Managing Large Programs Chapter Nine

9.1 Chapter Overview

When writing larger HLA programs you do not typically write the whole program as a single source file. This chapter discusses how to break up a large project into smaller pieces and assemble the pieces separately. This radically reduces development time on large projects.

9.2 Managing Large Programs

Most assembly language programs are not totally stand alone programs. In general, you will call various standard library or other routines that are not defined in your main program. For example, you've probably noticed by now that the 80x86 doesn't provide any machine instructions like "read", "write", or "printf" for doing I/O operations. Of course, you can write your own procedures to accomplish this. Unfortunately, writing such routines is a complex task, and beginning assembly language programmers are not ready for such tasks. That's where the HLA Standard Library comes in. This is a package of procedures you can call to perform simple I/O operations like *stdout.put*.

The HLA Standard Library contains tens of thousands of lines of source code. Imagine how difficult programming would be if you had to merge these thousands of lines of code into your simple programsl imagine how slow compiling your programs would be if you had to compile those tens of thousands of lines with each program you write. Fortunately, you don't have to.

For small programs, working with a single source file is fine. For large programs this gets very cumbersome (consider the example above of having to include the entire HLA Standard Library into each of your programs). Furthermore, once you've debugged and tested a large section of your code, continuing to assemble that same code when you make a small change to some other part of your program is a waste of time. The HLA Standard Library, for example, takes several minutes to assemble, even on a fast machine. Imagine having to wait five or ten minutes on a fast Pentium machine to assemble a program to which you've made a one line change!

As with high level languages, the solution is *separate compilation* . First, you break up your large source files into manageable chunks. Then you compile the separate files into object code modules. Finally, you link the object modules together to form a complete program. If you need to make a small change to one of the modules, you only need to reassemble that one module, you do not need to reassemble the entire program.

The HLA Standard Library works in precisely this way. The Standard Library is already compiled and ready to use. You simply call routines in the Standard Library and link your code with the Standard Library using a *linker* program. This saves a tremendous amount of time when developing a program that uses the Standard Library code. Of course, you can easily create your own object modules and link them together with your code. You could even add new routines to the Standard Library so they will be available for use in future programs you write.

"Programming in the large" is a term software engineers have coined to describe the processes, methodologies, and tools for handling the development of large software projects. While everyone has their own idea of what "large" is, separate compilation, and some conventions for using separate compilation, are among the more popular techniques that support "programming in the large." The following sections describe the tools HLA provides for separate compilation and how to effectively employ these tools in your programs.

9.3 The #INCLUDE Directive

The #INCLUDE directive, when encountered in a source file, switches program input from the current file to the file specified in the parameter list of the include directive. This allows you to construct text files containing common constants, types, source code, and other HLA items, and include such a file into the assembly of several separate programs. The syntax for the include directive is

```
#include( "filename" )
```

Filename must be a valid filename. HLA merges the specified file into the compilation at the point of the #INCLUDE directive. Note that you can nest #INCLUDE statements inside files you include. That is, a file being included into another file during assembly may itself include a third file. In fact, the "stdlib.hhf" header file you see in most example programs contains the following[1]:

```
#include( "hla.hhf" )
#include( "x86.hhf" )
#include( "misctypes.hhf" )
#include( "hll.hhf" )

#include( "excepts.hhf" )
#include( "memory.hhf" )

#include( "args.hhf" )
#include( "conv.hhf" )
#include( "strings.hhf" )
#include( "cset.hhf" )
#include( "patterns.hhf" )
#include( "tables.hhf" )
#include( "arrays.hhf" )
#include( "chars.hhf" )

#include( "math.hhf" )
#include( "rand.hhf" )

#include( "stdio.hhf" )
#include( "stdin.hhf" )
#include( "stdout.hhf" )
```

Program 9.1 The stdlib.hhf Header File, as of 01/01/2000

By including "stdlib.hhf" in your source code, you automatically include all the HLA library modules. It's often more efficient (in terms of compile time and size of code generated) to provide only those #INCLUDE statements for the modules you actually need in your program. However, including "stdlib.hhf" is extremely convenient and takes up less space in this text, which is why most programs appearing in this text use "stdlib.hhf".

