**Chapter Six**

**Directives & Pseudo-Opcodes**

1. **Directives and Pseudo-Opcodes**

Statements like mov ax,0 and add ax,bx are meaningless to the microprocessor. As arcane as these statements appear, they are still human readable forms of 80x86 instructions. The 80x86 responds to commands like B80000 and 03C3. An assembler is a program that converts strings like mov ax,0 to 80x86 machine code like ‘B80000’. An assembly language program consists of statements like mov ax,0 . The assembler converts an assembly language source Þle to machine code Ð the binary equivalent of the assembly language program. In this respect, the assembler program is much like a compiler; it reads an ASCII source file from the disk and produces a machine language program as output. The major difference between a compiler for a high level language (HLL) like Pascal and an assembler is that the compiler usually emits several machine instructions for each Pascal statement. The assembler generally emits a single machine instruction for each assembly language statement.

Attempting to write programs in machine language (i.e., in binary) is not particularly bright. This process is very tedious, prone to mistakes, and offers almost no advantages over programming in assembly language. The only major disadvantage to assembly language over pure machine code is that you must first assemble and link a program before you can execute it. However, attempting to assemble the code by hand would take far longer than the small amount of time that the assembler takes to perform the conversion for you.

There is another disadvantage to learning assembly language. An assembler like Microsoft’s Macro Assembler (MASM) provides a large number of features for assembly language programmers. Although learning about these features takes a fair amount of time, they are so useful that it is well worth the effort.

* 1. **Assembly Language Statements**

Assembly language statements in a source Þle use the following format:

{Label} {Mnemonic {Operand}} {;Comment}

Each entity above is a field. The four fields above are the *label field* , the *mnemonic field* , the *operand field* , and the *comment field*

.

The label Þeld is (usually) an optional Þeld containing a symbolic label for the current statement. Labels are used in assembly language, just as in HLLs, to mark lines as the targets of GOTOs (jumps). You can also specify variable names, procedure names, and other entities using symbolic labels. Most of the time the label Þeld is optional, meaning a label need be present only if you want a label on that particular line. Some mnemonics, however, require a label, others do not allow one. In general, you should always begin your labels in column one (this makes your programs easier to read).

A mnemonic is an instruction name (e.g., mov , add , etc.). The word mnemonic means memory aid. Mov is much easier to remember than the binary equivalent of the mov instruction! The braces denote that this item is optional. Note, however, that you cannot have an operand without a mnemonic.

The mnemonic field contains an assembler instruction. Instructions are divided into three classes: 80x86 machine instructions, assembler directives, and pseudo opcodes. 80x86 instructions, of course, are assembler mnemonics that correspond to the actual 80x86 instructions introduced in Chapter Five.

Assembler directives are special instructions that provide information to the assembler but do not generate any code. Examples include the segment directive, equ , assume , and end . These mnemonics are not valid 80x86 instructions. They are messages to the assembler, nothing else.

A pseudo-opcode is a message to the assembler, just like an assembler directive, however a pseudo opcode will emit object code bytes. Examples of pseudo-opcodes include byte , word , dword , qword , and tbyte . These instructions emit the bytes of data specified by their operands but they are not true 80X86 machine instructions.

The operand field contains the operands, or parameters, for the instruction specified in the mnemonic field. Operands never appear on lines by themselves. The type and number of operands (zero, one, two, or more) depend entirely on the specific instruction.

The comment field allows you to annotate each line of source code in your program. Note that the comment field always begins with a semicolon. When the assembler is processing a line of text, it completely ignores everything on the source line following a semicolon.

Each assembly language statement appears on its own line in the source file. You cannot have multiple assembly language statements on a single line. On the other hand, since all the fields in an assembly language statement are optional, blank lines are fine

.

You can use blank lines anywhere in your source file. Blank lines are useful for spacing out certain sections of code, making them easier to read. The Microsoft Macro Assembler is a free form assembler. The various fields of an assembly language statement may appear in any column (as long as they appear in the proper order). Any number of spaces or tabs can separate the various fields in the statement.

To the assembler, the following two code sequences are identical:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

mov ax, 0

mov bx, ax

add ax, dx

mov cx, ax

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

mov ax, 0

mov bx, ax

add ax, dx

mov cx, ax

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

The first code sequence is much easier to read than the second (if you don’t think so, perhaps you should go see a doctor!). With respect to readability, the judicial use of spacing within your program can make all the difference in the world.

Placing the labels in column one, the mnemonics in column 17 (two tab stops), the operand field in column 25 (the third tab stop), and the comments out around column 41or 49 (five or six tab stops) produces the best looking listings. Assembly language programs are hard enough to read as it is.

Formatting your listings to help make them easier to read will make them much easier to maintain. You may have a comment on the line by itself. In such a case, place the semicolon in column one and use the entire line for the comment, examples:

; The following section of code positions the cursor to the upper

; left hand position on the screen:

mov X, 0

mov Y, 0

; Now clear from the current cursor position to the end of the

; screen to clear the video display:

; etc.

