToDebugWindow* and initialize that variable on the very first call, saving the overhead of the call to *w.FindWindow* on future calls. Other inefficiencies include the fact that the *put* macro generates a ton of code to construct and destroy the string on each invocation of the *dbg.put* macro. Even within the *DebugWindow* application itself there are some situations where performance could be improved (e.g., this code could be smarter about how often it recomputes the maximum width and height values). On the other hand, keep in mind that the purpose of this code is to support debugging, not support a production version of an application. The minor inefficiencies present in this code are of little concern because such inefficiencies will not appear in the final code anyway.

We re not done with the *DebugWindow* application by any means. There are many, many, improvements we can make to this code. We ll revisit this application in later chapters when we cover features such as menus. At that time, we ll enhance the operation of this application to make it even more powerful and flexible.

6.7: The Windows Console API

A chapter on text manipulation in a Windows programming book isn t really complete without a brief look at the Win32 console API. Windows provides several features that let you manipulate text on the display from a console application (i.e., traditional text-based application), providing considerable control over the text display. Although the thrust of this book is to teach you how to write event-driven GUI applications under Windows, it s worth pointing out the console features that Windows provides because many applications run just fine in a text environment.

Note that, by default, HLA produces console applications when you link in and use the HLA Standard Library (e.g., when calling functions like HLA s *stdout.put* routine). In fact, HLA even provides a CONSOLE library module that provides a thin layer above the Win32 console API. The HLA Standard Library console mod-

ule, however, doesn t provide anywhere near the total capabilities that are possible with the Win32 API (HLAs console module purposefully limits its capabilities to produce a modicum of compatibility with Linux console code). We ll take a look at many of the Win32 Console API capabilities in this section.

6.7.1: Windows' Dirty Secret - GUI Apps Can Do Console I/O!

A well-kept Microsoft secret, well, at least a seldom-used feature in Windows is that every application, including GUI applications, can have a console window associated with them. Conversely, every application can open up a graphics window. In fact, the only difference between a console app and a GUI app under Windows is that console applications automatically allocate and use a console window when they begin execution, whereas GUI applications do not automatically create a console window. However, there is nothing stopping you from allocating your own console window in a GUI app. This section will demonstrate how to do that.

Creating a console window for use by a GUI app is very simple - just call the `w.AllocConsole` function. This function has the following prototype:

```
static
    AllocConsole: procedure;
        @stdcall;
        @returns( "eax" ); // Zero if failure
        @external( "__imp__AllocConsole@0" );
```

You may call `w.AllocConsole` from just about anywhere in your program that you want (generally, calling it as the first statement of your main program is a good idea). However, you may only call this function once (unless you call the corresponding `w.FreeConsole` function to deallocate the console window prior to calling `w.AllocConsole` a second time; see the Windows documentation for more details about `w.FreeConsole`).

Once you call `w.AllocConsole`, Windows attaches the standard input, standard output, and standard error devices to that console window. This means that you can use any code that writes to the standard output (or standard error) or reads from the standard input to do I/O in the new console window you ve created. Because HLAs `stdout.XXXX` and `stdin.XXXX` procedures read and write the standard input and standard output.

In the previous section, this chapter discussed the creation of the *DebugWindow* application that lets one application write debug information to a console-like text window. The *DebugWindow* application is really nice because if your main application crashes, the *DebugWindow* application is still functioning and you can scroll through the debug output your application produced to try and determine the root cause of your problem. However, if all you want to do is display some text while your program is operating normally, using an application like *DebugWindow* to capture the output of your application is overkill. A better solution is to allocate and open the console window associated with the application and then use HLAs Standard Library `stdout.XXXX` (and `stdin.XXXX`) functions to manipulate the console window. The following *ConsoleHello* program is very similar to the *DBGHello* program from the previous section, except it writes its output to the console window it opens up rather than sending this output to the *DebugWindow* application.

```
// ConsoleHello.hla:
//
// Displays "Hello World" in a window.

program ConsoleHello;
#include( "stdlib.hhf" )     // HLA Standard Library.
#include( "w.hhf" )          // Standard windows stuff.
#include( "wpa.hhf" )        // "Windows Programming in Assembly" specific stuff.
```

```
?@nodisplay := true;
?@nostackalign := true;

static
    hInstance:  dword;          // "Instance Handle" supplied by Windows.

    wc:     w.WNDCLASSEX;       // Our "window class" data.
    msg:    w.MSG;              // Windows messages go here.
    hwnd:   dword;              // Handle to our window.


readonly

    ClassName:  string := "TextAttrWinClass";       // Window Class Name
    AppCaption: string := "Text Attributes";        // Caption for Window

// The following data type and DATA declaration
// defines the message handlers for this program.

type
    MsgProc_t:  procedure( hwnd:dword; wParam:dword; lParam:dword );

    MsgProcPtr_t:
        record

            MessageValue:   dword;
            MessageHndlr:   MsgProc_t;

        endrecord;



// The dispatch table:
//
//  This table is where you add new messages and message handlers
//  to the program.  Each entry in the table must be a tMsgProcPtr
//  record containing two entries: the message value (a constant,
//  typically one of the wm.***** constants found in windows.hhf)
//  and a pointer to a "tMsgProc" procedure that will handle the
//  message.

readonly

    Dispatch:   MsgProcPtr_t; @nostorage;

        MsgProcPtr_t
            MsgProcPtr_t:[  w.WM_DESTROY, &QuitApplication   ],
            MsgProcPtr_t:[  w.WM_PAINT,   &Paint             ],
            MsgProcPtr_t:[  w.WM_CREATE,  &Create            ],
            MsgProcPtr_t:[  w.WM_SIZE,    &Size              ],

            // Insert new message handler records here.

            MsgProcPtr_t:[  0, NULL ];   // This marks the end of the list.
```

```
/*****************************************************************/
/*          A P P L I C A T I O N   S P E C I F I C   C O D E    */
/*****************************************************************/

// QuitApplication:
//
//  This procedure handles the "wm.Destroy" message.
//  It tells the application to terminate.  This code sends
//  the appropriate message to the main program's message loop
//  that will cause the application to terminate.


procedure QuitApplication( hwnd: dword; wParam:dword; lParam:dword );
begin QuitApplication;

    w.PostQuitMessage( 0 );

end QuitApplication;


// Create-
//
//  Just send a message to our console window when this message comes along:

procedure Create( hwnd: dword; wParam:dword; lParam:dword );
begin Create;

    // When the create message comes along, send a message to
    // the console window:

    stdout.put( "Create was called" nl );

end Create;



// Size-
//
//  Display the window's new size whenever this message comes along.

procedure Size( hwnd: dword; wParam:dword; lParam:dword );
begin Size;

    // When the size message comes along, send a message to
    // the console window:

    stdout.put
    (
        "Size was called, new X-size is ",
        (type uns16 lParam),
        " new Y-size is ",
        (type uns16 lParam[2]),
        nl
    );

end Size;
```

```
// Paint:
//
//  This procedure handles the "wm.Paint" message.
//  For this simple "Hello World" application, this
//  procedure simply displays "Hello World" centered in the
//  application's window.

procedure Paint( hwnd: dword; wParam:dword; lParam:dword ); @nodisplay;
var
    hdc:    dword;              // Handle to video display device context
    ps:     w.PAINTSTRUCT;      // Used while painting text.
    rect:   w.RECT;             // Used to invalidate client rectangle.

static
    PaintCnt    :uns32 := 0;

begin Paint;

    // Display a count to the console window:

    inc( PaintCnt );
    stdout.put( "Paint was called (cnt:", PaintCnt, ")" nl );

    // When Windows requests that we draw the window,
    // fill in the string in the center of the screen.
    // Note that all GDI calls (e.g., w.DrawText) must
    // appear within a BeginPaint..EndPaint pair.

    w.BeginPaint( hwnd, ps );
    mov( eax, hdc );

    w.GetClientRect( hwnd, rect );
    w.DrawText
    (
        hdc,
        "Hello World!",
        -1,
        rect,
        w.DT_SINGLELINE | w.DT_CENTER | w.DT_VCENTER
    );

    w.EndPaint( hwnd, ps );

end Paint;

/*****************************************************************************/
/*                   End of Application Specific Code                        */
/*****************************************************************************/




// The window procedure.  Since this gets called directly from
// windows we need to explicitly reverse the parameters (compared
// to the standard STDCALL declaration) in order to make HLA's
// Pascal calling convention compatible with Windows.
```

```
//
// This is actually a function that returns a return result in
// EAX.  If this function returns zero in EAX, then the event
// loop terminates program execution.

procedure WndProc( hwnd:dword; uMsg:uns32; wParam:dword; lParam:dword  );
    @stdcall;
    @nodisplay;
    @noalignstack;

begin WndProc;

    // uMsg contains the current message Windows is passing along to
    // us.  Scan through the "Dispatch" table searching for a handler
    // for this message.  If we find one, then call the associated
    // handler procedure.  If we don't have a specific handler for this
    // message, then call the default window procedure handler function.

    mov( uMsg, eax );
    mov( &Dispatch, edx );
    forever

        mov( (type MsgProcPtr_t [ edx] ).MessageHndlr, ecx );
        if( ecx = 0 ) then

            // If an unhandled message comes along,
            // let the default window handler process the
            // message.  Whatever (non-zero) value this function
            // returns is the return result passed on to the
            // event loop.

            w.DefWindowProc( hwnd, uMsg, wParam, lParam );
            exit WndProc;


        elseif( eax = (type MsgProcPtr_t [ edx] ).MessageValue ) then

            // If the current message matches one of the values
            // in the message dispatch table, then call the
            // appropriate routine.  Note that the routine address
            // is still in ECX from the test above.

            push( hwnd );   // (type tMsgProc ecx)(hwnd, wParam, lParam)
            push( wParam ); //  This calls the associated routine after
            push( lParam ); //  pushing the necessary parameters.
            call( ecx );

            sub( eax, eax ); // Return value for function is zero.
            break;

        endif;
        add( @size( MsgProcPtr_t ), edx );

    endfor;

end WndProc;
```

```
// Here's the main program for the application.

begin ConsoleHello;

    // Create a console window for this application:

    w.AllocConsole();

    // Set up the window class (wc) object:

    mov( @size( w.WNDCLASSEX ), wc.cbSize );
    mov( w.CS_HREDRAW | w.CS_VREDRAW, wc.style );
    mov( &WndProc, wc.lpfnWndProc );
    mov( NULL, wc.cbClsExtra );
    mov( NULL, wc.cbWndExtra );
    mov( w.COLOR_WINDOW+1, wc.hbrBackground );
    mov( NULL, wc.lpszMenuName );
    mov( ClassName, wc.lpszClassName );

    // Get this process' handle:

    w.GetModuleHandle( NULL );
    mov( eax, hInstance );
    mov( eax, wc.hInstance );

    // Get the icons and cursor for this application:

    w.LoadIcon( NULL, val w.IDI_APPLICATION );
    mov( eax, wc.hIcon );
    mov( eax, wc.hIconSm );

    w.LoadCursor( NULL, val w.IDC_ARROW );
    mov( eax, wc.hCursor );

    // Okay, register this window with Windows so it
    // will start passing messages our way.  Once this
    // is accomplished, create the window and display it.

    w.RegisterClassEx( wc );

    w.CreateWindowEx
    (
        NULL,
        ClassName,
        AppCaption,
        w.WS_OVERLAPPEDWINDOW,
        w.CW_USEDEFAULT,
        w.CW_USEDEFAULT,
        w.CW_USEDEFAULT,
        w.CW_USEDEFAULT,
        NULL,
        NULL,
        hInstance,
        NULL
    );
```

```
    mov( eax, hwnd );

    w.ShowWindow( hwnd, w.SW_SHOWNORMAL );
    w.UpdateWindow( hwnd );

    // Demonstrate printing to the console window:

    stdout.put( "Hello World to the console" nl );

    // Here's the event loop that processes messages
    // sent to our window.  On return from GetMessage,
    // break if EAX contains false and then quit the
    // program.

    forever

        w.GetMessage( msg, NULL, 0, 0 );
        breakif( !eax );
        w.TranslateMessage( msg );
        w.DispatchMessage( msg );

    endfor;

    // Read a newline from the console window in order to pause
    // before quitting this program:

    stdout.put( "Hit 'enter' to quit this program:" );
    stdin.readLn ();

    // The message handling inside Windows has stored
    // the program's return code in the wParam field
    // of the message.  Extract this and return it
    // as the program's return code.

    mov( msg.wParam, eax );
    w.ExitProcess( eax );

end ConsoleHello;
```