Note that the #INCLUDE directive does not need to end with a semicolon. If you put a semicolon after the #INCLUDE, that semicolon becomes part of the source file and is the first character following the included file during compilation. HLA generally allows spare semicolons in various parts of the program, so you will often see a #INCLUDE statement ending with a semicolon that produces no harm. In general,

1. Note that this file changes over time as new library modules appear in the HLA Standard Library, so this file is probably not up to date. Furthermore, there are some minor differences between the Linux and Windows version of this file. The OS-specific entries do not appear in this example.

though, you should not get in the habit of putting semicolons after #INCLUDE statements because there is the slight possibility this could create a syntax error in certain circumstances.

Using the #include directive by itself does not provide separate compilation. You *could* use the include directive to break up a large source file into separate modules and join these modules together when you compile your file. The following example would include the PRINTF.HLA and PUTC.HLA files during the compilation of your program:

```
#include( "printf.hla" )
#include( "putc.hla" )
```

Now your program *will* benefit from the modularity gained by this approach. Alas, you will not save any development time. The #INCLUDE directive inserts the source file at the point of the #INCLUDE during compilation, exactly as though you had typed that code in yourself. HLA still has to compile the code and that takes time. Were you to include all the files for the Standard Library routines in this manner, your compilations would take *forever*.

In general, you should *not* use the include directive to include source code as shown above[2]. Instead, you should use the #INCLUDE directive to insert a common set of constants, types, external procedure declarations, and other such items into a program. Typically an assembly language include file does *not* contain any machine code (outside of a macro, see the chapter on Macros and the Compile-Time Language for details). The purpose of using #INCLUDE files in this manner will become clearer after you see how the external declarations work.

9.4 Ignoring Duplicate Include Operations

As you begin to develop sophisticated modules and libraries, you eventually discover a big problem: some header files will need to include other header files (e.g., the stdlib.hhf header file includes all the other Standard Library Header files). Well, this isn't actually a big problem, but a problem will occur when one header file includes another, and that second header file includes another, and that third header file includes another, and ..., and that last header file includes the first header file. Now *this* is a big problem.

There are two problems with a header file indirectly including itself. First, this creates an infinite loop in the compiler. The compiler will happily go on about its business including all these files over and over again until it runs out of memory or some other error occurs. Clearly this is not a good thing. The second problem that occurs (usually before the problem above) is that the second time HLA includes a header file, it starts complaining bitterly about duplicate symbol definitions. After all, the first time it reads the header file it processes all the declarations in that file, the second time around it views all those symbols as duplicate symbols.

HLA provides a special include directive that eliminates this problem: #INCLUDEONCE. You use this directive exactly like you use the #include directive, e.g.,

```
#includeonce( "myHeaderFile.hhf" )
```

If myHeaderFile.hhf directly or indirectly includes itself (with a #INCLUDEONCE directive), then HLA will ignore the new request to include the file. Note, however, that if you use the #INCLUDE directive, rather than #INCLUDEONCE, HLA will include the file a second name. This was done in case you really do need to include a header file twice, for some reason (though it is hard to imagine needing to do this).

The bottom line is this: you should always use the #INCLUDEONCE directive to include header files you've created. In fact, you should get in the habit of always using #INCLUDEONCE, even for header files created by others (the HLA Standard Library already has provisions to prevent recursive includes, so you don't have to worry about using #INCLUDEONCE with the Standard Library header files).

There is another technique you can use to prevent recursive includes – using conditional compilation. For details on this technique, see the chapter on the HLA Compile-Time Language in a later volume.

2. There is nothing wrong with this, other than the fact that it does not take advantage of separate compilation.

9.5 UNITs and the EXTERNAL Directive

Technically, the #INCLUDE directive provides you with all the facilities you need to create modular programs. You can create several modules, each containing some specific routine, and include those modules, as necessary, in your assembly language programs using #INCLUDE. However, HLA provides a better way: external and public symbols.

One major problem with the include mechanism is that once you've debugged a routine, including it into a compilation still wastes a lot of time since HLA must recompile bug-free code every time you assemble the main program. A much better solution would be to preassemble the debugged modules and link the object code modules together rather than reassembling the entire program every time you change a single module. This is what the EXTERNAL directive allows you to do.

