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.