6.7.2: Win32 Console Functions

Whether you open your own console window within a GUI application or write a standard console app, Windows provides a large set of console related functions that let you manipulate the console window. Although there are far too many of these functions to describe in detail here, this section will attempt to describe the more useful functions available to console window programmers. For full details on the console window functions, check out the Microsoft document (e.g., on MSDN).

6.7.2.1: w.AllocConsole

```
static
    AllocConsole: procedure;
        @stdcall;
```

```
        @returns( "eax" ); // Zero if failure
        @external( "__imp__AllocConsole@0" );
```

The *w.AllocConsole* function will allocate a console for the current application. This function returns zero if the call fails, it returns a non-zero result if the function succeeds. An application may only have one allocated console associated with it. See the description of the *w.FreeConsole* function to learn how to deallocate a console window (so you can allocate another one with a second call to *w.AllocConsole*).

6.7.2.2: w.CreateConsoleScreenBuffer

```
static
    CreateConsoleScreenBuffer: procedure
    (
            dwDesiredAccess:        dword;
            dwShareMode:            dword;
        var lpSecurityAttributes:   SECURITY_ATTRIBUTES;
            dwFlags:                dword;
            lpScreenBufferData:     dword  // Should be NULL.
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__CreateConsoleScreenBuffer@20" );
```

A console window can have multiple screen buffers associated with it, though only one screen buffer can be active at a time (see the description of *w.SetConsoleActiveScreenBuffer* for details on switching console buffers). By default, a console window as a single console buffer associated with it when you first create a console window (via *w.AllocConsole*). However, you may create additional console buffers by calling the *w.CreateConsoleScreenBuffer* API function.

The *dwDesiredAccess* parameter specifies the access rights the current process will have to the buffer. This should be the constant *w.GENERIC_READ | w.GENERIC_WRITE* for full access to the buffer (at the very least, this parameter should have *w.GENERIC_WRITE*, but read access is useful, too).

In general, the *dwShareMode* parameter should be zero. See the Microsoft documentation for more details about file sharing. File sharing isn t a particularly useful feature in console windows, so you ll normally set this parameter to zero.

The *lpSecurityAttribute* parameter should be set to NULL. See the Microsoft documentation if you want to implement secure access to a console window (not very useful, quite frankly).

The *dwFlags* parameter should be set to the value *w.CONSOLE_TEXTMODE_BUFFER* (this is currently the only legal value you can pass as this parameter s value).

The *lpScreenBufferData* parameter must be passed as NULL. Undoubtedly, Microsoft originally intended to use this parameter for something else, and then changed their mind about it s purpose.

If this function fails, it returns zero in the EAX register. If it succeeds, it returns a handle to the console buffer that Windows creates. You will need to save this handle so you can use it to switch to this console buffer using the *w.SetConsoleActiveScreenBuffer* API function.