To use the external facilities, you must create at least two source files. One file contains a set of variables and procedures used by the second. The second file uses those variables and procedures without knowing how they're implemented. The only problem is that if you create two separate HLA programs, the linker will get confused when you try to combine them. This is because both HLA programs have their own main program. Which main program does the OS run when it loads the program into memory? To resolve this problem, HLA uses a different type of compilation module, the UNIT, to compile programs without a main program. The syntax for an HLA UNIT is actually simpler than that for an HLA program, it takes the following form:

```
unit unitname;

    << declarations >>

end unitname;
```

With one exception (the VAR section), anything that can go in the declaration section of an HLA program can go into the declaration section of an HLA unit. Notice that a unit does not have a BEGIN clause and there are no program statements in the unit[3]; a unit only contains declarations.

In addition to the fact that a unit does not contain any executable statements, there is one other difference between units and programs. Units cannot have a VAR section. This is because the VAR section declares variables that are local to the main program's source code. Since there is no source code associated with a unit, VAR sections are illegal[4]

To demonstrate, consider the following two modules:

```
unit Number1;

static
    Var1:   uns32;
    Var2:   uns32;

    procedure Add1and2;
    begin Add1and2;

        push( eax );
        mov( Var2, eax );
        add( eax, Var1 );

    end Add1and2;
```

3. Of course, units may contain procedures and those procedures may have statements, but the unit itself does not have any executable instructions associated with it.

4. Of course, procedures in the unit may have their own VAR sections, but the procedure's declaration section is separate from the unit's declaration section.


```
end Number1;
```

Program 9.2 Example of a Simple HLA Unit

```
program main;
#include( "stdlib.hhf" );

begin main;

    mov( 2, Var2 );
    mov( 3, Var1 );
    Add1and2();
    stdout.put( "Var1=", Var1, nl );

end main;
```

Program 9.3 Main Program that References External Objects

The main program references Var1, Var2, and Add1and2, yet these symbols are external to this program (they appear in unit *Number1*). If you attempt to compile the main program as it stands, HLA will complain that these three symbols are undefined.

Therefore, you must declare them external with the EXTERNAL option. An external procedure declaration looks just like a forward declaration except you use the reserved word EXTERNAL rather than FORWARD. To declare external static variables, simply follow those variables' declarations with the reserved word EXTERNAL. The following is a modification to the previous main program that includes the external declarations:

```
program main;
#include( "stdlib.hhf" );

    procedure Add1and2; external;

static
    Var1: uns32; external;
    Var2: uns32; external;

begin main;

    mov( 2, Var2 );
    mov( 3, Var1 );
    Add1and2();
    stdout.put( "Var1=", Var1, nl );

end main;
```


Program 9.4 Modified Main Program with EXTERNAL Declarations

If you attempt to compile this second version of *main*, using the typical HLA compilation command "HLA main2.hla" you will be somewhat disappointed. This program will actually compile without error. However, when HLA attempts to link this code it will report that the symbols *Var1*, *Var2*, and *Add1and2* are undefined. This happens because you haven't compiled and linked in the associated unit with this main program. Before you try that, and discover that it still doesn't work, you should know that all symbols in a unit, by default, are *private* to that unit. This means that those symbols are inaccessible in code outside that unit unless you explicitly declare those symbols as *public* symbols. To declare symbols as public, you simply put external declarations for those symbols in the unit before the actual symbol declarations. If an external declaration appears in the same source file as the actual declaration of a symbol, HLA assumes that the name is needed externally and makes that symbol a public (rather than private) symbol. The following is a correction to the *Number1* unit that properly declares the external objects:

```
unit Number1;

static
    Var1:   uns32; external;
    Var2:   uns32; external;

    procedure Add1and2; external;

static
    Var1:   uns32;
    Var2:   uns32;


    procedure Add1and2;
    begin Add1and2;

        push( eax );
        mov( Var2, eax );
        add( eax, Var1 );

    end Add1and2;

end Number1;
```

Program 9.5 Correct Number1 Unit with External Declarations

It may seem redundant declaring these symbols twice as occurs in Program 9.5, but you'll soon seen that you don't normally write the code this way.

If you attempt to compile the *main* program or the *Number1* unit using the typical HLA statement, i.e.,

```
HLA main2.hla
HLA unit2.hla
```

You'll quickly discover that the linker still returns errors. It returns an error on the compilation of main2.hla because you still haven't told HLA to link in the object code associated with unit2.hla. Likewise, the linker complains if you attempt to compile unit2.hla by itself because it can't find a main program. The simple solution is to compile both of these modules together with the following single command:

```
HLA main2.hla unit2.hla
```


This command will properly compile both modules and link together their object code.