* 1. **The Location Counter**

Recall that all addresses in the 80x86’s memory space consist of a segment address and an offset within that segment. The assembler, in the process of converting your source file into object code, needs to keep track of offsets within the current segment. The *location counter* is an assembler variable that handles this.

Whenever you create a segment in your assembly language source file (see segments later in this chapter), the assembler associates the current location counter value with it. The location counter contains the current offset into the segment. Initially (when the assembler Þrst encounters a segment) the location counter is set to zero. When encountering instructions or pseudo-opcodes, MASM increments the location counter for each byte written to the object code file. For example, MASM increments the location counter by two after encountering mov ax, bx since this instruction is two bytes long.

The value of the location counter varies throughout the assembly process. It changes for each line of code in your program that emits object code. We will use the term location counter to mean the value of the location counter at a particular statement before generating any code. Consider the following assembly language statements:

0 : or ah, 9

3 : and ah, 0c9h

6 : xor ah, 40h

9 : pop cx

A : mov al, cl

C : pop bp

D : pop cx

E : pop dx

F : pop ds

10: ret

The or , and , and xor instructions are all three bytes long; the mov instruction is two bytes long; the remaining instructions are all one byte long. If these instructions appear at the beginning of a segment, the location counter would be the same as the numbers that appear immediately to the left of each instruction above.

For example, the or instruction above begins at offset zero. Since the or instruction is three bytes long, the next instruction (and) follows at offset three. Likewise, and is three bytes long, so xor follows at offset six, etc.

* 1. **Symbols**

Consider the jmp instruction for a moment. This instruction takes the form:

jmp target

*Target* is the destination address. Imagine how painful it would be if you had to actuallyspecify the target memory address as a numeric value. If you’ve ever programmed inBASIC (where line numbers are the same thing as statement labels) you’ve experiencedabout 10% of the trouble you would have in assembly language if you had to specify thetarget of a jmpby an address.

To illustrate, suppose you wanted to jump to some group of instructions you’ve yet to write. What is the address of the target instruction? How can you tell until you’ve written every instruction before the target instruction? What happens if you change the program (remember, inserting and deleting instructions will cause the location counter values for all the following instructions within that segment to change). Fortunately, all these problems are of concern only to machine language programmers. Assembly language programmers can deal with addresses in a much more reasonable fashion - by using symbolic addresses.

A *symbol*, identifier*,* or *label* is a name associated with some particular value. This value can be an offset within a segment, a constant, a string, a segment address, an offset within a record, or even an operand for an instruction. In any case, a label provides us with the ability to represent some otherwise incomprehensible value with a familiar, mnemonic, name.

A symbolic name consists of a sequence of letters, digits, and special characters, with the following restrictions:

* A symbol cannot begin with a numeric digit.
* A name can have any combination of upper and lower case alphabetic characters. The assembler treats upper and lower case equivalently.
* A symbol may contain any number of characters, however only the first 31 are used. The assembler ignores all characters beyond the 31st.
* The \_, $, ?, and @ symbols may appear anywhere within a symbol. However, $ and ? are special symbols; you cannot create a symbol made up solely of these two characters.
* A symbol cannot match any name that is a reserved symbol. Students are expected to see reserved words in Elass.

In addition, all valid 80x86 instruction names and register names are reserved as well. Note that this list applies to Microsoft’s Macro Assembler version 6.0. Earlier versions of the assembler have fewer reserved words. Later versions may have more.

Some examples of valid symbols include:

L1 Bletch RightHere

Right\_Here Item1 \_\_Special

$1234 @Home $\_@1

Dollar$ WhereAmI? @1234

$1234 and @1234 are perfectly valid, strange though they may seem.

Some examples of illegal symbols include:

1TooMany - Begins with a digit.

Hello.There - Contains a period in the middle of the symbol.

$ - Cannot have $ or ? by itself.

LABEL - Assembler reserved word.

Right Here - Symbols cannot contain spaces.

Hi,There - or other special symbols besides \_, ?, $, and @.

Symbols, as mentioned previously, can be assigned numeric values (such as location counter values), strings, or even whole operands. To keep things straightened out, the assembler assigns a type to each symbol. Examples of types include near, far, byte, word, double word, quad word, text, and strings. How you declare labels of a certain type is the subject of much of the rest of this chapter. For now, simply note that the assembler always assigns some type to a label and will tend to complain if you try to use a label at some point where it does not allow that type of label.

* 1. **Literal Constants**

A *literal constant* is one whose value is implicit from the characters that make up the constant. Examples of literal constants include:

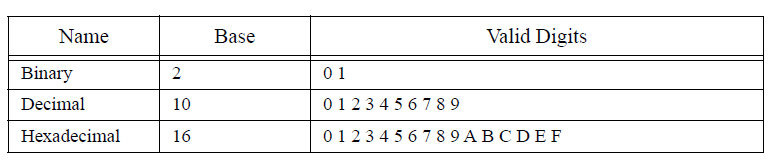
* 123
* 3.14159
* ‘Literal String Constant’
* 0FABCh
* ‘A’

Most of these literal constants should be reasonably familiar to anyone who has written a program in a high level language like Pascal or C++. Text constants are special forms of strings that allow textual substitution during assembly.