6.7.2.3: w.FillConsoleOutputAttribute

```
static
    FillConsoleOutputAttribute: procedure
```

```
    (
            hConsoleOutput :dword; // Handle to screen buffer
            wAttribute     :dword; // Color attribute (in L.O. 16 bits)
            nLength        :dword; // Number of character positions to fill
            dwWriteCoord   :w.COORD; // Coordinates of first position to fill
        var lpNumberOfAttrsWritten :dword //Returns # of positions written here
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__FillConsoleOutputAttribute@20" );

const
    FOREGROUND_BLUE := $1;
    FOREGROUND_GREEN := $2;
    FOREGROUND_RED := $4;
    FOREGROUND_INTENSITY := $8;
    BACKGROUND_BLUE := $10;
    BACKGROUND_GREEN := $20;
    BACKGROUND_RED := $40;
    BACKGROUND_INTENSITY := $80;

type
    COORD:
        record
            x: word;
            y: word;
        endrecord;
```

A *character attribute* in a console window is the combination of foreground and background colors for that character cell. The *w.FillConsoleOutputAttribute* function lets you set the foreground and background colors for a sequential set of characters on the screen without otherwise affecting the characters on the screen.

The *hConsoleOutput* parameter is the handle of a screen buffer (e.g., a handle returned by *w.CreateConsoleScreenBuffer* or by a call to *w.GetStdHandle(w.STD_OUTPUT_HANDLE)*).

The *wAttribute* value is a bitmap containing some combination of the foreground and background colors defined in the *windows.hhf* header file (included by *w.hhf*). The base foreground colors are *w.FOREGROUND_BLUE*, *w.FOREGROUND_GREEN*, and *w.FOREGROUND_RED*. OR ing these constants together forms additional colors, i.e.,

- *w.FOREGROUND_GREEN | w.FOREGROUND_BLUE* produces cyan.
- *w.FOREGROUND_BLUE | w.FOREGROUND_RED* produces magenta.
- *w.FOREGROUND_GREEN | w.FOREGROUND_RED* produces yellow.
- *w.FOREGROUND_GREEN | w.FOREGROUND_BLUE | w.FOREGROUND_REG* produces white.

Merging in w.FOREGROUND_INTENSITY with any of the individual colors or combination of colors produces a lighter (brighter) version of that color. Because you may combine four bits in any combination, it is possible to specify up to 16 different foreground colors with these values.

You set the background colors exactly the same way, except (of course) you use the *w.BACKGROUND_xxx* constants rather than the *w.FOREGROUND_xxx* constants. Note that you merge both the foreground and background colors into the same *wAttribute* parameter value. Also note that this bitmap consumes only eight of the 32 bits in the *wAttribute* parameter - the other bits must be zero.

The *nLength* parameter specifies the number of character positions to fill with the specified attribute. If this length would specify a character position beyond the end of the current line, then Windows will wrap around to the beginning of the next line and continue filling the attributes there. If the *nLength* parameter, along with the *dwWriteCoord* parameter, specifies a position beyond the end of the current screen buffer, Windows stops emitting attribute values at the end of the current screen buffer.

The *dwWriteCoord* parameter specifies the starting coordinate in the screen buffer to begin the attribute fill operation. Note that this is a record object consisting of a 16-bit X-coordinate and 16-bit Y-coordinate value pair. Generally, most people simply pass this parameter as a 32-bit value with the X-coordinate value appearing in the L.O. 32 bits of the double word and the Y-coordinate appearing in the upper 32-bits of the double word. You can use HLA's type coercion operation to pass a 32-bit register as this parameter value, e.g.,

```
mov( $1_0002, edx );  // x=2, y=1
w.FillConsoleOutputAttribute
(
   hOutputWindow,
   w.BACKGROUND_GREEN | w.BACKKGROUND_BLUE | w.BACKGROUND_INTENSITY,
   (type w.COORD edx),
   charsWritten
);
```

The last parameter, *lpNumberOfAttrsWritten*, is a pointer to an integer variable that receives the total number of attribute positions modified in the screen buffer by this function. The *w.FillConsoleOutputAttr* function stores a value different from *nLength* into the variable referenced by this address if *nLength* specifies output positions beyond the end of the screen buffer (forcing *w.FillConsoleOutputAttr* to write fewer than *nLength* attribute values to the screen buffer).

You would normally use this function to change the attributes (colors) of existing text on the display; you would not normally use this function to write new data to the screen (the *w.SetConsoleTextAttribute* function lets you set the default character attributes to use when writing characters to the console). You could use this function, for example, to highlight a line of existing text on the display (perhaps a menu selection) without otherwise affecting the text data.

The *w.FillConsoleOutputAttribute* function writes only a single foreground/background color combination to the console. If you want to write a sequence of different attribute values to the display, you should consider using the *w.WriteConsoleOutputAttribute* function, instead.

6.7.2.4: w.FillConsoleOutputCharacter

```
static
    FillConsoleOutputCharacter: procedure
    (
        hConsoleOutput:             dword;
        cCharacter:                 char;
        nLength:                    dword;
        dwWriteCoord:               COORD;
        var lpNumberOfAttrsWritten: dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__FillConsoleOutputCharacterA@20" );
```

The *w.FillConsoleOutputCharacter* function fills a sequence of character cells on the console with a single character value. This function, for example, is useful for clearing a portion of the screen by writing a sequence of space characters (or some other background fill character). This function does not affect the attribute values at each character cell location; therefore, the characters that this function writes to the console will retain the foreground/background colors of the previous characters in those same character cell positions.

This function writes *nLength* copies of the character *cCharacter* to the console screen buffer specified by *hConsoleOutput* starting at the (x,y) coordinate position specified by *dwWriteCoord*. If *nLength* is sufficiently large that it will write characters beyond the end of a line, *w.FillConsoleOutputCharacter* will wrap around to the beginning of the next line. If *w.FillConsoleOutputCharacter* would write characters beyond the end of the screen buffer, then this function ignores the write request for those characters beyond the buffer s end.

6.7.2.5: w.FlushConsoleInputBuffer

```
static
    FlushConsoleInputBuffer: procedure
    (
        hConsoleInput:  dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__FlushConsoleInputBuffer@4" );
```

A call to this function flushes all the input events from the input buffer. Input events include key presses, mouse events, and window resize events. Most console programs ignore mouse and resize events; such programs would call this function to flush any keystrokes from the type ahead buffer.

The *hConsoleInput* handle value is the handle to the console s input buffer. You may obtain the current console input buffer s handle using the API call *w.GetStdHandle(w.STD_INPUT_HANDLE);*

6.7.2.6: w.FreeConsole

```
static
    FreeConsole: procedure;
        @stdcall;
        @returns( "eax" );
        @external( "__imp__FreeConsole@0" );
```

This API call frees (and detaches) the console window associated with the current process (this also closes that console window). Because you can only have one console window associated with an application, you must call this function if you want to create a new window via a *w.AllocConsole* call if the application already has a console window associated with it. Because Windows automatically frees the console window when an application terminates and you can only have one console window active at a time, few programs actually call this function.

6.7.2.7: w.GetConsoleCursorInfo

```
static
    GetConsoleCursorInfo: procedure
    (
            hConsoleOutput:         dword;
```

```
        var lpConsoleCursorInfo:    CONSOLE_CURSOR_INFO
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__GetConsoleCursorInfo@8" );

type
    CONSOLE_CURSOR_INFO:
        record
            dwSize: dword;
            bVisible: dword;
        endrecord;
```

This function returns the size and visibility of the console cursor in a *w.CONSOLE_CURSOR_INFO* data structure. The *dwSize* field of the record is a value in the range 1..100 and represents the percentage of the character cell that the cursor fills. This ranges from an underline at the bottom of the character cell to filling up the entire character cell (at 100%). The *hConsoleOutput* parameter must be the handle of the console s output buffer.