Unfortunately, the command above defeats one of the major benefits of separate compilation. When you issue this command it will compile both main2 and unit2 prior to linking them together. Remember, a major reason for separate compilation is to reduce compilation time on large projects. While the above command is convenient, it doesn't achieve this goal.

To separately compile the two modules you must run HLA separately on them. Of course, we saw earlier that attempting to compile these modules separately produced linker errors. To get around this problem, you need to compile the modules without linking them. The "-c" (compile-only) HLA command line option achieves this. To compile the two source files without running the linker, you would use the following commands:

```
HLA -c main2.hla
HLA -c unit2.hla
```

This produces two object code files, main2.obj and unit2.obj, that you can link together to produce a single executable. You could run the linker program directly, but an easier way is to use the HLA compiler to link the object modules together for you:

```
HLA main2.obj unit2.obj
```

Under Windows, this command produces an executable file named *main2.exe*[5]; under Linux, this command produces a file named *main2*. You could also type the following command to compile the main program and link it with a previously compiled unit2 object module:

```
HLA main2.hla unit2.obj
```

In general, HLA looks at the suffixes of the filenames following the HLA commands. If the filename doesn't have a suffix, HLA assumes it to be ".HLA". If the filename has a suffix, then HLA will do the following with the file:

- If the suffix is ".HLA", HLA will compile the file with the HLA compiler.
- If the suffix is ".ASM", HLA will assemble the file with MASM.
- If the suffix is ".OBJ" or ".LIB"(Windows), or ".o" or ".a" (Linux), then HLA will link that module with the rest of the compilation.

9.5.1 Behavior of the EXTERNAL Directive

Whenever you declare a symbol EXTERNAL using the external directive, keep in mind several limitations of EXTERNAL objects:

- Only one EXTERNAL declaration of an object may appear in a given source file. That is, you cannot define the same symbol twice as an EXTERNAL object.
- Only PROCEDURE, STATIC, READONLY, and STORAGE variable objects can be external. VAR and parameter objects cannot be external.
- External objects must be at the global declaration level. You cannot declarare EXTERNAL objects within a procedure or other nested structure.
- EXTERNAL objects publish their name globally. Therefore, you must carefully choose the names of your EXTERNAL objects so they do not conflict with other symbols.

This last point is especially important to keep in mind. As this text is being written, the HLA compiler translates your HLA source code into assembly code. HLA assembles the output by using MASM (the Microsoft Macro Assembler), Gas (Gnu's as), or some other assembler. Finally, HLA links your modules using a linker. At each step in this process, your choice of external names could create problems for you.

5. If you want to explicitly specify the name of the output file, HLA provides a command-line option to achieve this. You can get a menu of all legal command line options by entering the command "HLA -?".

Consider the following HLA external/public declaration:

```
static
       extObj:       uns32; external;
       extObj:       uns32;
       localObject:  uns32;
```

When you compile a program containing these declarations, HLA automatically generates a "munged" name for the *localObject* variable that probably isn't ever going to have any conflicts with system-global external symbols[6]. Whenever you declare an external symbol, however, HLA uses the object's name as the default external name. This can create some problems if you inadvertently use some global name as your variable name. Worse still, the assembler will not be able to properly process HLA's output if you happen to choose an identifier that is legal in HLA but is one of the assembler's reserved word. For example, if you attempt to compile the following code fragment as part of an HLA program (producing MASM output), it will compile properly but MASM will not be able to assemble the code:

```
static
    c: char; external;
    c: char;
```

The reason MASM will have trouble with this is because HLA will write the identifier "c" to the assembly language output file and it turns out that "c" is a MASM reserved word (MASM uses it to denote C-language linkage).

To get around the problem of conflicting external names, HLA supports an additional syntax for the EXTERNAL option that lets you explicitly specify the external name. The following example demonstrates this extended syntax:

```
static
    c: char; external( "var_c" );
    c: char;
```

If you follow the EXTERNAL keyword with a string constant enclosed by parentheses, HLA will continue to use the declared name (*c* in this example) as the identifier within your HLA source code. Externally (i.e., in the assembly code) HLA will substitute the name *var_c* whenever you reference *c*. This features helps you avoid problems with the misuse of assembler reserved words, or other global symbols, in your HLA programs.