A literal constant’s representation corresponds to what we would normally expect for its real world value. “Literal constants “are also known as *non symbolic constants* since they use the value’s actual representation, rather than some symbolic name, within your program. MASM also lets you define symbolic, or *manifest*, constants in a program, but more on that later.

* + 1. **Integer Constants**

An integer constant is a numeric value that can be specified in binary, decimal, or hexadecimal. The choice of the base (or radix) is up to you. The following table shows the legal digits for each radix:



To differentiate between numbers in the various bases, you use a suffix character. If you terminate a number with a ‘b’ or ‘B’, then MASM assumes that it is a binary number. If it contains any digits other than zero or one the assembler will generate an error. If the suffix is‘t’, ‘T’, ‘d’ or ‘D’, then the assembler assumes that the number is a decimal (base 10) value. A suffix of ‘h’ or ‘H’ will select the hexadecimal radix.

* + 1. **String Constants**

A string constant is a sequence of characters surrounded by apostrophes or quotation marks.

Examples:

"This is a string"

'So is this'

You may freely place apostrophes inside string constants enclosed by quotation marks and vice versa. If you want to place an apostrophe inside a string delimited by apostrophes, you must place a pair of apostrophes next to each other in the string, e.g.,

'Doesn''t this look weird?'

Quotation marks appearing within a string delimited by quotes must also be doubled up,

e.g.,

"Microsoft claims ""Our software is very fast."" Do you believe them?"

Although you can double up apostrophes or quotes as shown in the examples above, the easiest way to include these characters in a string is to use the *other* character as the string delimiter:

“Doesn’t this look weird?”

‘Microsoft claims “Our software is very fast.” Do you believe them?’

The only time it would be absolutely necessary to double up quotes or apostrophes in a string is if that string contained *both* symbols. This rarely happens in real programs.

Like the C and C++ programming languages, there is a subtle difference between a character value and a string value. A single character (that is, a string of length one) may appear anywhere MASM allows an integer constant or a string. If you specify a character constant where MASM expects an integer constant, MASM uses the ASCII code of that character as the integer value. Strings (whose length is greater than one) are allowed only within certain contexts.

* + 1. **Declaring Manifest Constants Using Equates**

A manifest constant is a symbol name that represents some Þxed quantity during the assembly process. That is, it is a symbolic name that represents some value. *Equates* are the mechanism MASM uses to declare symbolic constants. Equates take three basic forms:

symbol equ expression

symbol = expression

symbol textequ expression

The expression operand is typically a numeric expression or a text string. The symbol is given the value and type of the expression. The equ and ‘=‘ directives have been with MASM since the beginning. Microsoft added the textequ directive starting with MASM 6.0.

The purpose of the ‘=‘ directive is to deÞne symbols that have an integer (or single character) quantity associated with them. This directive does not allow real, string, or text operands. This is the primary directive you should use to create numeric symbolic constants in your programs.

Some examples:

NumElements = 16

.

.

.

Array byte NumElements dup (?)

.

.

.

mov cx, NumElements

mov bx, 0

ClrLoop: mov Array[bx], 0

inc bx

loop ClrLoop

* 1. **Procedures**

Unlike HLLs, MASM doesn’t enforce strict rules on exactly what constitutes a procedure4. You can call a procedure at any address in memory. The first ret instruction encountered along that execution path terminates the procedure. Such expressive freedom, however, is often abused yielding programs that are very hard to read and maintain. Therefore, MASM provides facilities to declare procedures within your code. The basic mechanism for declaring a procedure is:

procname proc {NEAR or FAR}

<statements>

procname endp

As you can see, the definition of a procedure looks similar to that for a segment. One difference is that procname (that is the name of the procedure you’re defining) must be a unique identifier within your program. Your code calls this procedure using this name, it wouldn’t do to have another procedure by the same name; if you did, how would the program determine which routine to call?

Proc allows several different operands, though we will only consider three: the single keyword near, the single keyword far, or a blank operand Þeld5. MASM uses these operands to determine if you’re calling this procedure with a near or far call instruction. They also determine which type of ret instruction MASM emits within the procedure. Consider the following two procedures:

NProc proc near

mov ax, 0

ret

NProc endp

FProc proc far

mov ax, 0FFFFH

ret

FProc endp

and:

call NPROC

call FPROC

The assembler automatically generates a three-byte (near) call for the Þrst call instruction above because it knows that NProc is a near procedure. It also generates a Þve-byte (far) call instruction for the second call because FProc is a far procedure. Within the procedures themselves, MASM automatically converts all ret instructions to near or far returns depending on the type of routine.