6.7.2.8: w.GetConsoleScreenBufferInfo

```
static
    GetConsoleScreenBufferInfo: procedure
    (
            handle: dword;
        var csbi:   CONSOLE_SCREEN_BUFFER_INFO
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__GetConsoleScreenBufferInfo@8" );

const
    FOREGROUND_BLUE := $1;
    FOREGROUND_GREEN := $2;
    FOREGROUND_RED := $4;
    FOREGROUND_INTENSITY := $8;
    BACKGROUND_BLUE := $10;
    BACKGROUND_GREEN := $20;
    BACKGROUND_RED := $40;
    BACKGROUND_INTENSITY := $80;

type
    CONSOLE_SCREEN_BUFFER_INFO:
        record
            dwSize: COORD;
            dwCursorPosition: COORD;
            wAttributes: word;
            srWindow: SMALL_RECT;
            dwMaximumWindowSize: COORD;
        endrecord;

    SMALL_RECT:
        record
            Left: word;
            Top: word;
            Right: word;
```

```
            Bottom: word;
        endrecord;

    COORD:
        record
            x: word;
            y: word;
        endrecord;
```

w.GetConsoleScreenBufferInfo retrieves information about the size of the screen buffer, the position of the cursor in the window, and the scrollable region of the console window. The *handle* parameter is the handle of the console screen buffer, the *csbi* parameter is where this function returns its data.

The *dwSize* field in the *w.CONSOLE_SCREEN_BUFFER* data structure specifies the size of the console screen buffer. Remember, these values specify the size of the buffer, not the physical size of the console window. Usually, the buffer is larger than the physical window. The *dwSize.x* field specifies the width of the screen buffer, the *dwSize.y* field specifies the height of the screen buffer. These values are in character cell coordinates.

The *dwCursorPosition* field specifies the (x,y) coordinate of the cursor within the screen buffer. Remember, this is the position of the cursor within the screen buffer, not in the console window.

The *wAttribute* field specifies the default foreground and background colors that Windows will use when you write a character to the display. This field may contain any combination of the *w.FOREGROUND_xxx* and *w.BACKGROUND_xxx* character constants. See the discussion of output attributes in the section on *w.FillConsoleOutputAttribute* for more details.

The *srWindow* field is a pair of coordinates that specify the position of the upper left hand corner and the lower right hand corner of the display window within the screen buffer. Note that you can scroll the screen buffer through the display window or even resize the display window by changing the *srWindow* coordinate values and calling *w.SetConsoleScreenBufferInfo*.

The *dwMaximumWindowSize* field specifies the maximum size of the console display window based on the current font in use, the physical screen size, and the screen buffer size.

6.7.2.9: w.GetConsoleTitle

```
static
    GetConsoleTitle: procedure
    (
        var lpConsoleTitle: var;
            nSize:          dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__GetConsoleTitleA@8" );
```

The *w.GetConsoleTitle* function returns a string containing the name found in the console window s title bar. The *lpConsoletitle* parameter must be the address of a buffer that will receive a zero-terminated string holding the window s title. The *nSize* parameter specifies the maximum number of characters (including the zero byte) that *w.GetConsoleTitle* will transfer. This function returns the actual length of the string (not counting the zero-terminating byte) in the EAX register.

There are two things to note about the *lpConsoleTitle* parameter. First of all, this is not the address of an HLA string, but the address of a buffer that will receive a zero-terminated string. Second, this is an untyped reference parameter, so HLA will automatically compute the address of whatever object you pass as this parameter.

So if you pass a pointer variable (e.g., a string object), HLA will compute the address of the pointer, not the address of the buffer that the pointer references. You may override this behavior by using the VAL keyword to tell HLA to use the value of the pointer rather than its address. If you want this function to fill in an HLA string variable, you could use code like the following:

```
mov( stringToUse, edx );
w.GetConsoleTitle( [edx], (type str.strRec [edx]).MaxStrLen ) );
mov( stringToUse, edx ); // Windows' API calls don't preserve EDX!
mov( eax, (type str.strRec [edx]).length );
```

This code loads EDX with the address of the first byte of the character data buffer in the string referenced by *stringToUse*. This code passes the address of this character buffer in EDX to the function (brackets are necessary around EDX to tell HLA that this is a memory address rather than just a register value). This function passes the string s maximum length field as the buffer size to the *w.GetConsoleTitle* function.

On return, this function stores the actual string length (returned in EAX) into the HLA string s string length field. The end result is that stringToUse now contains valid HLA string data (the console window s title string).

6.7.2.10: w.GetConsoleWindow

```
static
    GetConsoleWindow: procedure;
        @stdcall;
        @returns( "eax" );
        @external( "__imp__GetConsoleWindow@0" );
```

The *w.GetConsoleWindow* function returns the window handle associated with the application s console window. Note that there is a big difference between a console screen buffer handle and a console window handle. The window handle is the same type of handle that GUI apps use when painting to the screen. Screen buffer handles are the handles you pass to console function to manipulate data in the screen buffer.

6.7.2.11: w.GetNumberOfConsoleInputEvents

```
static
    GetNumberOfConsoleInputEvents: procedure
    (
            hConsoleInput:      dword;
        var lpcNumberOfEvents:  dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__GetNumberOfConsoleInputEvents@8" );
```

The *w.GetNumberOfConsoleInputEvents* returns the total number of pending keystroke, mouse, or resize input events sent to the console window. This function stores the count at the address specified by the *lpcNumberOfEvents* parameter (note that this function returns success/failure in EAX, it does not return the count in EAX). There is, unfortunately, no way to directly check to see if there are any keystrokes sitting in the input queue. However, if w.GetNumberOfConsoleInputEvents tells you that there are pending input events, you can use w.PeekConsoleInput to determine if the input event at the front of the queue is keystroke waiting to be read. If not, you can call w.ReadConsolInput to remove that first input event from the queue. You can repeat the get-

number/peek/read sequence until you empty the input buffer or you peek at a keyboard event. By this processes, you can determine if there is at least one keyboard input event in the queue.

6.7.2.12: w.GetStdHandle

```
static
    GetStdHandle: procedure
    (
        nStdHandle:dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__GetStdHandle@4" );
const
    STD_INPUT_HANDLE := -10;
    STD_OUTPUT_HANDLE := -11;
    STD_ERROR_HANDLE := -12;
```

This function returns the current handle for the standard input, standard output, or standard error devices. The single parameter (*nStdHandle*) is one of the values *w.STD_INPUT_HANDLE, w.STD_OUTPUT_HANDLE,* or *w.STD_ERROR_HANDLE,* depending on which of these functions you want to retrieve. By default (meaning you ve not redirected the standard input, standard output, or standard error devices), the standard output and standard error handles will be the same as the output screen buffer handle. The standard input, by default, will be the handle of the console s input buffer.

6.7.2.13: w.PeekConsoleInput

```
static
    PeekConsoleInput: procedure
    (
            hConsoleInput:          dword;
        var lpBuffer:               INPUT_RECORD;
            nLength:                dword;
        var lpNumberOfEventsRead:   dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__PeekConsoleInputA@16" );

const
    KEY_EVENT := $1;
    mouse_eventC := $2;
    WINDOW_BUFFER_SIZE_EVENT := $4;
    MENU_EVENT := $8;
    FOCUS_EVENT := $10;

type
    INPUT_RECORD:
        record
            EventType:word;
            Event:
                union
                    KeyEvent                :KEY_EVENT_RECORD;
```

```
                MouseEvent              :MOUSE_EVENT_RECORD;
                WindowBufferSizeEvent   :WINDOW_BUFFER_SIZE_RECORD;
                FocusEvent              :FOCUS_EVENT_RECORD;
            endunion;
    endrecord;

KEY_EVENT_RECORD:
    record
        bKeyDown: dword;
        wRepeatCount: word;
        wVirtualKeyCode: word;
        wVirtualScanCode: word;
        _uChar: CHARTYPE;
        dwControlKeyState: dword;
    endrecord;

MOUSE_EVENT_RECORD:
    record
        dwMousePosition: COORD;
        dwButtonState: dword;
        dwControlKeyState: dword;
        dwEventFlags: dword;
    endrecord;

 WINDOW_BUFFER_SIZE_RECORD:
    record
        dwSize: COORD;
   endrecord;

FOCUS_EVENT_RECORD:
    record
        bSetFocus: dword;
    endrecord;
```

The *w.PeekConsoleInput* function reads one or more input records (if available) from the console s input queue without removing those records from the input queue. You may use the *w.GetNumberOfConsoleInputEvents* to determine how many pending events there are, allocate an array of *w.INPUT_RECORD* objects with the number of elements returned by *w.GetNumberOfConsoleInputEvents*, and then call *w.PeekConsoleInput* to read all of those events into the array you ve allocated. Then you can scan through the array at your leisure to determine what type of input records are in the buffer.