You should also note that this feature of the EXTERNAL option lets you create *aliases*. For example, you may want to refer to an object by the name *StudentCount* in one module while refer to the object as *PersonCount* in another module (you might do this because you have a general library module that deals with counting people and you want to use the object in a program that deals only with students). Using a declaration like the following lets you do this:

```
static
    StudentCount: uns32; external( "PersonCount" );
```

Of course, you've already seen some of the problems you might encounter when you start creating aliases. So you should use this capability sparingly in your programs. Perhaps a more reasonable use of this feature is to simplify certain OS APIs. For example, Win32 uses some really long names for certain procedure calls. You can use the EXTERNAL directive to provide a more meaningful name than the standard one supplied by the operating system.

9.5.2 Header Files in HLA

HLA's technique of using the same EXTERNAL declaration to define public as well as external symbols may seem somewhat counter-intuitive. Why not use a PUBLIC reserved word for public symbols and

6. Typically, HLA creates a name like *?001A_localObject* out of *localObject*. This is a legal MASM identifier but it is not likely it will conflict with any other global symbols when HLA compiles the program with MASM.

the EXTERNAL keyword for external definitions? Well, as counter-intuitive as HLA's external declarations may seem, they are founded on decades of solid experience with the C/C++ programming language that uses a similar approach to public and external symbols[7]. Combined with a *header file*, HLA's external declarations make large program maintenance a breeze.

An important benefit of the EXTERNAL directive (versus separate PUBLIC and EXTERNAL directives) is that it lets you minimize duplication of effort in your source files. Suppose, for example, you want to create a module with a bunch of support routines and variables for use in several different programs (e.g., the HLA Standard Library). In addition to sharing some routines and some variables, suppose you want to share constants, types, and other items as well.

The #INCLUDE file mechanism provides a perfect way to handle this. You simply create a #INCLUDE file containing the constants, macros, and external definitions and include this file in the module that implements your routines and in the modules that use those routines (see Figure 9.1).

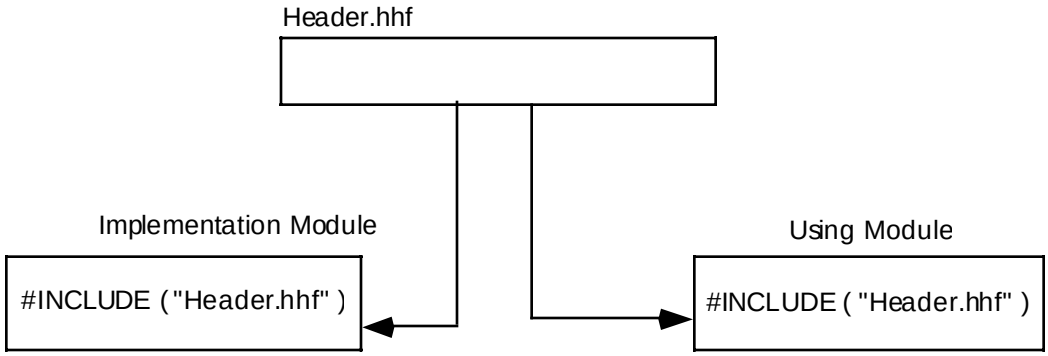

Figure 9.1 Using Header Files in HLA Programs

A typical header file contains only CONST, VAL, TYPE, STATIC, READONLY, STORAGE, and procedure prototypes (plus a few others we haven't look at yet, like macros). Objects in the STATIC, READONLY, and STORAGE sections, as well as all procedure declarations, are always EXTERNAL objects. In particular, you generally should not put any VAR objects in a header file, nor should you put any non-external variables or procedure bodies in a header file. If you do, HLA will make duplicate copies of these objects in the different source files that include the header file. Not only will this make your programs larger, but it will cause them to fail under certain circumstances. For example, you generally put a variable in a header file so you can share the value of that variable amongst several different modules. However, if you fail to declare that symbol as external in the header file and just put a standard variable declaration there, each module that includes the source file will get its own separate variable - the modules will not share a common variable.