Note that if you do not terminate a proc/endp section with a ret or some other transfer of control instruction and program flow runs into the endp directive, execution will continue with the next executable instruction following the endp. For example, consider the following:

Proc1 proc

mov ax, 0

Proc1 endp

Proc2 proc

mov bx, 0FFFFH

ret

Proc2 endp

If you call Proc1, control will ßow on into Proc2 starting with the mov bx,0FFFFh instruction. Unlike high level language procedures, an assembly language procedure does not contain an implicit return instruction before the endp directive. So always be aware of how the proc/endp directives work.

There is nothing special about procedure declarations. They’re a convenience provided by the assembler, nothing more. You could write assembly language programs for the rest of your life and never use the proc and endp directives. Doing so, however, would be poor programming practice. Proc and endp are marvelous documentation features which, when properly used, can help make your programs much easier to read and maintain.

MASM versions 6.0 and later treat all statement labels inside a procedure as *local*. That is, you cannot refer directly to those symbols outside the procedure.

* 1. **Segments**

All programs consist of one or more segments. Of course, while your program is running, the 80x86’s segment registers point at the currently active segments. On 80286 and earlier processors, you can have up to four active segments at once (code, data, extra, and stack); on the 80386 and later processors, there are two additional segment registers: fs and gs. Although you cannot access data in more than four or six segments at any one given instant, you can modify the 80x86’s segment registers and point them at other segments in memory under program control. This means that a program can access more than four or six segments. The question is “how do you create these different segments in a program and how do you access them at run-time?”

Segments, in your assembly language source file, are defined with the segment and ends directives. You can put as many segments as you like in your program. Well, actually you are limited to 65,536 different segments by the 80x86 processors and MASM probably doesn’t even allow that many, but you will probably never exceed the number of segments MASM allows you to put in your program. When MS-DOS begins execution of your program, it initializes two segment registers. It points cs at the segment containing your main program and it points ss at your stack segment. From that point forward, you are responsible for maintaining the segment registers yourself.

To access data in some particular segment, an 80x86 segment register must contain the address of that segment. If you access data in several different segments, your program will have to load a segment register with that segment’s address before accessing it. If you are frequently accessing data in different segments, you will spend considerable time reloading segment registers. Fortunately, most programs exhibit locality of reference when accessing data. This means that a piece of code will likely access the same group of variables many times during a given time period. It is easy to organize your programs so that variables you often access together appear in the same segment. By arranging your programs in this manner, you can minimize the number of times you need to reload the segment registers. In this sense, a segment is nothing more than a cache of often accessed data.

In real mode, a segment can be up to 64 Kilobytes long. Most pure assembly language programs use less than 64K code, 64K global data, and 64K stack space. Therefore, you can often get by with no more than three or four segments in your programs

A segment in your file should take the following form:

segmentname segment

statements

segmentname ends

* + 1. **Segment Names**

Whenever you specify a segment name as an operand to an instruction, MASM will use the immediate addressing mode and substitute the address of that segment for its name. Since you cannot load an immediate value into a segment register with a single instruction, loading the segment address into a segment register typically takes two instructions. For example, the following three instructions appear at the beginning of the any assembly code (in Elass), they initialize the ds and es registers so they point at the dseg segment:

mov ax, dseg ;Loads ax with segment address of dseg.

mov ds, ax ;Point ds at dseg.

mov es, ax ;Point es at dseg.

The other purpose for segment names is to provide the segment component of a variable name. Remember, 80x86 addresses contain two components: a segment and an offset. Since the 80x86 hardware defaults most data references to the data segment, it is common practice among assembly language programmers to do the same thing; that is, not bother to specify a segment name when accessing variables in the data segment. In fact, a full variable reference consists of the segment name, a colon, and the offset name:

mov ax, dseg:Item1

mov dseg:Item2, ax

Technically, you should preÞx all your variables with the segment name in this fashion. However, most programmer’s don’t bother because of the extra typing involved. Most of the time you can get away with this; however, there are a few times when you really will need to specify the segment name. Fortunately, those situations are rare and only occur in very complex programs, not the kind you’re likely to run into for a while.

It is important that you realize that specifying a segment name before a variable’s name does not mean that you can access data in a segment without having some segment register pointing at that segment. Except for the jmp and call instructions, there are no 80x86 instructions that let you specify a full 32 bit segmented direct address. All other memory references use a segment register to supply the segment component of the address.

* + 1. **Segment Prefixes**

When the 80x86 references a memory operand, it usually references a location within the current data segment8. However, you can instruct the 80x86 microprocessor to reference data in one of the other segments using a segment prefix before an address expression.

A segment prefix is either ds:, cs:, ss:, es:, fs:, or gs:. When used in front of an address expression, a segment prifix instructs the 80x86 to fetch its memory operand from the specified segment rather than the default segment. For example, mov ax, cs:I[bx] loads the accumulator from address I+bx *within the current code segment*. If the cs: prefix were absent, this instruction would normally load the data from the current data segment. Likewise, mov ds:[bp],ax stores the accumulator into the memory location pointed at by the bp register in the current data segment (remember, whenever using bp as a base register it points into the stack segment).