The *hConsoleInput* parameter is the handle of the console input buffer you want to peek at. The *lpBuffer* parameter is a pointer to one or more *w.INPUT_RECORD* objects that will hold the returned records; note that you may specify a pointer, a scalar variable, or an array variable here, HLA will do the right thing with your parameter. The *nLength* parameter specifies the number of array elements you re passing via the *lpBuffer* parameter. If you re only passing a scalar *w.INPUT_BUFFER* object as the *lpBuffer* parameter, you should specify an *nLength* value of one. Note that the *nLength* parameter specifies the maximum number of input records that *w.PeekConsoleInput* will read; if there are fewer records than this number specifies, *w.PeekConsoleInput* will only read those records that are available. The *lpNumberOfEventsRead* parameter is a pointer to a 32-bit integer variable where *w.PeekConsoleInput* will store the actual number of events read from the input queue. This value will always be less than or equal to the value you pass in nLength. Note that it is perfectly possible (and reasonable) for *w.PeekConsoleInput* to read zero input records (if there are no pending input records) and return zero in the variable referenced by *lpNumberOfEventsRead*.

The *lpBuffer* parameter to *w.PeekConsoleInput* must contain the address of a *w.INPUT_EVENT* record. The first field of this record is the *EventType* field that specifies the type of the event that the record holds. This field will contain one of the following values: *w.KEY_EVENT, w.mouse_eventC, w.WINDOW_BUFFER_SIZE_EVENT, w.MENU_EVENT,* or *w.FOCUS_EVENT*. Yes, *w.mouse_eventC* is spelled funny and doesn t match the Window s documentation. This is one of those rare cases where Windows reused an identifier by simply changing the case of the identifier, thus creating a case conflict in HLA. The solution was to rename Windows s constant *MOUSE_EVENT* as *w.mouse_eventC*. The value of *EventType* determines whether the *Event* union s data is a *w.KEY_EVENT_RECORD, w.MOUSE_EVENT_RECORD, w.WINDOW_BUFFER_SIZE_RECORD*, or a *w.FOCUS_EVENT_RECORD* type.

If *Event* contains the value *w.KEY_EVENT* then the *KeyEvent* field of the *w.INPUT_RECORD* union contains data representing a keyboard event. The fields of the *w.KEY_EVENT_RECORD* data structure have the following meanings:

- *bKeyDown* contains true (non-zero value) if the key was just pressed, false (0) if it was just released.
- *wRepeatCount* may contain a repeat count for the key saying that the key auto-repeated this many times. You must treat this as n keypresses of the current key code.
- *wVirtualKeyCode* specifies a Windows virtual keycode for this particular keypress. See the appendices for a list of the Windows virtual keycodes.
- wVirtualScanCode specifies an OEM/Hardware dependent scan code for this particular keystroke. Most applications should ignore this value.
- *uChar* is the translated ASCII/ANSI character code for this keystroke. Keystrokes that do not have ANSI character code equivalents generally return zero here and you have to use the *wVirtualKeyCode* field to determine which key was pressed (or released).
- *dwControlKeyState* reports the state of the modifier keys when this key event occurred. This field is a bit map with a set bit indicating that a particular modifier key (e.g., control or shift) was held down. Here are the values associated with the various modifier keys on a PC s keyboard:

```
w.RIGHT_ALT_PRESSED     $1
w.LEFT_ALT_PRESSED      $2
w.RIGHT_CTRL_PRESSED    $4
w.LEFT_CTRL_PRESSED     $8
w.SHIFT_PRESSED         $10
w.NUMLOCK_ON            $20
w.SCROLLLOCK_ON         $40
w.CAPSLOCK_ON           $80
w.ENHANCED_KEY          $100
```

The *w.ENHANCED_KEY* modifier specifies that the user has pressed an enhanced key on a keyboard. Enhanced keys are the INS, DEL, HOME, END, PAGEUP, PAGE DN, and arrow keys that are separate from the numeric keypad. The slash (/) and ENTER keys on the numeric keypad are also enhanced keys.

If Event contains the value *w.MOUSE_EVENT* then the *KeyEvent* field of the *w.INPUT_RECORD* union contains data representing a mouse event. The fields of the *w.MOUSE_EVENT_RECORD* data structure have the following meanings:

- *dwMousePosition* specifies the location of the mouse cursor in screen buffer character cell coordinates. Note that the Y-coordinate appears in the H.O. word and the X-coordinate appears in the L.O. word of the *w.COORD* data type.
- *dwButtonState* is a bitmap specifying the state of the buttons on the mouse. The w.hhf header files defines the following constants specifying button positions with in the bitmap:

```
w.FROM_LEFT_1ST_BUTTON_PRESSED  $1
w.RIGHTMOST_BUTTON_PRESSED      $2
w.FROM_LEFT_2ND_BUTTON_PRESSED  $4
w.FROM_LEFT_3RD_BUTTON_PRESSED  $8
w.FROM_LEFT_4TH_BUTTON_PRESSED  $10
```

If a mouse supports more than five buttons, then the mouse driver will, undoubtedly, set additional H.O. bits in this bitmap.

- *dwControlKeyState* is a bitmap specifying the current state of the modifier keys on the keyboard when this mouse event occurred. See the description of *dwControlKeyState* in the description of *w.KEY_EVENT_RECORD* for more details.
- dwEventFlags specifies the type of mouse event that has occurred. If this field contains zero, it means that a mouse button has been pressed or released. If this field is not zero, it contains one of the following values:

```
w.MOUSE_MOVED    $1
w.DOUBLE_CLICK   $2
```

Console applications should ignore *w.WINDOW_BUFFER_SIZE_RECORD* and *w.FOCUS_EVENT* as these are intended for internal use by Windows.

6.7.2.14: w.ReadConsole

```
static
    ReadConsole: procedure
    (
            hConsoleInput:          dword;
        var lpBuffer:               var;
            nNumberOfCharsToRead:   dword;
        var lpNumberOfCharsRead:    dword;
        var lpReserved:             var
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__ReadConsoleA@20" );
```

The *w.ReadConsole* function reads keyboard events from the console input buffer. The *hConsoleInput* parameter specifies the console input buffer to read from (this should be the standard input handle). The *lpBuffer* parameter is the address of a character array (buffer) that will receive the characters read from the console input buffer. The *nNumberOfCharsToRead* parameter specifies the number of keystrokes that will be read from the input buffer. By default, *w.ReadConsole* will return when the user types this many characters at the keyboard or when the user presses the enter key (prior to typing this many characters). Please see the Windows documentation for details on the *w.GetConsoleMode* and *w.SetConsoleMode* functions that let you specify Windows behavior with respect to console input. the *lpNumberOfCharactersRead* is a pointer to an integer variable; Windows will store the actual number of characters read from the keyboard into this integer variable. The *lpReserved* field is reserved and you must pass NULL in this parameter.

If the input buffer contains records other than keyboard events, *w.ReadConsole* removes those events from the buffer and ignores them. If you want to be able to read mouse events, use *w.ReadConsoleInput* instead of *w.ReadConsole*. Also note that *w.ReadConsole* is virtually the same as the *w.ReadFile* function. The pure Windows version of *w.ReadConsole* allows you to read Unicode characters, but the HLA prototype for *w.Read-*

Console only reads ANSI characters (there is nothing stopping you from creating your own prototype to call the actual Windows *ReadConsole* API function, though).

6.7.2.15: w.ReadConsoleInput

```
static
    ReadConsoleInput: procedure
    (
            hConsoleInput:          dword;
        var lpBuffer:               INPUT_RECORD;
            nLength:                dword;
        var lpNumberOfEventsRead:   dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__ReadConsoleInputA@16" );
```

This function reads console input records from the console input buffer. This reads both keyboard and mouse events and stores them into a *w.INPUT_RECORD* array whose address you pass to this function.