If you create a standard header file, containing CONST, VAL, and TYPE declarations, and external objects, you should always be sure to include that file in the declaration section of all modules that need the definitions in the header file. Generally, HLA programs include all their header files in the first few statements after the PROGRAM or UNIT header.

This text adopts the HLA Standard Library convention of using an ".hhf" suffix for HLA header files ("HHF" stands for HLA Header File).

7. Actually, C/C++ is a little different. All global symbols in a module are assumed to be public unless explicitly declared private. HLA's approach (forcing the declaration of public items via EXTERNAL) is a little safer.

9.6 Make Files

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, there is a tool that can help you manage large projects: *make*[8]. The make program, with a little help from you, 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[9].

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. This example is Windows-specific, here's the same makefile statement in a Linux-friendly form:

Example :

```
pgm: pgma.o pgmb.o              --Linux/make example
```

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

8. Under Windows, Microsoft calls this program *nmake*. This text will use the more generic name "make" when refering to this program. If you are using Microsoft tools under Windows, just substitute "nmake" for "make" throughout this chapter.

9. 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.

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 (the Windows example) would probably look something like the following:

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

(The "-opgm.exe" option tells HLA to name the executable file "pgm.exe.") Here's the same example for Linux users:

```
pgm: pgma.o pgmb.o
    hla -opgm pgma.obj pgmb.obj
```

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 or pgm.exe) depends upon the object files (pgma.obj or pgma.o and pgmb.obj or pgmb.o). 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 (Windows) make file:

```
pgm.exe: pgma.obj pgmb.obj
    hla -opgm.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).

1. 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.
2. 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.
3. 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.
4. 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 (Linux example):

```
pgm: pgma.o pgmb.o
    hla -opgm pgma.o pgmb.o

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

pgmb.o: 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.o and pgmb.o 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 nor the Standard Library ".lib" (Windows) or ".a" (Linux) files in the dependency list. True, your resulting exectuable file does depend on this code, but the Standard Library 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[10]. 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.

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 the appropriate documentation. Note that several versions of MAKE exist. Microsoft produces nmake.exe, Borland has their own MAKE.EXE program and various versions of MAKE have been ported to Windows from UNIX systems (e.g., GMAKE). Linux users will typically employ the GNU make program. While these various make programs are not equivalent, they all do a pretty good job of handling the simple make syntax that this chapter describes.

9.7 Code Reuse

One of the principle goals of Software Engineering is to reduce program development time. Although the techniques we've studied in this chapter will certainly reduce development effort, there are bigger prizes to be had here. Consider for a moment a simple program that reads an integer from the user and then displays the value of that integer on the standard output device. You can easily write a trivial version of this program with about eight lines of HLA code. That's not too difficult. However, suppose you did not have the HLA Standard Library at your disposal. Now, instead of an eight line program, you'd be faced with writing a program that hundreds if not thousands of lines long. Obviously, this program will take a lot longer to write than the original eight-line version. The difference between these two applications is the fact that in the first version of this program you got to reuse some code that was already written; in the second version of the program you had to write everything from scratch. This concept of code reuse is very important when

10. 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.

writing large programs – you can get large programs working much more quickly if you reuse code from previous projects.

The idea behind code reuse is that many code sequences you write will be usable in future programs. As time passes and you write more code, progress on your projects will be faster since you can reuse code you've written (or others have written) on previous projects. The HLA Standard Library functions are the classic example, somebody had to write those functions so you could use them. And use them you do. As of this writing, the Standard Library represented about 50,000 lines of HLA source code. Imagine having to write a fair portion of that everytime you wanted to write an HLA program!

Although the HLA Standard Library contains lots of very useful routines and functions, this code base cannot possible predict the type of code you will need in every future project. The HLA Standard Library provides some of the more common routines you'll need when writing programs, but you're certainly going to have need for routines that the HLA Standard Library cannot satisfy. Unless you can find a source for the code you need from some third party, you're probably going to have to write the new routines yourself.

The trick when writing a program is to try and figure out which routines are general purpose and could be used in future programs; once you make this determination, you should write such routines separately from the rest of your application (i.e., put them in a separate source file for compilation). By keeping them separate, you can use them in future projects. If "try and figure out which routines are general purpose..." sounds a bit difficult, well, you're right it is. Even after 30 years of Software Engineering research, no one has really figured out how to effectively reuse code. There are some obvious routines we can reuse (that's why there are "standard libraries") but it is quite difficult for the practicing engineer to successfully predict which routines s/he will need in the future and write these as separate modules.