Segment prefixes are instruction opcodes. Therefore, whenever you use a segment prefix you are increasing the length (and decreasing the speed) of the instruction utilizing the segment prefix. Therefore, you don’t want to use segment prefixes unless you have a good reason to do so.

* + 1. **Why Even Bother With Segments?**

After reading the previous sections, you’re probably wondering what possible good could come from using segments in your programs. As a beginning assembly language programmer, it’s probably a good idea to ignore much of this discussion on segmentation until you are much more comfortable with 80x86 assembly language programming. However, there are three reasons you’ll want to learn more about segmentation if you continue writing assembly language programs for any length of time: the real-mode 64K segment limitation, program modularity, and interfacing with high level languages.

When operating in real mode, segments can be a maximum of 64 kilobytes long. If you need to access more than 64K of data or code in your programs, you will need to use more than one segment. This fact, more than any other reason, has dragged programmers (kicking and screaming) into the world of segmentation. Unfortunately, this is as far as many programmers get with segmentation. They rarely learn more than just enough about segmentation to write a program that accesses more than 64K of data. As a result, when a segmentation problem occurs because they don’t fully understand the concept, they blame segmentation for their problems and they avoid using segmentation as much as possible.

This is too bad because segmentation is a powerful memory management tool that lets you organize your programs into logical entities (*segments*) that are, in theory, independent of one another. The field of software engineering studies how to write correct, large programs. Modularity and independence are two of the primary tools software engineers use to write large programs that are correct and easy to maintain. The 80x86 family provides, in hardware, the tools to implement segmentation. On other processors, segmentation is enforced strictly by software. As a result, it is easier to work with segments on the 80x86 processors.

Although this text does not deal with protected mode programming, it is worth pointing out that when you operate in protected mode on 80286 and later processors, the 80x86 hardware can actually prevent one module from accessing another module’s data (indeed, the term “protected mode” means that segments are protected from illegal access). Many debuggers available for MS-DOS operate in protected mode allowing you to catch array and segment bounds violations. Soft-ICE and Bounds Checker from NuMega are examples of such products. Most people who have worked with segmentation in a protected mode environment (e.g., OS/2 or Windows) appreciate the benefits that segmentation offers.

Another reason for studying segmentation on the 80x86 is because you might want to write an assembly language function that a high level language program can call. Since the HLL compiler makes certain assumptions about the organization of segments in memory, you will need to know a little bit about segmentation in order to write such code.

* 1. **The END Directive**

The end directive terminates an assembly language source file. In addition to telling MASM that it has reached the end of an assembly language source file, the end directive’s optional operand tells MS-DOS where to transfer control when the program begins execution; that is, you specify the name of the main procedure as an operand to the end directive. If the end directive’s operand is not present, MS-DOS will begin execution starting at the first byte in the .exe file. Since it is often inconvenient to guarantee that your main program begins with the first byte of object code in the .exe file, most programs specify a starting location as the operand to the end directive.

If you are using separate assembly and you’re linking together several different object code files (see “Managing Large Programs” on the coming sections), only one module can have a main program. Likewise, only one module should specify the starting location of the program. If you specify more than one starting location, you will confuse the linker and it will generate an error.

* 1. **Variables**

Global variable declarations use the byte/sbyte/db, word/sword/dw, dword/sdword/dd, qword/dq, and tbyte/dt pseudo-opcodes. Although you can place your variables in any segment (including the code segment), most beginning assembly language programmers place all their global variables in a single data segment.

A typical variable declaration takes the form:

varname byte initial\_value

Varname is the name of the variable you’re declaring and initial\_value is the initial value you want that variable to have when the program begins execution. ‘?’ is a special initial value. It means that you don’t want to give a variable an initial value. When DOS loads a program containing such a variable into memory, it does not initialize this variable to any particular value.

The declaration above reserves storage for a single byte. This could be changed to any other variable type by simply changing the byte mnemonic to some other appropriate pseudo-opcode.

For the most part, this text will assume that you declare all variables in a *data segment*, that is, a segment that the 80x86’s ds register will point at. In particular, most of the programs herein will place all variables in the DSEG segment (CSEG is for code, DSEG is for data, and SSEG is for the stack).

Since the previous chapters covers the declaration of variables, data types, structures and arrays in depth, this chapter will not waste any more time discussing this subject.

* 1. **Label Types**

One unusual feature of Intel syntax assemblers (like MASM) is that they are *strongly typed*. A strongly typed assembler associates a certain type with symbols declared appearingin the source file and will generate a warning or an error message if you attempt to usethat symbol in a context that doesn’t allow its particular type. Although unusual in anassembler, most high level languages apply certain typing rules to symbols declared in thesource file. Pascal, of course, is famous for being a strongly typed language. You cannot,in Pascal, assign a string to a numeric variable or attempt to assign an integer value to aprocedure label. Intel, in designing the syntax for 8086 assembly language, decided thatall the reasons for using a strongly typed language apply to assembly language as well asPascal. Therefore, standard Intel syntax 80x86 assemblers, like MASM, impose certaintype restrictions on the use of symbols within your assembly language programs.