The *hConsoleInput* parameter is the console input handle (e.g., the standard input handle). The *lpBuffer* parameter is the address of a *w.INPUT_RECORD* structure or an array of these structures (see the description of this data structure in the section on *w.PeekConsoleInput*). The *nLength* parameter specifies the number of input records to read from the console input queue. As with *w.ReadConsole*, Windows may actually read fewer than this number of input records if the console mode is set to return when the user presses the Enter key. In any case, *w.ReadConsoleInput* will store the actual number of input records at the address specified by the *lpNumberOfEventsRead* parameter. See the description of *w.ReadConsole* for more details. Also see the discussion of *w.PeekConsoleInput* for a discussion of the *w.INPUT_RECORD* type.

6.7.2.16: w.ReadConsoleOutput

```
static
    ReadConsoleOutput: procedure
    (
            hConsoleOutput: dword;
        var lpBuffer:       CHAR_INFO;
            dwBufferSize:   COORD;
            dwBufferCoord:  COORD;
        var lpReadRegion:   SMALL_RECT
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__ReadConsoleOutputA@20" );

type
    COORD:
        record
            x: word;
            y: word;
        endrecord;

    SMALL_RECT:
        record
```

```
            Left    :word;
            Top     :word;
            Right   :word;
            Bottom  :word;
         endrecord;

    CHARTYPE:
        union
            UnicodeChar :word;  // Note: HLA's prototype only returns ASCII characters.
            AsciiChar   :byte;
        endunion;

    CHAR_INFO:
        record
            CharVal    :CHARTYPE;
            Attributes :word;
        endrecord;
```

The *w.ReadConsoleOutput* function reads a rectangular region off the screen and stores the character data into a two-dimensional array of characters. The *hConsoleOutput* parameter is the handle of the console s output screen buffer (i.e., the standard output handle). The *lpBuffer* parameter is a pointer to a two-dimensional array of characters that will receive the character data read from the screen buffer. The *dwBufferSize* parameter specifies the size of the buffer that will receive the characters; the L.O. word (*x*) specifies the number of columns in the array, the H.O. word (*y*) specifies the number of rows in the array. The *dwBufferCoord* parameter specifies the (x,y) coordinates (in character cell positions) of the upper-leftmost character in the array specified by *lpBuffer* where *w.ReadConsoleOutput* will begin storing the character data. On entry, the values pointed at by the *lpReadRegion* parameter specify the upper left hand corner and the lower right hand corner of a rectangular region in the screen buffer to copy into the buffer specified by *lpBuffer*. On exit, the *w.ReadConsoleOutput* routine stores the actual coordinates used by the copy operation. The input and output values will be different if the input values exceeded the boundaries of the screen buffer (in which case, *w.ReadConsoleOutput* truncates the coordinates so that they remain within the screen buffer).

6.7.2.17: w.ReadConsoleOutputAttribute

```
static
    ReadConsoleOutputAttribute: procedure
    (
            hConsoleOutput:         dword;
        var lpAttribute:            word;
            nLength:                dword;
            dwReadCoord:            COORD;
        var lpNumberOfAttrsRead:    dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__ReadConsoleOutputAttribute@20" );

type
    COORD:
        record
            x: word;
            y: word;
        endrecord;
```

This function is very similar to *w.ReadConsoleOutput* except it copies attribute values rather than characters to the *lpAttribute* array. The *lpAttribute* parameter should hold the address of a word array than contains at least *nLength* elements. The *dwReadCoord* specifies the coordinate in the screen console buffer where *w.ReadConsoleOutputtAttribute* begins reading attributes (the H.O. word holds the y-coordinate, the L.O. word holds the x-coordinate). The *lpNumberOfAttrsRead* parameter holds the address of an integer where *w.ReadConsoleOutputAttribute* will store the number of attribute values it reads from the screen buffer. The value this function returns will be less than *nLength* if the function attempts to read data beyond the end of the screen buffer.

Unlike *w.ReadConsoleOutput*, *w.ReadConsoleOutputAttribute* does not read a rectangular block of data into a two-dimensional array. Instead, this function simply reads a linear block of attributes from the screen buffer starting at the specified coordinate. If this function reaches the end of the current line and *nLength* specifies additional attributes to read, this function returns the attributes beginning at the start of the next line. It stops, however, at the end of the screen buffer.

6.7.2.18: w.ReadConsoleOutputCharacter

```
static
    ReadConsoleOutputCharacter: procedure
    (
            hConsoleOutput:         dword;
        var lpCharacter:            char;
            nLength:                dword;
            dwReadCoord:            COORD;
        var lpNumberOfCharsRead:    dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__ReadConsoleOutputCharacterA@20" );

type
    COORD:
        record
            x: word;
            y: word;
        endrecord;
```

The *w.ReadConsoleOutputCharacter* function is like *w.ReadConsoleOutput* except it reads a linear sequence of characters from the screen buffer (like *w.ReadConsoleOutputAttribute* reads attributes) from the buffer rather than reading a rectangular block of characters. The *hConsoleOutput* parameter specifies the handle for the screen output buffer (i.e., the standard output handle). The *lpCharacter* parameter holds the address of the buffer to receive the characters; it should have room for at least *nLength* characters (the third parameter, that specifies the number of characters to read). The *dwReadCoord* parameter specifies the character cell coordinate in the screen buffer where this function begins reading characters. This function returns the number of characters actually read in the integer whose address you pass in the *lpNumberOfCharsRead* parameter; this number will match *nLength* unless you attempt to read beyond the end of the screen buffer.

6.7.2.19: w.ScrollConsoleScreenBuffer

```
static
```

```
    ScrollConsoleScreenBuffer: procedure
    (
            hConsoleOutput:         dword;
        var lpScrollRectangle:      SMALL_RECT;
        var lpClipRectangle:        SMALL_RECT;
            dwDestinationOrigin:    COORD;
        var lpFill:                 CHAR_INFO
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__ScrollConsoleScreenBufferA@20" );

type
    COORD:
        record
            x: word;
            y: word;
        endrecord;

    CHARTYPE:
        union
            UnicodeChar :word;  // Note: HLA's prototype only returns ASCII characters.
            AsciiChar   :byte;
        endunion;

    CHAR_INFO:
        record
            CharVal    :CHARTYPE;
            Attributes :word;
        endrecord;

    SMALL_RECT:
        record
            Left    :word;
            Top     :word;
            Right   :word;
            Bottom  :word;
         endrecord;
```

The *w.ScrollConsoleScreenBuffer* function scrolls a rectangular region of a screen buffer. This function achieves this by moving a source rectangular region of the screen buffer to some other point in the screen buffer. The *lpScrollRectangle* parameter specifies the source rectangle (via the coordinates of the upper left hand and lower right hand corners of the rectangle). The *dwDestinationOrigin* parameter specifies the upper left hand corner of the target location where this function will move the source rectangle. Any locations in the source rectangle that do not overlap with the source rectangle wind up getting filled with the character and attribute specified by the *lpFill* parameter. The *lpClipRectangle* parameter specifies a clipping rectangle . Actual scrolling only takes place within this clipping rectangle. For example, if you wanted to scroll the screen up one line, you would set the *lpScrollRectangle* parameter to encompass the whole screen (say, 80x25), set the destination origin to (0,-1), and then set the *lpClipRectangle* to the same values as the *lpScrollRectangle* value. You may also pass NULL as the value for *lpClipRectangle*, in which case *w.ScrollConsoleScreenBuffer* assumes that the clip region is equal to the source region (*lpScrollRectangle*).

6.7.2.20: w.SetConsoleActiveScreenBuffer

```
static
    SetConsoleActiveScreenBuffer: procedure
    (
        hConsoleOutput: dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__SetConsoleActiveScreenBuffer@4" );
```

This function displays the screen buffer whose handle you pass as the parameter in the console window. Although an application may have only one console window, it may have multiple screen buffers that it can display in that window. When the program first begins execution, the handle associated with the default console screen buffer is the standard output handle. You may create new screen buffers (and obtain their handles) via the *w.CreateConsoleScreenBuffer* call and then you can switch to these screen buffers using the *w.SetConsoleActiveScreenBuffer* API invocation.

