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 file to machine code – 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.

### 8.0 Chapter Overview

Like Chapter Six, much of the information in this chapter is reference material. Like any reference section, some knowledge is essential, other material is handy, but optional, and some material you may never use while writing programs. The following list outlines the information in this text. A “•” symbol marks the essential material. The “o” symbol marks the optional and lesser used subjects.

- Assembly language statement source format
  - The location counter
  - Symbols and identifiers
  - Constants
  - Procedure declarations
- Segments in an assembly language program
- Variables
- Symbol types
- Address expressions (later subsections contain advanced material)
- Conditional assembly
- Macros
- Listing directives
- Separate assembly

### 8.1 Assembly Language Statements

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

```
[Label]  [Mnemonic  [Operand]}  ;Comment
```

Page 355
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 field is (usually) an optional field 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 field 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 Six.

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
```

1. Unless, of course, the semicolon appears inside a string constant.
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 tabstops), the operand field in column 25 (the third tabstop), and the comments out around column 41 or 49 (five or six tabstops) 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.
```

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

\[
\text{jmp target}
\]

Target is the destination address. Imagine how painful it would be if you had to actually specify the target memory address as a numeric value. If you’ve ever programmed in BASIC (where line numbers are the same thing as statement labels) you’ve experienced about 10% of the trouble you would have in assembly language if you had to specify the target of a jmp by 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. The following symbols are reserved:
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.

### 8.4 Literal Constants

The Microsoft Macro Assembler (MASM) is capable of processing five different types of constants: integers, packed binary coded decimal integers, real numbers, strings, and text. In this chapter we'll consider integers, reals, strings, and text only. For more information about packed BCD integers please consult the Microsoft Macro Assembler Programmer's Guide.

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’
- <Text Constant>
Except for the last example above, 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.

### 8.4.1 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Base</th>
<th>Valid Digits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>2</td>
<td>0 1</td>
</tr>
<tr>
<td>Decimal</td>
<td>10</td>
<td>0 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>16</td>
<td>0 1 2 3 4 5 6 7 8 9 A B C D E F</td>
</tr>
</tbody>
</table>

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.

All integer constants must begin with a decimal digit, including hexadecimal constants. To represent the value “FDED” you must specify 0FDEDh. The leading decimal digit is required by the assembler so that it can differentiate between symbols and numeric constants; remember, “FDEDh” is a perfectly valid symbol to the Microsoft Macro Assembler.

Examples:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0F000h</td>
<td>12345d</td>
<td>0110010100b</td>
</tr>
<tr>
<td>1234h</td>
<td>100h</td>
<td>08h</td>
</tr>
</tbody>
</table>

If you do not specify a suffix after your numeric constants, the assembler uses the current default radix. Initially, the default radix is decimal. Therefore, you can usually specify decimal values without the trailing “D” character. The radix assembler directive can be used to change the default radix to some other base. The .radix instruction takes the following form:

```
.radix  base  ;Optional comment
```

*Base* is a decimal value between 2 and 16.

The .radix statement takes effect as soon as MASM encounters it in the source file. All the statements before the .radix statement will use the previous default base for numeric constants. By sprinkling multiple .radix instructions throughout your source file, you can switch the default base amongst several values depending upon what’s most convenient at each point in your program.

Generally, decimal is fine as the default base so the .radix instruction doesn’t get used much. However, faced with entering a gigantic table of hexadecimal values, you can save

---

2. Actually, you can also specify the octal (base 8) radix. We will not use octal in this text.
a lot of typing by temporarily switching to base 16 before the table and switching back to
decimal after the table. Note: if the default radix is hexadecimal, you should use the “T”
suffix to denote decimal values since MASM will confuse the “D” suffix with a hexadecimal
digit.

8.4.2 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.

8.4.3 Real Constants

Within certain contexts, you can use floating point constants. MASM allows you to express floating point constants in one of two forms: decimal notation or scientific notation. These forms are quite similar to the format for real numbers that Pascal, C, and other HLLs use.

The decimal form is just a sequence of digits containing a decimal point in some position of the number:

1.0 3.14159 625.25 -128.0 0.5

Scientific notation is also identical to the form used by various HLLs:

1e5 1.567e-2 -6.02e-10 5.34e+12

The exact range of precision of the numbers depend on your particular floating point package. However, MASM generally emits binary data for the above constants that is compatible with the 80x87 numeric coprocessors. This form corresponds to the numeric format specified by the IEEE standard for floating point values. In particular, the constant 1.0 is not the binary equivalent of the integer one.
8.4.4 Text Constants

Text constants are not the same thing as string constants. A textual constant substitutes verbatim during the assembly process. For example, the characters 5[bx] could be a textual constant associated with the symbol VAR1. During assembly, an instruction of the form mov ax, VAR1 would be converted to the instruction mov ax, 5[bx].

Textual equates are quite useful in MASM because MASM often insists on long strings of text for some simple assembly language operands. Using text equates allows you to simplify such operands by substituting some string of text for a single identifier in a statement.

A text constant consists of a sequence of characters surrounded by the “<” and “>” symbols. For example the text constant 5[bx] would normally be written as <5[bx]>. When the text substitution occurs, MASM strips the delimiting “<” and “>” characters.