* 1. **Address Expressions**

An *address expression* is an algebraic expression that produces a numeric result that MASM merges into the displacement Þeld of an instruction. An integer constant is probably the simplest example of an address expression. The assembler simply substitutes the value of the numeric constant for the speciÞed operand. For example, the following instruction Þlls the immediate data Þelds of the mov instruction with zeros:

mov ax, 0

Another simple form of an addressing mode is a symbol. Upon encountering a symbol, MASM substitutes the value of that symbol. For example, the following two statements emit the same object code as the instruction above:

Value equ 0

mov ax, Value

An address expression, however, can be much more complex than this. You can use various arithmetic and logical operators to modify the basic value of some symbols or constants.

Keep in mind that MASM computes address expressions during assembly, not at run time. For example, the following instruction does not load ax from location Var and add one to it:

mov ax, Var1+1

Instead, this instruction loads the al register with the byte stored at the address of Var1 plus one and then loads the ah register with the byte stored at the address of Var1 plus two.

Beginning assembly language programmers often confuse computations done at assembly time with those done at run time. Take extra care to remember that MASM computes all address expressions at assembly time!

* 1. **Coercion**

Consider the following program segment:

DSEG segment public 'DATA'

I byte ?

J byte ?

DSEG ends

CSEG segment

.

.

.

mov al, I

mov ah, J

.

.

.

CSEG ends

Since I and J are adjacent, there is no need to use two mov instructions to load al and ah, a simple mov ax, I instruction would do the same thing. Unfortunately, the assembler will balk at mov ax, I since I is a byte. The assembler will complain if you attempt to treat it as a word. As you can see, however, there are times when you’d probably like to treat a byte variable as a word (or treat a word as a byte or double word, or treat a double word as a something else)

.

Temporarily changing the type of a label for some particular occurrence is *coercion*. Expressions can be coerced to a different type using the MASM ptr operator. You use the ptr operator as follows:

*type* PTR *expression*

*Type* is any of byte, word, dword, tbyte, near, far, or other type and *expression* is any general expression that is the address of some object. The coercion operator returns an expression with the same value as *expression*, but with the type specified by *type*. To handle the above problem you’d use the assembly language instruction:

mov ax, word ptr I

This instructs the assembler to emit the code that will load the ax register with the word at address I. This will, of course, load al with I and ah with J.

Code that uses double word values often makes extensive use of the coercion operator. Since lds and les are the only 32-bit instructions on pre-80386 processors, you cannot (without coercion) store an integer value into a 32-bit variable using the mov instruction on those earlier CPUs. If you’ve declared DBL using the dword pseudo-opcode, then an instruction of the form mov DBL,ax will generate an error because it’s attempting to move a 16 bit quantity into a 32 bit variable. Storing values into a double word variable requires the use of the ptr operator. The following code demonstrates how to store the ds and bx registers into the double word variable DBL:

mov word ptr DBL, bx

mov word ptr DBL+2, ds

You will use this technique often as various UCR Standard Library and MS-DOS calls return a double word value in a pair of registers.

**Warning**: If you coerce a jmp instruction to perform a far jump to a near label, other than performance degradation (the far jmp takes longer to execute), your program will work Þne. If you coerce a call to perform a far call to a near subroutine, you’re headed for trouble. Remember, far calls push the cs register onto the stack (with the return address). When executing a near ret instruction, the old cs value will not be popped off the stack, leaving junk on the stack. The very next pop or ret instruction will not operate properly since it will pop the cs value off the stack rather than the original value pushed onto the

Stack.

Expression coercion can come in handy at times. Other times it is essential. However, you shouldn’t get carried away with coercion since data type checking is a powerful debugging tool built in to MASM. By using coercion, you override this protection provided by the assembler. Therefore, always take care when overriding symbol types with the ptr operator.

One place where you’ll need coercion is with the mov memory, immediate instruction. Consider the following instruction:

mov [bx], 5

Unfortunately, the assembler has no way of telling whether bx points at a byte, word, or double word item in memory. The value of the immediate operand isn’t of any use. Even though Þve is a byte quantity, this instruction might be storing the value 0005h into a word variable, or 00000005 into a double word variable. If you attempt to assemble this statement, the assembler will generate an error to the effect that you must specify the size of the memory operand. You can easily accomplish this using the byte ptr, word ptr, and dword ptr operators as follows:

mov byte ptr [bx], 5 ;For a byte variable

mov word ptr [bx], 5 ;For a word variable

mov dword ptr [bx], 5 ;For a dword variable

Lazy programmers might complain that typing strings like “word ptr” or “far ptr” is too much work. Wouldn’t it have been nice had Intel chosen a single character symbol rather than these long phrases? Students are expected to dig for short hand forms of coercion.