Attempting to teach you how to decide which routines are worthy of saving for future programs and which are specific to your current application is well beyond the scope of this text. There are several Software Engineering texts out there that try to explain how to do this, but keep in mind that even after the publication of these texts, practicing engineers still have problems picking the right routines to save. Hopefully, as you gain experience, you will begin to recognize those routines that are worth keeping for future programs and those that aren't worth bothering with. This text will take the easy way out and assume that you know which routines you want to keep and which you don't.

9.8 Creating and Managing Libraries

Imagine that you've created a few hundred routines over the past couple of years and you would like to have the object code ready to link with any new projects you begin. You could move all this code into a single source file, stick in a bunch of EXTERNAL declarations, and then link the resulting object file with any new programs you write that can use the routines in your "library". Unfortunately, there are a couple of problems with this approach. Let's take a look at some of these problems.

Problem number one is that your library will grow to a fairly good size with time; if you put the source code to every routine in a single source file, small additions or changes to the file will require a complete recompilation of the whole library. That's clearly not what we want to do, based on what you've learned from this chapter.

Another problem with this "solution" is that whenever you link this object file to your new applications, you link in the entire library, not just the routines you want to use. This makes your applications unnecessarily large, especially if your library has grown. Were you to link your simple projects with the entire HLA Standard library, for example, the result would be positively huge.

A solution to both of the above problems is to compile each routine in a separate file and produce a unique object file for it. Unfortunately, with hundreds of routines you're going to wind up with hundreds of object files; any time you want to call a dozen or so library routines, you'd have to link your main application with a dozen or so object modules from your library. Clearly, this isn't acceptable either.

You may have noticed by now that when you link your applications with the HLA Standard Library, you only link with a single file: *hlalib.lib (Windows)* or *hlalib.a (Linux)*. .LIB (library) and ".a" (archive) files are a collection of object files. When the linker processes a library file, it pulls out only the object files it

needs, it does not link the entire file with your application. Hence you get to work with a single file and your applications don't grow unnecessarily large.

Linux provids the "ar" (archiver) program to manage library files. To use this program to combine several object files into a single ".a" file, you'd use a command line like the following:

```
ar -q library.a list_of_.o_files
```

For more information on this command, check out the man page on the "ar" program ("man ar").

9.9 Name Space Pollution

One problem with creating libraries with lots of different modules is name space pollution. A typical library module will have a #INCLUDE file associated with it that provides external definitions for all the routines, constants, variables, and other symbols provided in the library. Whenever you want to use some routines or other objects from the library, you would typically #INCLUDE the library's header file in your project. As your libraries get larger and you add more declarations in the header file, it becomes more and more likely that the names you've chosen for your library's identifiers will conflict with names you want to use in your current project. This conflict is what is meant by name space pollution: library header files pollute the name space with many names you typically don't need in order to gain easy access to the few routines in the library you actually use. Most of the time those names don't harm anything – unless you want to use those names yourself in your program.

HLA requires that you declare all external symbols at the global (PROGRAM/UNIT) level. You cannot, therefore, include a header file with external declarations within a procedure[11]. As such, there will be no naming conflicts between external library symbols and symbols you declare locally within a procedure; the conflicts will only occur between the external symbols and your global symbols. While this is a good argument for avoiding global symbols as much as possible in your program, the fact remains that most symbols in an assembly language program will have global scope. So another solution is necessary.

HLA's solution, which it certainly uses in the Standard Library, is to put most of the library names in a NAMESPACE declaration section. A NAMESPACE declaration encapsulates all declarations and exposes only a single name (the NAMESPACE identifier) at the global level. You access the names within the NAMESPACE by using the familiar dot-notation (see "Namespaces" on page 496). This reduces the effect of namespace pollution from many dozens or hundreds of names down to a single name.