6.7.2.21: w.SetConsoleCursorInfo

```
static
    SetConsoleCursorInfo: procedure
    (
            hConsoleOutput:         dword;
        var lpConsoleCursorInfo:    CONSOLE_CURSOR_INFO
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__SetConsoleCursorInfo@8" );

type
    CONSOLE_CURSOR_INFO:
        record
            dwSize    :dword;
            bVisible  :dword;
        endrecord;
```

The *w.SetConsoleCursorInfo* function lets you specify the size of the cursor in the console window and whether the cursor is visible or invisible. The *hConsoleOutput* parameter is the screen buffer handle of the screen buffer to which you wish the operation to apply. The *lpConsoleCursorInfo* parameter is a pointer to a *w.CONSOLE_CURSOR_INFO* data structure whose two fields (*dwSize* and *bVisible*) specify the size of the cursor and whether or not it is visible. The dwSize field must contain a value between 1 and 100 and specifies the percentage of a character cell that the cursor will fill. The cursor fills the cell from the bottom of the cell to the top, so a 1 would produce a cursor that is just an underline, 100 would produce a cursor that completely fills the character cell, and 50 would produce a cursor that fills the bottom half of the character cell. The *bVisible* field specifies whether the cursor will be visible in the console window. If this field contains true, then the cursor will be visible, if it contains false then Windows will not display the cursor.

6.7.2.22: w.SetConsoleCursorPosition

```
static
```

```
    SetConsoleCursorPosition: procedure
    (
        hConsoleOutput:     dword;
        dwCursorPosition:   COORD
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__SetConsoleCursorPosition@8" );

type
    COORD:
        record
            x: word;
            y: word;
        endrecord;
```

This function positions the cursor at the coordinate specified by the *dwCursorPosition* parameter. The *hConsoleOutput* parameter specifies the handle of the console screen buffer to which the cursor movement operation applies. If this function moves the cursor to a point in the screen buffer that is not displayed in the console window, then Windows will adjust the window to make the character cell specified by *dwCursorPosition* visible in the window.

6.7.2.23: w.SetConsoleScreenBufferSize

```
static
    SetConsoleScreenBufferSize: procedure
    (
        hConsoleOutput: dword;
        dwSize:         COORD
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__SetConsoleScreenBufferSize@8" );

type
    COORD:
        record
            x: word;
            y: word;
        endrecord;
```

The *w.SetConsoleScreenBufferSize* sets the size of the screen buffer. The *hConsoleOutput* parameter is the handle of the screen buffer whose size you want to change, the *dwSize* parameter specifies the new size of the buffer (the *dwSize.x* field specifies the new width, the *dwSize.y* field specifies the new height). This function fails if the screen buffer size (in either dimension) is smaller than the size of the console window or less than the minimum specified by the system (see the *w.GetSystemMetrics* call to find the minimally permissible size).

6.7.2.24: w.SetConsoleTextAttribute

```
static
    SetConsoleTextAttribute: procedure
    (
        hConsoleOutput: dword;
```

```
        wAttributes:    word
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__SetConsoleTextAttribute@8" );

const
    FOREGROUND_BLUE := $1;
    FOREGROUND_GREEN := $2;
    FOREGROUND_RED := $4;
    FOREGROUND_INTENSITY := $8;
    BACKGROUND_BLUE := $10;
    BACKGROUND_GREEN := $20;
    BACKGROUND_RED := $40;
    BACKGROUND_INTENSITY := $80;
```

The *w.SetConsoleTextAttribute* function lets you specify the default text attribute that Windows will use when writing text to the console window (e.g., via the HLA Standard Library stdout.put call). The hConsoleOutput parameter is the handle of the screen buffer whose default attribute you wish to set; the wAttributes parameter is the attribute value. This is a bitmap of color values created by combining the *w.FOREGROUND_xxx* and *w.BACKGROUND_xxx* values, e.g.,

w.SetConsoleTextAttribute(hndl, w.FOREGROUND_RED | w.BACKGROUND_BLUE);

6.7.2.25: w.SetConsoleTitle

```
static
    SetConsoleTitle: procedure
    (
        lpConsoleTitle: string
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__SetConsoleTitleA@4" );
```

The *w.SetConsoleTitle* function copies the string you pass as a parameter (*lpConsoleTitle*) to the title bar of the console window.

6.7.2.26: w.SetConsoleWindowInfo

```
static
    SetConsoleWindowInfo: procedure
    (
            hConsoleOutput:     dword;
            bAbsolute:          dword;
        var lpConsoleWindow:    SMALL_RECT
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__SetConsoleWindowInfo@12" );

type
    SMALL_RECT:
```

```
        record
            Left    :word;
            Top     :word;
            Right   :word;
            Bottom  :word;
         endrecord;
```

The *w.SetConsoleWindowInfo* function sets the size and position of the console s window. The *hConsoleOutput* parameter specifies the handle of a screen buffer (to which this operation applies). The *bAbsolute* parameter is a boolean value (true/1 or false/0) that specifies whether the coordinates you re supplying are relative to the current coordinates of the window (that is, relative to the current upper left hand corner of the window within the screen buffer) or absolute (specifies the coordinates of the upper left hand corner of the window within the screen buffer). If *bAbsolute* is true, then the coordinates are absolute and specify a character cell position within the screen buffer for the upper-left hand corner of the window. Thee lpConsoleWindow parameter is the address of a *w.SMALL_RECT* structure that specifies the new origin and size of the console display window. This function fails if the new window size would exceed the boundaries of the screen buffer.

You may use this function to scroll through vertically the screen buffer by adjusting the *Top* and *Bottom* fields of the *w.SMALL_RECT* record by the same amount. Similarly, you can use this function to scroll horizontally through the screen buffer by adjusting the *Left* and *Right* fields by the same amount.

6.7.2.27: w.SetStdHandle

```
static
    SetStdHandle: procedure
    (
        nStdHandle: dword;
        hHandle:    dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__SetStdHandle@8" );

const
    STD_INPUT_HANDLE := 10;
    STD_OUTPUT_HANDLE := 11;
    STD_ERROR_HANDLE := 12;
```

This function redirects the standard input, standard output, or standard error devices to a handle you specify. The first parameter is one of the constant values *w.STD_INPUT_HANDLE*, *w.STD_OUTPUT_HANDLE*, or *w.STD_ERROR_HANDLE* that specifies which of the standard handles you want to redirect. The *hHandle* parameter is a file handle that specifies where you want the output redirected. For example, you may redirect the standard output to a specific console screen buffer by specifying that screen output buffer s handle.

6.7.2.28: w.WriteConsole

```
static
    WriteConsole: procedure
    (
            hConsoleOutput:         dword;
        var lpBuffer:               var;
            nNumberOfCharsToWrite:  dword;
```

```
        var lpNumberOfCharsWritten: dword;
        var lpReserved:             var
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__WriteConsoleA@20" );
```

The *w.WriteConsole* function writes a string to the console screen buffer specified by the *hConsoleOutput* handle. The *lpBuffer* parameter specifies the address of the character data to write to the console buffer. The *nNumberOfCharsToWrite* parameter specifies the character count. This function returns the actual number of characters written in the integer variable whose address you pass in the *lpNumberOfCharsWritten* parameter. The *lpReserved* parameter is reserved and you must pass NULL for its value.

Note that Windows actually provides two versions of this function - one for ASCII characters and one for Unicode characters. The HLA prototype specifies the ASCII version. If you need to call the Unicode version, you will need to create your own prototype for that function. Because the HLA version calls the ASCII version, this function is virtually identical to a w.WriteFile call (e.g., stdout.put).