* 1. **Conditional Assembly**

MASM provides a very powerful conditional assembly facility. With conditional assembly, you can decide, based on certain conditions, whether MASM will assemble the code. There are several conditional assembly directives; the following section covers most of them.

It is important that you realize that these directives evaluate their expressions at *assembly time*, not at run time. The if conditional assembly directive is not the same as a Pascal or C “if” statement. If you are familiar with C, the #ifdef directive in C is roughly’ equivalent to some of MASM’s conditional assembly directives.

MASM’s conditional assembly directives are important because they let you generate different object code for different operating environments and different situations. For example, suppose you want to write a program that will run on all machines but you would like to optimize the code for 80386 and later processors. Obviously, you cannot execute 80386 code on an 8086 processor, so how can you solve this problem?

One possible solution is to determine the processor type at run time and execute different sections of code in the program depending on the presence or absence of a 386 or later CPU. The problem with this approach is that your program needs to contain two code sequences Ð an optimal 80386 sequence and a compatible 8086 sequence. On any given system the CPU will only execute one of these code sequences in the program, so the other sequence will be wasting memory and may have adverse affects on any cache in the system.

A second possibility is to write two versions of the code, one that uses only 8086 instructions and one that uses the full 80386 instruction set. During installation, the user (or the installation program) selects the 80386 version if they have an 80386 or later processor. Otherwise they select the 8086 version. While this marginally increases the cost of the software since it will require more disk space, the program will consume less memory while running. The problem with this approach is that you will need to maintain *two* separate versions of the program. If you correct a bug in the 8086 version of the code, you will probably need to correct that same bug in the 80386 program. Maintaining multiple source files is a difficult task.

A third solution is to use *conditional assembly*. With conditional assembly, you can merge the 8086 and 80386 versions of the code into the same source file. During assembly, you can *conditionally* choose whether MASM assembles the 8086 or the 80386 version. By assembling the code twice, you can produce an 8086 and an 80386 version of the code. Since both versions of the code appear in the same source file, the program will be much easier to maintain since you will not have to correct the same bug in two separate source files. You *may* need to correct the same bug twice in two separate code sequences in the program, but generally the bug will appear in two adjacent code sequences, so it is less likely that you will forget to make the change in both places.

MASM’s conditional assembly directives are especially useful within *macros*. They can help you produce efficient code when a macro would normally produce sub-optimal code. For more information about macros and how you can use conditional assembly within a macro.

Macros and conditional assembly actually provide “a programming language within a programming language.” Macros and conditional assembly let you write programs (in the “macro language”) that write segments of assembly language code for you. This introduces an independent way to generate bugs in your application programs. Not only can a bug develop in your assembly language code, you can also introduce bugs in your macro code (e.g., conditional assembly), that wind up producing bugs in your assembly language code. Keep in mind that if you get too sophisticated when using conditional assembly, you can produce programs that are very difficult to read, understand, and debug.

* 1. **Macros**

A macro is like a procedure that inserts a block of statements at various points in your program during assembly. There are three general types of macros that MASM supports: procedural macros, functional macros, and looping macros. Along with conditional assembly, these tools provide the traditional if, loop, procedure, and function constructs found in many high level languages. Unlike the assembly instructions you write, the conditional assembly and macro language constructs execute *during assembly*. The conditional assembly and macros statements do not exist when your assembly language program is running. The purpose of these statements is to control which statements MASM assembles into your Þnal “.exe” file. While the conditional assembly directives select or omit certain statements for assembly, the macro directives let you emit repetitive sequences of instructions to an assembly language Þle like high level language procedures and loops let you repetitively execute sequences of high level language statements.

* 1. **Managing Large Programs**

Most assembly language programs are not totally stand alone programs. In general, you will call various standard library or other routines which are not deÞned in your main program. For example, you’ve probably noticed by now that the 80x86 doesn’t provide any instructions like ‘read’, ‘write’, or “printf for doing I/O operations. In fact, the only instructions you’ve seen that do I/O include the 80x86 in and out instructions, which are really just special mov instructions, and the echo/%out directives that perform assembly- time output, not the run-time output you want. Is there no way to do I/O from assembly language? Of course there is. You can write procedures that perform the I/O operations like ‘read’ and ‘write’. Unfortunately, writing such routines is a complex task, and beginning assembly language programmers are not ready for such tasks. That’s where the UCR Standard Library for 80x86 Assembly Language Programmers comes in. This is a package of procedures you can call to perform simple I/O operations like ‘printf’.

The UCR Standard Library contains 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 programs. Fortunately, you don’t have to.

For small programs, working with a single source Þle is Þne. For large programs this gets very cumbersome (consider the example above of having to include the entire UCR 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 UCR Standard Library, for example, takes several minutes to assemble, even on a fast machine. Imagine having to wait Þve or ten minutes on a fast Pentium machine to assemble a program to which you’ve made a one line change!