Of course, one disadvantage of using a NAMESPACE declaration is that you have to type a longer name in order to reference a particular identifier in that name space (i.e., you have to type the NAMESPACE identifier, a period, and then the specific identifier you wish to use). For a few identifiers you use frequently, you might elect to leave those identifiers outside of any NAMESPACE declaration. For example, the HLA Standard Library does not define the symbols *malloc, free,* or *nl* (among others) within a NAMESPACE. However, you want to minimize such declarations in your libraries to avoid conflicts with names in your own programs. Often, you can choose a NAMESPACE identifier to complement your routine names. For example, the HLA Standard Libraries string copy routine was named after the equivalent C Standard Library function, *strcpy.* HLA's version is *str.cpy.* The actual function name is *cpy*; it happens to be a member of the *str* NAMESPACE, hence the full name *str.cpy* which is very similar to the comparable C function. The HLA Standard Library contains several examples of this convention. The *arg.c* and *arg.v* functions are another pair of such identifiers (corresponding to the C identifiers *argc* and *argv*).

Using a NAMESPACE in a header file is no different than using a NAMESPACE in a PROGRAM or UNIT. Here's an example of a typical header file containing a NAMESPACE declaration:

```
// myHeader.hhf -
//
// Routines supported in the myLibrary.lib file.

namespace myLib;
```

11. Or within an Iterator or Method, as you will see in later chapters.


```
    procedure func1; external;
    procedure func2; external;
    procedure func3; external;

end myLib;
```

Typically, you would compile each of the functions (func1..func3) as separate units (so each has its own object file and linking in one function doesn't link them all). Here's what a sample UNIT declaration for one of these functions:

```
unit func1Unit;
#includeonce( "myHeader.hhf" )

procedure myLib.func1;
begin func1;

    << code for func1 >>

end func1;

end func1Unit;
```

You should notice two important things about this unit. First, you do not put the actual *func1* procedure code within a NAMESPACE declaration block. By using the identifier *myLib.func1* as the procedure's name, HLA automatically realizes that this procedure declaration belongs in a name space. The second thing to note is that you do not preface *func1* with "myLib." after the BEGIN and END clauses in the procedure. HLA automatically associates the BEGIN and END identifiers with the PROCEDURE declaration, so it knows that these identifiers are part of the *myLib* name space and it doesn't make you type the whole name again.

Important note: when you declare external names within a name space, as was done in *func1Unit* above, HLA uses only the function name (*func1* in this example) as the external name. This creates a name space pollution problem in the external name space. For example, if you have two different name spaces, *myLib* and *yourLib* and they both define a *func1* procedure, the linker will complain about a duplicate definition for *func1* if you attempt to use functions from both these library modules. There is an easy work-around to this problem: use the extended form of the EXTERNAL directive to explicitly supply an external name for all external identifiers appearing in a NAMESPACE declaration. For example, you could solve this problem with the following simple modification to the myHeader.hhf file above:

```
// myHeader.hhf -
//
// Routines supported in the myLibrary.lib file.

namespace myLib;

    procedure func1; external( "myLib_func1" );
    procedure func2; external( "myLib_func2" );
    procedure func3; external( "myLib_func3" );

end myLib;
```

This example demonstrates an excellent convention you should adopt: when exporting names from a name space, always supply an explicit external name and construct that name by concatenating the NAMESPACE identifier with an underscore and the object's internal name.

The use of NAMESPACE declarations does not completely eliminate the problems of name space pollution (after all, the name space identifier is still a global object, as anyone who has included stdlib.hhf and attempted to define a "cs" variable can attest), but NAMESPACE declarations come pretty close to eliminating this problem. Therefore, you should use NAMESPACE everywhere practical when creating your own libraries.

9.10 Putting It All Together

Managing large projects is considerably easier if you break your program up into separate modules and work on them independently. In this chapter you learned about HLA's UNITs, include files, and the EXTERNAL directive. These provide the tools you need to break a program up into smaller modules. In addition to HLA's facilities, you'll also use a separate tool, nmake.exe, to automatically compile and link only those files that are necessary in a large project.

This chapter provided a very basic introduction to the use of makefiles and the make utility. Note that the MAKE programs are quite sophisticated. The presentation of the make program in this chapter barely scratches the surface. If you're interested in more information about MAKE facilities you should consult one of the excellent texts available on this subject. Lots of good information is also available on the Internet (just use the usual search tools).

In addition to breaking up large HLA projects, UNITs are also the basis for letting you write assembly language functions that you can call from high level languages like C/C++ and Delphi/Kylix. A later volume in this text will describe how you can use UNITs for this purpose.