6.7.2.29: w.WriteConsoleInput

```
static
    WriteConsoleInput: procedure
    (
            hConsoleInput:              dword;
        var lpBuffer:                   INPUT_RECORD;
            nLength:                    dword;
        var lpNumberOfEventsWritten:    dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__WriteConsoleInputA@16" );
```

The *w.WriteConsoleInput* function writes one or more *w.INPUT_RECORD* structures to the console s input buffer. This function places any input records written at the end of the buffer, behind any pending input events. You may use this function to simulate keyboard or mouse input.

The *hConsoleInput* parameter specifies the handle of the console input buffer. You could, for example, supply the handle of the standard input device here. The lpBuffer parameter is a pointer to an array of one or more *w.INPUT_RECORD* objects (please see the discussion of input records in the section on the *w.ReadConsoleInput* function). The *nLength* parameter is the number of input records in the array that *lpBuffer* points at that you want this function to process. This function stores the number of input records processed in the integer variable whose address you pass in the *lpNumberOfEventsWritten* record.

6.7.2.30: w.WriteConsoleOutput

```
static
    WriteConsoleOutput: procedure
    (
            hConsoleOutput: dword;
        var lpBuffer:       CHAR_INFO;
            dwBufferSize:   COORD;
            dwBufferCoord:  COORD;
```

```
        VAR lpWriteRegion:  SMALL_RECT
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__WriteConsoleOutputA@20" );

type
    COORD:
        record
            x: word;
            y: word;
        endrecord;

    SMALL_RECT:
        record
            Left    :word;
            Top     :word;
            Right   :word;
            Bottom  :word;
         endrecord;

    CHARTYPE:
        union
            UnicodeChar :word;  // Note: HLA's prototype only returns ASCII characters.
            AsciiChar   :byte;
        endunion;

    CHAR_INFO:
        record
            CharVal    :CHARTYPE;
            Attributes :word;
        endrecord;
```

The *w.WriteConsoleOutput* function writes a rectangular block of character and attribute data to a screen buffer. The *hConsoleOutput* parameter is the handle of the screen buffer where this function is to write the data. The *lpBuffer* parameter is the address of an array of character/attribute values that this function is to write to the screen. Note that each character/attribute element in *lpBuffer* consumes 32-bits (16 bits for the attribute and 16 bits for the character). The *dwBufferSize* parameter specifies the of the character/attribute data at which *lpBuffer* points; the *dwBufferSize.x* field specifies the number of columns, the *dwBufferSize.y* field specifies the number of rows of character/attribute data in the buffer area. Note that this parameter specifies the size of the source buffer, not the size of the data to write; the actual data written may be a subset of this two-dimensional array. The *dwBufferCoord* parameter specifies the coordinate of the upper-left hand corner of the rectangular area to write within the source buffer. The *lpWriteRegion* parameter is the address of a *w.SMALL_RECT* data structure; on input, these record specifies the region of the screen buffer where Windows is to write the data from the source buffer. When Windows returns, it writes the size of the actual rectangle written into this record structure (the actual rectangle may be smaller if the source rectangle s size violates the boundaries of the screen buffer).

6.7.2.31: w.WriteConsoleAttribute

```
static
    WriteConsoleOutputAttribute: procedure
    (
```

```
            hConsoleOutput:         dword;
        var lpAttribute:            word;
            nLength:                dword;
            dwWriteCoord:           COORD;
        var lpNumberOfAttrsWritten: dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__WriteConsoleOutputAttribute@20" );

type
    COORD:
        record
            x: word;
            y: word;
        endrecord;
```

The *w.WriteConsoleAttribute* function writes a linear sequence of attribute values to the console without affecting the character data at the cell positions where it writes those attributes. The *hConsoleOutput* parameter specifies the handle of a console screen buffer. The *lpAttribute* parameter is the address of an array of one or more 16-bit attribute values. The *nLength* parameter specifies the number of attribute values to write to the screen. The *dwWriteCoord* parameter specifies the (*x,y*) coordinate of the screen buffer where this function is to begin writing the attribute values (the *dwWriteCoord.x* field appears in the L.O. word and the *dwWriteCoord.y* field appears in the H.O. word of this double word parameter value). The *lpNumberOfAttrsWritten* parameter is the address of an integer variable that will receive the number of attributes actually written to the screen buffer. This number will be less than *nLength* if the function attempts to write data beyond the end of the screen buffer.

6.7.2.32: w.WriteConsoleOutputCharacter

```
static
    WriteConsoleOutputCharacter: procedure
    (
            hConsoleOutput:         dword;
            lpCharacter:            string;
            nLength:                dword;
            dwWriteCoord:           COORD;
        var lpNumberOfCharsWritten: dword
    );
        @stdcall;
        @returns( "eax" );
        @external( "__imp__WriteConsoleOutputCharacterA@20" );

type
    COORD:
        record
            x: word;
            y: word;
        endrecord;
```

The *w.WriteConsoleOutputCharacter* function writes a linear sequence of attribute values to the console without affecting the character data at the cell positions where it writes those attributes. The *hConsoleOutput* parameter specifies the handle of a console screen buffer. The *lpCharacter* parameter is the address of an array of one or more character values. The *nLength* parameter specifies the number of character values to write to the

screen. The *dwWriteCoord* parameter specifies the (*x,y*) coordinate of the screen buffer where this function is to begin writing the character values (the *dwWriteCoord.x* field appears in the L.O. word and the *dwWriteCoord.y* field appears in the H.O. word of this double word parameter value). The *lpNumberOfCharsWritten* parameter is the address of an integer variable that will receive the number of attributes actually written to the screen buffer. This number will be less than *nLength* if the function attempts to write data beyond the end of the screen buffer.

Note that Windows actually provides two versions of this function, one for Unicode and one for ASCII character data. The HLA Standard Library function prototypes the ASCII version. If you need the Unicode version, you can easily prototype that function yourself.

6.7.2.33: Plus More!

There are many, many, additional console related functions beyond the more common ones presented here. If you re interested in high-performance console application programming under Windows, you ll want to read the API documentation appearing on the accompanying CD-ROM or read up on console I/O on Microsoft s MSDN system.

6.7.3: HLA Standard Library Console Support

The HLA Standard Library provides a small console library module that simulates an ANSI terminal. Although this set of library functions provides but a small set of available Win32 console features, it is slightly easier to use for the more common operations (like cursor positioning, clearing portions of the screen, and things like that). Another advantage to the HLA Standard Library Console module is that it is portable - console applications that use this console module will compile and run, unchanged, under Linux. However, because this is a book on Windows programming, we won t bother exploring this portability issue any farther.

There is also a set of obsolete console library functions available with in the HLA Examples code (available on the accompanying CD-ROM or via download at http://webster.cs.ucr.edu). This set of library routines provided a much stronger set of console capabilities than the current HLA Standard Library (at lot of functionality was removed in order to achieve portability to Linux). These older functions are mostly wrappers around Win32 console API calls that make them more convenient to use. They re also great examples of calls to the Win32 API functions. The major drawback to this source code is that it is quite old - hailing from the days when HLA was missing several important features that make interfacing with Windows a lot easier; so the source code makes the calls to the Win32 API functions in some peculiar ways.

6.8: This Isn’t the Final Word!

There is quite a bit more to text processing under Windows than a even a long chapter like this one can present. However, most people who are reading this text probably want to learn how to write GUI applications involving graphics, so it s time to set the discussion of text down for a while and start looking at how to do graphics in a Windows environment.

Chapter 7: Graphics

7.1:

Chapter 8: Event-Driven Input/Output

8.1:

Chapter 9: Debugging Windows Applications

9.1:

```
hla -aZf -aZi -lDEBUG -lDEBUGTYPE:COFF  t
```

Chapter 10: Controls, Dialogs, Menus, and Windows

10.1:

Chapter 11: The Resource Compiler

11.1:

Chapter 12: Icons, Cursors, Bitmaps, and Strings

12.1:

Chapter 13: File I/O and Other Traditional Kernel Services

13.1:

Index: Windows Programming in Assembly Language