8.5 Declaring Manifest Constants Using Equates

A manifest constant is a symbol name that represents some fixed 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 define 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

Count textequ <6[bp]>
DataPtr textequ <8[bp]>
les bx, DataPtr ; Same as les bx, 8[bp]
mov cx, Count ; Same as mov cx, 6[bp]
mov al, 0
ClrLp: mov es:[ bx ], al
       inc bx
       loop ClrLp
```
Note that it is perfectly legal to equate a symbol to a blank operand using an equate like the following:

```
BlankEqu      textequ <>
```

The purpose of such an equate will become clear in the sections on conditional assembly and macros.

The `equ` directive provides almost a superset of the capabilities of the `="` and `textequ` directives. It allows operands that are numeric, text, or string literal constants. The following are all legal uses of the `equ` directive:

```
One          equ     1
Minus1       equ     -1
TryAgain     equ     'Y'
StringEqu    equ     "Hello there"
TxtEqu       equ     $4[si]

HTString     byte    StringEqu ;Same as HTString equ "Hello there"

mov ax, TxtEqu ;Same as mov ax, 4[si]
mov bl, One ;Same as mov bl, 1
cmp al, TryAgain ;Same as cmp al, 'Y'
```

Manifest constants you declare with equates help you parameterize a program. If you use the same value, string, or text, multiple times within a program, using a symbolic equate will make it very easy to change that value in future modifications to the program. Consider the following example:

```
Array        byte    16 dup (?)

mov cx, 16
mov bx, 0
ClrLoop:     mov Array[bx], 0
inc bx
loop C1rLoop
```

If you decide you want `Array` to have 32 elements rather than 16, you will need to search throughout your program an locate every reference to this data and adjust the literal constants accordingly. Then there is the possibility that you missed modifying some particular section of code, introducing a bug into your program. On the other hand, if you use the `NumElements` symbolic constant shown earlier, you would only have to change a single statement in your program, reassemble it, and you would be in business; MASM would automatically update all references using `NumElements`.

MASM lets you redefine symbols declared with the `="` directive. That is, the following is perfectly legal:

```
SomeSymbol   = 0

SomeSymbol   = 1
```

Since you can change the value of a constant in the program, the symbol’s scope (where the symbol has a particular value) becomes important. If you could not redefine a symbol, one would expect the symbol to have that constant value everywhere in the program. Given that you can redefine a constant, a symbol’s scope cannot be the entire program. The solution MASM uses is the obvious one, a manifest constant’s scope is from the point it is defined to the point it is redefined. This has one important ramification — you must declare all manifest constants with the `="` directive before you use that constant. Of course, once you redefine a symbolic constant, the previous value of that constant is forgotten. Note that you cannot redefine symbols you declare with the `textequ` or `equ` directives.
8.6 Processor Directives

By default, MASM will only assemble instructions that are available on all members of the 80x86 family. In particular, this means it will not assemble instructions that are not available on the 8086 and 8088 microprocessors. By generating an error for non-8086 instructions, MASM prevents the accidental use of instructions that are not available on various processors. This is great unless, of course, you actually want to use those instructions available on processors beyond the 8086 and 8088. The processor directives let you enable the assembly of instructions available on later processors.

The processor directives are

.8086 .8087 .186 .286 .287
.286P .386 .387 .386P .486
.486P .586 .586P

None of these directives accept any operands.

The processor directives enable all instructions available on a given processor. Since the 80x86 family is upwards compatible, specifying a particular processor directive enables all instructions on that processor and all earlier processors as well.

The .8087, .287, and .387 directives activate the floating point instruction set for the given floating point coprocessors. However, the .8086 directive also enables the 8087 instruction set; likewise, .286 enables the 80287 instruction set and .386 enables the 80387 floating point instruction set. About the only purpose for these FPU (floating point unit) directives is to allow 80287 instructions with the 8086 or 80186 instruction set or 80387 instruction with the 8086, 80186, or 80286 instruction set.

The processor directives ending with a “P” allow assembly of privileged mode instructions. Privileged mode instructions are only useful to those writing operating systems, certain device drivers, and other advanced system routines. Since this text does not discuss privileged mode instructions, there is little need to discuss these privileged mode directives further.

The 80386 and later processors support two types of segments when operating in protected mode – 16 bit segments and 32 bit segments. In real mode, these processors support only 16 bit segments. The assembler must generate subtly different opcodes for 16 and 32 bit segments. If you’ve specified a 32 bit processor using .386, .486, or .586, MASM generates instructions for 32 bit segments by default. If you attempt to run such code in real mode under MS-DOS, you will probably crash the system. There are two solutions to this problem. The first is to specify use16 as an operand to each segment you create in your program. The other solution is slightly more practical, simply put the following statement after the 32 bit processor directive:

```
option segment:use16
```

This directive tells MASM to generate 16 bit segments by default, rather than 32 bit segments.

Note that MASM does not require an 80486 or Pentium processor if you specify the .486 or .586 directives. The assembler itself is written in 80386 code2 so you only need an 80386 processor to assemble any program with MASM. Of course, if you use 80486 or Pentium processor specific instructions, you will need an 80486 or Pentium processor to run the assembled code.

You can selectively enable or disable various instruction sets throughout your program. For example, you can turn on 80386 instructions for several lines of code and then return back to 8086 only instructions. The following code sequence demonstrates this:

---

It is possible to write a routine that detects, at run-time, what processor a program is actually running on. Therefore, you can detect an 80386 processor and use 80386 instructions. If you do not detect an 80386 processor, you can stick with 8086 instructions. By selectively turning 80386 instructions on in those sections of your program that executes if an 80386 processor is present, you can take advantage of the additional instructions. Likewise, by turning off the 80386 instruction set in other sections of your program, you can prevent the inadvertent use of 80386 instructions in the 8086-only portion of the program.

8.7 Procedures

Unlike HLLs, MASM doesn’t enforce strict rules on exactly what constitutes a procedure\(^4\). 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 field\(^5\). 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 first call instruction above because it knows that `NProc` is a near procedure. It also generates a five-byte (far) call instruction for the second call because `FProc` is a far procedure. Within the proce-

---

4. “Procedure” in this text means any program unit such as procedure, subroutine, subprogram, function, operator, etc.

5. Actually, there are many other possible operands but we will not consider them in this text.
dures 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, OFFFFH
    ret
Proc2 endp

If you call Proc1, control will flow 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. For more details, see “How to Give a Symbol a Particular Type” on page 385.

8.8 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. In fact, the SHELL.ASM file (containing the skeletal assembly language program) only defines four segments and you will generally only use three of them. If you use the SHELL.ASM file as the basis for your programs, you will rarely need to worry about segmentation on the 80x86. On the other hand, if you want to write complex 80x86 programs, you will need to understand segmentation.

A segment in your file should take the following form:

```
segmentname segment {READONLY} {align} {combine} {use} {'class'}
    statements
    segmentname ends
```

The following sections describe each of the operands to the `segment` directive.

Note: segmentation is a concept that many beginning assembly language programmers find difficult to understand. Note that you do not have to completely understand segmentation to begin writing 80x86 assembly language programs. If you make a copy of the SHELL.ASM file for each program you write, you can effectively ignore segmentation issues. The main purpose of the SHELL.ASM file is to take care of the segmentation details for you. As long as you don’t write extremely large programs or use a vast amount of data, you should be able to use SHELL.ASM and forget about segmentation. Nonetheless, eventually you may want to write larger assembly language programs, or you may want to write assembly language subroutines for a high level language like Pascal or C++. At that point you will need to know quite a bit about segmentation. The bottom line is this, you can get by without having to learn about segmentation right now, but sooner or later you will need to understand it if you intend to continue writing 80x86 assembly language code.

### 8.8.1 Segment Names

The segment directive requires a label in the label field. This label is the segment’s name. MASM uses segment names for three purposes: to combine segments, to determine if a `segment override prefix` is necessary, and to obtain the address of a segment. You must also specify the segment’s name in the label field of the `ends` directive that ends the segment.

If the segment name is not unique (i.e., you’ve defined it somewhere else in the program), the other uses must also be segment definitions. If there is another segment with this same name, then the assembler treats this segment definition as a continuation of the previous segment using the same name. Each segment has its own location counter value associated with it. When you begin a new segment (that is, one whose name has not yet appeared in the source file) MASM creates a new location counter variable, initially zero, for the segment. If MASM encounters a segment definition that is a continuation of a previous segment, then MASM uses the value of the location counter at the end of that previous segment. E.g.,

---

6. MASM 5.0 and later also provide simplified segment directives. In MASM 5.0 they actually were simplified. Since then Microsoft has enhanced them over and over again. Today they are quite complex beasts. They are useful for simplifying the interface between assembly and HLLs. However, we will ignore those directives.
The first segment (CSEG) starts with a location counter value of zero. The mov ax,bx instruction is two bytes long and the ret instruction is one byte long, so the location counter is three at the end of the segment. DSEG is another three byte segment, so the location counter associated with DSEG also contains three at the end of the segment. The third segment has the same name as the first segment (CSEG), therefore the assembler will assume that they are the same segment with the second occurrence simply being an extension of the first. Therefore, code placed in the second CSEG segment will be assembled starting at offset three within CSEG – effectively continuing the code in the first CSEG segment.

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 SHELL.ASM file, 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 prefix all your variables with the segment name in this fashion. However, most programmers 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.

### 8.8.2 Segment Loading Order

Segments normally load into memory in the order that they appear in your source file. In the example above, DOS would load the CSEG segment into memory before the DSEG
segment. Even though the CSEG segment appears in two parts, both before and after DSEG. CSEG’s declaration before any occurrence of DSEG tells DOS to load the entire CSEG segment into memory before DSEG. To load DSEG before CSEG, you could use the following program:

```assembly
DSEG segment public
DSEG ends
CSEG segment public
mov ax, bx
ret
CSEG ends
DSEG segment public
Item1 byte 0
Item2 word 0
DSEG ends
CSEG segment public
mov ax, 10
add ax, Item1
ret
CSEG ends
end
```

The empty segment declaration for DSEG doesn’t emit any code. The location counter value for DSEG is zero at the end of the segment definition. Hence it’s zero at the beginning of the next DSEG segment, exactly as it was in the previous version of this program. However, since the DSEG declaration appears first in the program, DOS will load it into memory first.

The order of appearance is only one of the factors controlling the loading order. For example, if you use the “.alpha” directive, MASM will organize the segments alphabetically rather than in order of first appearance. The optional operands to the segment directive also control segment loading order. These operands are the subject of the next section.

### 8.8.3 Segment Operands

The segment directive allows six different items in the operand field: an align operand, a combine operand, a class operand, a readonly operand, a “uses” operand, and a size operand. Three of these operands control how DOS loads the segment into memory, the other three control code generation.

#### 8.8.3.1 The ALIGN Type

The align parameter is one of the following words: byte, word, dword, para, or page. These keywords instruct the assembler, linker, and DOS to load the segment on a byte, word, double word, paragraph, or page boundary. The align parameter is optional. If one of the above keywords does not appear as a parameter to the segment directive, the default alignment is paragraph (a paragraph is a multiple of 16 bytes).

Aligning a segment on a byte boundary loads the segment into memory starting at the first available byte after the last segment. Aligning on a word boundary will start the segment at the first byte with an even address after the last segment. Aligning on a dword boundary will locate the current segment at the first address that is an even multiple of four after the last segment.

For example, if segment #1 is declared first in your source file and segment #2 immediate follows and is byte aligned, the segments will be stored in memory as follows (see Figure 8.1).
If segments one and two are declared as below, and segment #2 is word aligned, the segments appear in memory as shown in Figure 8.2.

```assembly
seg1    segment
  .     
  .
seg1    ends
seg2    segment    byte
  .     
  .
seg2    ends
```

Another example: if segments one and two are as below, and segment #2 is double word aligned, the segments will be stored in memory as shown in Figure 8.3.

```assembly
seg1    segment
  .     
  .
seg1    ends
seg2    segment    dword
  .     
  .
seg2    ends
```
Since the 80x86's segment registers always point at paragraph addresses, most segments are aligned on a 16 byte paragraph (para) boundary. For the most part, your segments should always be aligned on a paragraph boundary unless you have a good reason to choose otherwise.

For example, if segments one and two are declared as below, and segment #2 is paragraph aligned, DOS will store the segments in memory as shown in Figure 8.4.