As with HLLs, the solution is *separate compilation* (or *separate assembly* in MASM’s case). First, you break up your large source Þles into manageable chunks. Then you assemble the separate Þles 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 UCR Standard Library works in precisely this way. The Standard Library is already assembled 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 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 one of the big techniques for ‘programming in the large.’ The following sections describe the tools MASM provides for separate compilation and how to effectively employ these tools in your programs.

* + 1. **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. This allows you to construct text files containing common equates, macros, source code, and other assembler 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 DOS filename. MASM merges the specified file into the assembly 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.

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 jointhese modules together when you assemble your file. The following example wouldinclude the PRINTF.ASM and PUTC.ASM files during the assembly of your program:

include printf.asm

include putc.asm

<Code for your program goes here>

End

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 thefinclude during assembly, exactly as though you had typed that code in yourself.

MASM still has to assemble the code and that takes time. Were you to include all the files for the Standard Library routines, your assemblies would take *forever*. In general, you should *not* use the include directive to include source code as shown above. Instead, you should use the include directive to insert a common set of constants (equates), macros, 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). The purpose of using include files in this manner will become clearer after you see how the public and external declarations work.

* + 1. **The PUBLIC, EXTERN, and EXTRN Directives**

Technically, the include directive provides you with all the facilities you need to create modular programs. You can build up a library of modules, each containing some speciÞc routine, and include any necessary modules into an assembly language program using the appropriate include commands. MASM (and the accompanying LINK program) 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 an assembly wastes a lot of time since MASM must reassemble 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 public and extern directives provide for you. Extrn is an older directive that is a synonym for extern. It provides compatibility with old source Þles. You should always use the extern directive in new source code.

To use the public and extern 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. To demonstrate, consider the following two modules:

;Module #1:

public Var1, Var2, Proc1

DSEG segment para public 'data'

Var1 word ?

Var2 word ?

DSEG ends

CSEG segment para public 'code'

assume cs:cseg, ds:dseg

Proc1 proc near

mov ax, Var1

add ax, Var2

mov Var1, ax

ret

Proc1 endp

CSEG ends

End

;Module #2:

extern Var1:word, Var2:word, Proc1:near

CSEG segment para public 'code'

.

.

.

mov Var1, 2

mov Var2, 3

call Proc1

.

.

.

CSEG ends

End

Module #2 references Var1, Var2, and Proc1, yet these symbols are external to module #2. Therefore, you must declare them external with the extern directive. This directive takes the following form:

extern name:type {,name:type...}

Name is the name of the external symbol, and type is the type of that symbol. Type may be any of near, far, proc, byte, word, dword, qword, tbyte, abs (absolute, which is a constant), or some other user defined type.

The current module uses this type declaration. Neither MASM nor the linker checks the declared type against the module defining name to see if the types agree. Therefore, you must exercise caution when defining external symbols. The public directive lets you export a symbol’s value to external modules. A public declaration takes the form:

public name {,name ...}

Each symbol appearing in the operand field of the public statement is available as an external symbol to another module. Likewise, all external symbols within a module must appear within a public statement in some other module.

Once you create the source modules, you should assemble the file containing the public declarations first.

This produces a “pubs.obj” object module. Next, assemble the file containing the external definitions and link in the code. Assuming there are no errors, this will produce a file “exts.exe” which is the linked and executable form of the program.

Note that the extern directive defines a symbol in your source file. Any attempt to redefine that symbol elsewhere in your program will produce a “duplicate symbol” error.

* + 1. **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.asm and pgmb.asm. Also suppose that you’ve already assembled both modules so that the files pgma.obj andfpgmb.obj exist. Finally, you make changes to pgma.asm and pgmb.asm and assemble the pgma.asm *but forget to assemble the*f*pgmb.asm file.* Therefore, the pgmb.obj file will be *out of date* since this object file does not reflect the changes made to the pgmb.asm file. If you link the program’s modules together, the resulting .exe file will only contain the changes to the pgma.asm file, it will not have the updated object code associated with pgmb.asm. As projects get larger, as they 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 reassemble (or recompile) *all* modules in a project, even if many of the .obj 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. The make program, will a little help from you, can figure out which files need to be reassemble and which files have up to date .obj Þles. 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 assembly-time dependencies between files. An .exe file, for example, is *dependent* on the source code whose assembly produces 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 .exe file. Typical dependencies include the following:

* An executable file (.exe) generally depends only on the set of object files (.obj) that the linker combines to form the executable.
* A given object code file (.obj) depends on the assembly language source files that were assembled to produce that object file. This includes the assembly language source Þles (.asm) and any files included during that assembly (generally .a 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

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 generate of a new pgm.exe file.

The nmake.exe 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, MS-DOS and Windows will update a *modification time and date* associated with the file. The nmake.exe 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 nmake.exe assumes that some operation must be necessary to update the dependent file.

When an update is necessary, nmake.exe executes the set of (MS-DOS) 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 tab stop. The pgm.exe statement above would probably look something like the following:

pgm.exe: pgma.obj pgmb.obj

elink Fepgm.exe pgma.obj pgmb.obj