```assembly
seg1    segment
  .
  .
seg1    ends

seg2    segment    para
  .
  .
seg2    ends
```

Page boundary alignment forces the segment to begin at the next address that is an even multiple of 256 bytes. Certain data buffers may require alignment on 256 (or 512) byte boundaries. The page alignment option can be useful in this situation.

For example, if segments one and two are declared as below, and segment #2 is page aligned, the segments will be stored in memory as shown in Figure 8.5.

```assembly
seg1    segment
  .
  .
seg1    ends

seg2    segment    page
  .
  .
seg2    ends
```
If you choose any alignment other than byte, the assembler, linker, and DOS may insert several dummy bytes between the two segments, so that the segment is properly aligned. Since the 80x86 segment registers must always point at a paragraph address (that is, they must be paragraph aligned), you might wonder how the processor can address a segment that is aligned on a byte, word, or double word boundary. It's easy. Whenever you specify a segment alignment which forces the segment to begin at an address that is not a paragraph boundary, the assembler/linker will assume that the segment register points at the previous paragraph address and the location counter will begin at some offset into that segment other than zero. For example, suppose that segment #1 above ends at physical address 10F87h and segment #2 is byte aligned. The code for segment #2 will begin at segment address 10F80h. However, this will overlap segment #1 by eight bytes. To overcome this problem, the location counter for segment #2 will begin at 8, so the segment will be loaded into memory just beyond segment #1.

If segment #2 is byte aligned and segment #1 doesn't end at an even paragraph address, MASM adjusts the starting location counter for segment #2 so that it can use the previous paragraph address to access it (see Figure 8.6).

Since the 80x86 requires all segments to start on a paragraph boundary in memory, the Microsoft Assembler (by default) assumes that you want paragraph alignment for your segments. The following segment definition is always aligned on a paragraph boundary:

```
CSEG segment
       mov    ax, bx
       ret
CSEG ends
end
```
8.8.3.2 The COMBINE Type

The combine type controls the order that segments with the same name are written out to the object code file produced by the assembler. To specify the combine type you use one of the keywords public, stack, common, memory, or at. Memory is a synonym for public provided for compatibility reasons; you should always use public rather than memory. Common and at are advanced combine types that won't be considered in this text. The stack combine type should be used with your stack segments (see “The SHELL.ASM File” on page 170 for an example). The public combine type should be used with most everything else.

The public and stack combine types essentially perform the same operation. They concatenate segments with the same name into a single contiguous segment, just as described earlier. The difference between the two is the way that DOS handles the initialization of the stack segment and stack pointer registers. All programs should have at least one stack type segment (or the linker will generate a warning); the rest should all be public. MS-DOS will automatically point the stack segment register at the segment you declare with the stack combine type when it loads the program into memory.

If you do not specify a combine type, then the assembler will not concatenate the segments when producing the object code file. In effect, the absence of any combine type keyword produces a private combine type by default. Unless the class types are the same (see the next section), each segment will be emitted as MASM encounters it in the source file. For example, consider the following program:

```
CSEG segment public
    mov ax, 0
    mov VAR1, ax
CSEG ends

DSEG segment public
    I word ?
DSEG ends

CSEG segment public
    mov bx, ax
    ret
CSEG ends

DSEG segment public
    J word ?
DSEG ends
end
```

This program section will produce the same code as:

```
CSEG segment public
    mov ax, 0
    mov VAR1, ax
    mov bx, ax
    ret
CSEG ends

DSEG segment public
    I word ?
    J word ?
DSEG ends
end
```

The assembler automatically joins all segments that have the same name and are public. The reason the assembler allows you to separate the segments like this is for convenience. Suppose you have several procedures, each of which requires certain variables. You could declare all the variables in one segment somewhere, but this is often distracting. Most people like to declare their variables right before the procedure that uses them. By using the public combine type with the segment declaration, you may declare your variables right before using them and the assembler will automatically move those variable declarations into the proper segment when assembling the program. For example,
Note that you can nest segments any way you please. Unfortunately, Microsoft’s Macro Assembler scoping rules do not work the same way as a HLL like Pascal. Normally, once you define a symbol within your program, it is visible everywhere else in the program.

### 8.8.4 The CLASS Type

The final operand to the segment directive is usually the class type. The class type specifies the ordering of segments that do not have the same segment name. This operand consists of a symbol enclosed by apostrophes (quotation marks are not allowed here). Generally, you should use the following names: CODE (for segments containing program code); DATA (for segments containing variables, constant data, and tables); CONST (for segments containing constant data and tables); and STACK (for a stack segment). The following program section illustrates their use:

```
CSEG    segment  public
        mov ax, bx
        ret
        ends

CSEG    segment  public 'CODE'
        mov ax, 10
        add AX, Item1
        ret
        ends

DSEG    segment  public 'DATA'
Item1   byte  0
Item2   word  0
        ends

DSEG    segment  public 'DATA'
Item1   byte  0
Item2   word  0
        ends

SSEG    segment  stack 'STACK'
STK     word  4000 dup (?)
        ends

C2SEG   segment  public 'CODE'
        ret
        ends
```

7. The major exception are statement labels within a procedure.
The actual loading procedure is accomplished as follows. The assembler locates the first segment in the file. Since it's a public combined segment, MASM concatenates all other CSEG segments to the end of this segment. Finally, since its combine class is 'CODE', MASM appends all segments (C2SEG) with the same class afterwards. After processing these segments, MASM scans the source file for the next uncombined segment and repeats the process. In the example above, the segments will be loaded in the following order: CSEG, CSEG (2nd occurrence), C2SEG, DSEG, and then SSEG. The general rule concerning how your files will be loaded into memory is the following:

- (1) The assembler combines all public segments that have the same name.
- (2) Once combined, the segments are output to the object code file in the order of their appearance in the source file. If a segment name appears twice within a source file (and it's public), then the combined segment will be output to the object code file at the position denoted by the first occurrence of the segment within the source file.
- (3) The linker reads the object code file produced by the assembler and rearranges the segments when creating the executable file. The linker begins by writing the first segment found in the object code file to the .EXE file. Then it searches throughout the object code file for every segment with the same class name. Such segments are sequentially written to the .EXE file.
- (4) Once all the segments with the same class name as the first segment are emitted to the .EXE file, the linker scans the object code file for the next segment which doesn't belong to the same class as the previous segment(s). It writes this segment to the .EXE file and repeats step (3) for each segment belonging to this class.
- (5) Finally, the linker repeats step (4) until it has linked all the segments in the object code file.

8.8.5 The Read-only Operand

If readonly is the first operand of the segment directive, the assembler will generate an error if it encounters any instruction that attempts to write to this segment. This is most useful for code segments, though is it possible to imagine a read-only data segment. This option does not actually prevent you from writing to this segment at run-time. It is very easy to trick the assembler and write to this segment anyway. However, by specifying readonly you can catch some common programming errors you would otherwise miss. Since you will rarely place writable variables in your code segments, it's probably a good idea to make your code segments readonly.

Example of READONLY operand:

```
seg1 segment readonly para public 'DATA'
  .
  .
seg1 ends
```

8.8.6 The USE16, USE32, and FLAT Options

When working with an 80386 or later processor, MASM generates different code for 16 versus 32 bit segments. When writing code to execute in real mode under DOS, you must always use 16 bit segments. Thirty-two bit segments are only applicable to programs running in protected mode. Unfortunately, MASM often defaults to 32 bit mode whenever you select an 80386 or later processor using a directive like .386, .486, or .586 in your program. If you want to use 32 bit instructions, you will have to explicitly tell MASM to use 16 bit segments. The use16, use32, and flat operands to the segment directive let you specify the segment size.
For most DOS programs, you will always want to use the use16 operand. This tells MASM that the segment is a 16 bit segment and it assembles the code accordingly. If you use one of the directives to activate the 80386 or later instruction sets, you should put use16 in all your code segments or MASM will generate bad code.

Example of use16 operand:
```
seg1 segment para public use16 'data'
  ...
seg1 ends
```

The use32 and flat operands tell MASM to generate code for a 32 bit segment. Since this text does not deal with protected mode programming, we will not consider these options. See the MASM Programmer’s Guide for more details.

If you want to force use16 as the default in a program that allows 80386 or later instructions, there is one way to accomplish this. Place the following directive in your program before any segments:
```
.option    segment:use16
```

### 8.8.7 Typical Segment Definitions

Has the discussion above left you totally confused? Don’t worry about it. Until you’re writing extremely large programs, you needn’t concern yourself with all the operands associated with the segment directive. For most programs, the following three segments should prove sufficient:
```
DSEG    segment    para public 'DATA'
  ; Insert your variable definitions here
DSEG    ends
CSEG    segment    para public use16 'CODE'
  ; Insert your program instructions here
CSEG    ends
SSEG    segment    para stack 'STACK'
stk     word  1000h dup (?)
EndStk  equ     this word
SSEG    ends
end
```

The SHELL.ASM file automatically declares these three segments for you. If you always make a copy of the SHELL.ASM file when writing a new assembly language program, you probably won’t need to worry about segment declarations and segmentation in general.

### 8.8.8 Why You Would Want to Control the Loading Order

Certain DOS calls require that you pass the length of your program as a parameter. Unfortunately, computing the length of a program containing several segments is a very difficult process. However, when DOS loads your program into memory, it will load the entire program into a contiguous block of RAM. Therefore, to compute the length of your program, you need only know the starting and ending addresses of your program. By simply taking the difference of these two values, you can compute the length of your program.

In a program that contains multiple segments, you will need to know which segment was loaded first and which was loaded last in order to compute the length of your program. As it turns out, DOS always loads the program segment prefix, or PSP, into mem-
ory just before the first segment of your program. You must consider the length of the PSP when computing the length of your program. MS-DOS passes the segment address of the PSP in the ds register. So computing the difference of the last byte in your program and the PSP will produce the length of your program. The following code segment computes the length of a program in paragraphs:

```
CSEG segment public 'CODE'
mov ax, ds ;Get PSP segment address
sub ax, seg LASTSEG ;Compute difference

; AX now contains the length of this program (in paragraphs)
.
.
.
CSEG ends

; Insert ALL your other segments here.
LASTSEG segment para public 'LASTSEG'
LASTSEG ends
end
```

### 8.8.9 Segment Prefixes

When the 80x86 references a memory operand, it usually references a location within the current data segment. 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 prefix instructs the 80x86 to fetch its memory operand from the specified segment rather than the default segment. For example, `mov ax, cs:[bx]` loads the accumulator from address `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.

### 8.8.10 Controlling Segments with the ASSUME Directive

The 80x86 generally references data items relative to the ds segment register (or stack segment). Likewise, all code references (jumps, calls, etc.) are always relative to the current code segment. There is only one catch – how does the assembler know which segment is the data segment and which is the code segment (or other segment)? The segment directive doesn’t tell you what type of segment it happens to be in the program. Remember, a data segment is a data segment because the ds register points at it. Since the ds register can be changed at run time (using an instruction like `mov ds,ax`), any segment can be a data segment. This has some interesting consequences for the assembler. When you specify a segment in your program, not only must you tell the CPU that a segment is a data segment, but you must also tell the assembler where and when that segment is a data (or code/stack/extra/F/G) segment. The assume directive provides this information to the assembler.

---

8. The exception, of course, are those instructions and addressing modes that use the stack segment by default (e.g., push/pop and addressing modes that use bp or sp).
9. By loading DS with the address of that segment.
The assume directive takes the following form:

\begin{verbatim}
assume {CS:seg} {DS:seg} {ES:seg} {FS:seg} {GS:seg} {SS:seg}
\end{verbatim}

The braces surround optional items, you do not type the braces as part of these operands. Note that there must be at least one operand. Seg is either the name of a segment (defined with the segment directive) or the reserved word nothing. Multiple operands in the operand field of the assume directive must be separated by commas. Examples of valid assume directives:

\begin{verbatim}
assume DS:DSEG
assume CS:CSEG, DS:DSEG, ES:DSEG, SS:SSEG
assume CS:CSEG, DS:NOTHING
\end{verbatim}

The assume directive tells the assembler that you have loaded the specified segment register(s) with the segment addresses of the specified value. Note that this directive does not modify any of the segment registers, it simply tells the assembler to assume the segment registers are pointing at certain segments in the program. Like the processor selection and equate directives, the assume directive modifies the assembler’s behavior from the point MASM encounters it until another assume directive changes the stated assumption.

Consider the following program:

```
DSEG1 segment para public 'DATA'
    var1 word ?
DSEG1 ends

DSEG2 segment para public 'DATA'
    var2 word ?
DSEG2 ends

CSEG segment para public 'CODE'
    assume CS:CSEG, DS:DSEG1, ES:DSEG2
    mov ax, seg DSEG1
    mov ds, ax
    mov ax, seg DSEG2
    mov es, ax
    mov var1, 0
    mov var2, 0
    .
    assume DS:DSEG2
    mov ax, seg DSEG2
    mov ds, ax
    mov var2, 0
    .
CSEG ends
end
```

Whenever the assembler encounters a symbolic name, it checks to see which segment contains that symbol. In the program above, var1 appears in the DSEG1 segment and var2 appears in the DSEG2 segment. Remember, the 80x86 microprocessor doesn’t know about segments declared within your program, it can only access data in segments pointed at by the cs, ds, es, ss, fs, and gs segment registers. The assume statement in this program tells the assembler the ds register points at DSEG1 for the first part of the program and at DSEG2 for the second part of the program.

When the assembler encounters an instruction of the form mov var1, 0, the first thing it does is determine var1’s segment. It then compares this segment against the list of assumptions the assembler makes for the segment registers. If you didn’t declare var1 in one of these segments, then the assembler generates an error claiming that the program cannot access that variable. If the symbol (var1 in our example) appears in one of the currently assumed segments, then the assembler checks to see if it is the data segment. If so, then the instruction is assembled as described in the appendices. If the symbol appears in
a segment other than the one that the assembler assumes \texttt{ds} points at, then the assembler emits a segment override prefix byte, specifying the actual segment that contains the data.

In the example program above, MASM would assemble \texttt{mov VAR1,0} without a segment prefix byte. MASM would assemble the first occurrence of the \texttt{mov VAR2,0} instruction with an \texttt{es:} segment prefix byte since the assembler assumes \texttt{es}, rather than \texttt{ds}, is pointing at segment \texttt{DSEG2}. MASM would assemble the second occurrence of this instruction without the \texttt{es:} segment prefix byte since the assembler, at that point in the source file, assumes that \texttt{ds} points at \texttt{DSEG2}. Keep in mind that it is very easy to confuse the assembler. Consider the following code:

```assembly
CSEG segment para public 'CODE'
assume CS:CSEG, DS:DSEG1, ES:DSEG2
mov ax, seg DSEG1
mov ds, ax
jmp SkipFixDS
FixDS: mov ax, seg DSEG2
mov ds, ax
SkipFixDS:
CSEG ends
end
```

Notice that this program jumps around the code that loads the \texttt{ds} register with the segment value for \texttt{DSEG2}. This means that at label \texttt{SkipFixDS} the \texttt{ds} register contains a pointer to \texttt{DSEG1}, not \texttt{DSEG2}. However, the assembler isn’t bright enough to realize this problem, so it blindly assumes that \texttt{ds} points at \texttt{DSEG2} rather than \texttt{DSEG1}. This is a disaster waiting to happen. Because the assembler assumes you’re accessing variables in \texttt{DSEG2} while the \texttt{ds} register actually points at \texttt{DSEG1}, such accesses will reference memory locations in \texttt{DSEG1} at the same offset as the variables accessed in \texttt{DSEG2}. This will scramble the data in \texttt{DSEG1} (or cause your program to read incorrect values for the variables assumed to be in segment \texttt{DSEG2}).

For beginning programmers, the best solution to the problem is to avoid using multiple (data) segments within your programs as much as possible. Save the multiple segment accesses for the day when you’re prepared to deal with problems like this. As a beginning assembly language programmer, simply use one code segment, one data segment, and one stack segment and leave the segment registers pointing at each of these segments while your program is executing. The \texttt{assume} directive is quite complex and can get you into a considerable amount of trouble if you misuse it. Better not to bother with fancy uses of \texttt{assume} until you are quite comfortable with the whole idea of assembly language programming and segmentation on the 80x86.

The \texttt{nothing} reserved word tells the assembler that you haven’t the slightest idea where a segment register is pointing. It also tells the assembler that you’re not going to access any data relative to that segment register unless you explicitly provide a segment prefix to an address. A common programming convention is to place \texttt{assume} directives before all procedures in a program. Since segment pointers to declared segments in a program rarely change except at procedure entry and exit, this is the ideal place to put \texttt{assume} directives:
assume ds:P1Dseg, cs:cseg, es:nothing
Procedure1 proc near
push ds ;Preserve DS
push ax ;Preserve AX
mov ax, P1Dseg ;Get pointer to P1Dseg into the
ds register.
mov ds, ax ; ds register.
.
.
pop ax ;Restore ax’s value.
pop ds ; Restore ds’ value.
ret
Procedure1 endp
The only problem with this code is that MASM still assumes that ds points at P1Dseg when it encounters code after Procedure1. The best solution is to put a second assume directive after the endp directive to tell MASM it doesn’t know anything about the value in the ds register:
.
.
ret
Procedure1 endp
assume ds:nothing
Although the next statement in the program will probably be yet another assume directive giving the assembler some new assumptions about ds (at the beginning of the procedure that follows the one above), it’s still a good idea to adopt this convention. If you fail to put an assume directive before the next procedure in your source file, the assume ds:nothing statement above will keep the assembler from assuming you can access variables in P1Dseg.

Segment override prefixes always override any assumptions made by the assembler. mov ax, cs:var1 always loads the ax register with the word at offset var1 within the current code segment, regardless of where you’ve defined var1. The main purpose behind the segment override prefixes is handling indirect references. If you have an instruction of the form mov ax,[bx] the assembler assumes that bx points into the data segment. If you really need to access data in a different segment you can use a segment override, thusly, mov ax, es:[bx].

In general, if you are going to use multiple data segments within your program, you should use full segment:offset names for your variables. E.g., mov ax, DSEG1:I and mov bx, DSEG2:J. This does not eliminate the need to load the segment registers or make proper use of the assume directive, but it will make your program easier to read and help MASM locate possible errors in your program.

The assume directive is actually quite useful for other things besides just setting the default segment. You’ll see some more uses for this directive a little later in this chapter.

8.8.11 Combining Segments: The GROUP Directive

Most segments in a typical assembly language program are less than 64 Kilobytes long. Indeed, most segments are much smaller than 64 Kilobytes in length. When MS-DOS loads the program’s segments into memory, several of the segments may fall into a single 64K region of memory. In practice, you could combine these segments into a single segment in memory. This might possibly improve the efficiency of your code if it saves having to reload segment registers during program execution.

So why not simply combine such segments in your assembly language code? Well, as the next section points out, maintaining separate segments can help you structure your programs better and help make them more modular. This modularity is very important in your programs as they get more complex. As usual, improving the structure and modularity of your programs may cause them to become less efficient. Fortunately, MASM provides a directive, group, that lets you treat two segments as the same physical segment without abandoning the structure and modularity of your program.
The group directive lets you create a new segment name that encompasses the segments it groups together. For example, if you have two segments named "Module1Data" and "Module2Data" that you wish to combine into a single physical segment, you could use the group directive as follows:

```
ModuleData group Module1Data, Module2Data
```

The only restriction is that the end of the second module’s data must be no more than 64 kilobytes away from the start of the first module in memory. MASM and the linker will not automatically combine these segments and place them together in memory. If there are other segments between these two in memory, then the total of all such segments must be less than 64K in length. To reduce this problem, you can use the class operand to the segment directive to tell the linker to combine the two segments in memory by using the same class name:

```
ModuleData group Module1Data, Module2Data
Module1Data segment para public 'MODULES'
  ...
Module1Data ends
Module2Data segment byte public 'MODULES'
  ...
Module2Data ends
```

With declarations like those above, you can use "ModuleData" anywhere MASM allows a segment name, as the operand to a mov instruction, as an operand to the assume directive, etc. The following example demonstrates the usage of the ModuleData segment name:

```
assume ds:ModuleData
Module1Proc proc near
  push ds ;Preserve ds' value.
push ax ;Preserve ax's value.
mov ax, ModuleData ;Load ds with the segment address
  mov ds, ax ; of ModuleData.
  ...
  pop ax ;Restore ax's and ds' values.
pop ds
ret
Module1Proc endp
```

Of course, using the group directive in this manner hasn’t really improved the code. Indeed, by using a different name for the data segment, one could argue that using group in this manner has actually obfuscated the code. However, suppose you had a code sequence that needed to access variables in both the Module1Data and Module2Data segments. If these segments were physically and logically separate you would have to load two segment registers with the addresses of these two segments in order to access their data concurrently. This would cost you a segment override prefix on all the instructions that access one of the segments. If you cannot spare an extra segment register, the situation will be even worse, you’ll have to constantly load new values into a single segment register as you access data in the two segments. You can avoid this overhead by combining the two logical segments into a single physical segment and accessing them through their group rather than individual segment names.

If you group two or more segments together, all you’re really doing is creating a pseudo-segment that encompasses the segments appearing in the group directive’s operand field. Grouping segments does not prevent you from accessing the individual segments in the grouping list. The following code is perfectly legal:
assume ds:Module1Data
mov   ax, Module1Data
mov   ds, ax
.
< Code that accesses data in Module1Data >
.
assume ds:Module2Data
mov   ax, Module2Data
mov   ds, ax
.
< Code that accesses data in Module2Data >
.
assume ds:ModuleData
mov   ax, ModuleData
mov   ds, ax
.
< Code that accesses data in both Module1Data and Module2Data >
.
.
When the assembler processes segments, it usually starts the location counter value for a given segment at zero. Once you group a set of segments, however, an ambiguity arises; grouping two segments causes MASM and the linker to concatenate the variables of one or more segments to the end of the first segment in the group list. They accomplish this by adjusting the offsets of all symbols in the concatenated segments as though they were all symbols in the same segment. The ambiguity exists because MASM allows you to reference a symbol in its segment or in the group segment. The symbol has a different offset depending on the choice of segment. To resolve the ambiguity, MASM uses the following algorithm:

- If MASM doesn’t know that a segment register is pointing at the symbol’s segment or a group containing that segment, MASM generates an error.
- If an assume directive associates the segment name with a segment register but does not associate a segment register with the group name, then MASM uses the offset of the symbol within its segment.
- If an assume directive associates the group name with a segment register but does not associate a segment register with the symbol’s segment name, MASM uses the offset of the symbol with the group.
- If an assume directive provides segment register association with both the symbol’s segment and its group, MASM will pick the offset that would not require a segment override prefix. For example, if the assume directive specifies that ds points at the group name and es points at the segment name, MASM will use the group offset if the default segment register would be ds since this would not require MASM to emit a segment override prefix opcode. If either choice results in the emission of a segment override prefix opcode, MASM will choose the offset (and segment override prefix) associated with the symbol’s segment.

MASM uses the algorithm above if you specify a variable name without a segment prefix. If you specify a segment register override prefix, then MASM may choose an arbitrary offset. Often, this turns out to be the group offset. So the following instruction sequence, without an assume directive telling MASM that the BadOffset symbol is in seg1 may produce bad object code:

```
DataSegs group Data1, Data2, Data3
.
Data2 segment
.
.
BadOffset word ?
.
.
Data2 ends
.
```

When the assembler processes a group of segments, it usually starts the location counter value for a given segment at zero. Once you group a set of segments, however, an ambiguity arises; grouping two segments causes MASM and the linker to concatenate the variables of one or more segments to the end of the first segment in the group list. They accomplish this by adjusting the offsets of all symbols in the concatenated segments as though they were all symbols in the same segment. The ambiguity exists because MASM allows you to reference a symbol in its segment or in the group segment. The symbol has a different offset depending on the choice of segment. To resolve the ambiguity, MASM uses the following algorithm:

- If MASM doesn’t know that a segment register is pointing at the symbol’s segment or a group containing that segment, MASM generates an error.
- If an `assume` directive associates the segment name with a segment register but does not associate a segment register with the group name, then MASM uses the offset of the symbol within its segment.
- If an `assume` directive associates the group name with a segment register but does not associate a segment register with the symbol’s segment name, MASM uses the offset of the symbol with the group.
- If an `assume` directive provides segment register association with both the symbol’s segment and its group, MASM will pick the offset that would not require a segment override prefix. For example, if the `assume` directive specifies that `ds` points at the group name and `es` points at the segment name, MASM will use the group offset if the default segment register would be `ds` since this would not require MASM to emit a segment override prefix opcode. If either choice results in the emission of a segment override prefix opcode, MASM will choose the offset (and segment override prefix) associated with the symbol’s segment.

MASM uses the algorithm above if you specify a variable name without a segment prefix. If you specify a segment register override prefix, then MASM may choose an arbitrary offset. Often, this turns out to be the group offset. So the following instruction sequence, without an `assume` directive telling MASM that the `BadOffset` symbol is in `seg1` may produce bad object code:
assume ds:nothing, es:nothing, fs:nothing, gs:nothing

Despite

mov ax, Data2 ; Force ds to point at data2

mov ds, ax ; the assume directive above.

mov ax, ds:BadOffset ; May use the offset from DataSegs

DataSegs

; rather than Data2!

If you want to force the correct offset, use the variable name containing the complete segment:offset address form:

; To force the use of the offset within the DataSegs group use an instruction like the following:

mov ax, DataSegs:BadOffset

; To force the use of the offset within Data2, use:

mov ax, Data2:BadOffset

You must use extra care when working with groups within your assembly language programs. If you force MASM to use an offset within some particular segment (or group) and the segment register is not pointing at that particular segment or group, MASM may not generate an error message and the program will not execute correctly. Reading the offsets MASM prints in the assembly listing will not help you find this error. MASM always displays the offsets within the symbol’s segment in the assembly listing. The only way to really detect that MASM and the linker are using bad offsets is to get into a debugger like CodeView and look at the actual machine code bytes produced by the linker and loader.

8.8.12 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. To be perfectly frank, if you use the SHELL.ASM file as a skeleton for the assembly language programs you write, you can get by quite easily without ever worrying about segments, groups, segment override prefixes, and full segment:offset names. 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.

### 8.9 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 the SHELL.ASM file as a skeleton for your assembly language programs, you will notice that the end directive already specifies the procedure main as the starting point for the program.

If you are using separate assembly and you’re linking together several different object code files (see “Managing Large Programs” on page 425), 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.

### 8.10 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). See the SHELL.ASM program in Chapter Four for more details on these segments.
Directives and Pseudo Opcodes

Since Chapter Five covers the declaration of variables, data types, structures, arrays, and pointers in depth, this chapter will not waste any more time discussing this subject. Refer to Chapter Five for more details.

8.11 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 appearing in the source file and will generate a warning or an error message if you attempt to use that symbol in a context that doesn’t allow its particular type. Although unusual in an assembler, most high level languages apply certain typing rules to symbols declared in the source 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 a procedure label. Intel, in designing the syntax for 8086 assembly language, decided that all the reasons for using a strongly typed language apply to assembly language as well as Pascal. Therefore, standard Intel syntax 80x86 assemblers, like MASM, impose certain type restrictions on the use of symbols within your assembly language programs.

8.11.1 How to Give a Symbol a Particular Type

Symbols, in an 80x86 assembly language program, may be one of eight different primitive types: byte, word, dword, qword, tbyte, near, far, and abs (constant). Anytime you define a label with the byte, word, dword, qword, or tbyte pseudo-opcodes, MASM associates the type of that pseudo-opcode with the label. For example, the following variable declaration will create a symbol of type byte:

```
BVar byte ?
```

Likewise, the following defines a dword symbol:

```
DWord dword ?
```

Variable types are not limited to the primitive types built into MASM. If you create your own types using the typedef or struct directives MASM will associate those types with any associated variable declarations.

You can define near symbols (also known as statement labels) in a couple of different ways. First, all procedure symbols declared with the proc directive (with either a blank operand field or near in the operand field) are near symbols. Statement labels are also near symbols. A statement label takes the following form:

```
label: instr
```

Instr represents an 80x86 instruction. Note that a colon must follow the symbol. It is not part of the symbol, the colon informs the assembler that this symbol is a statement label and should be treated as a near typed symbol.

Statement labels are often the targets of jump and loop instructions. For example, consider the following code sequence:

```
mov cx, 25
Loop1: mov ax, cx
call PrintInteger
loop Loop1
```

10. MASM also supports an FWORD type. FWORD is for programmers working in 32-bit protected mode. This text will not consider that type.
11. Note: if you are using the simplified directives, a blank operand field might not necessarily imply that the procedure is near. If your program does not contain a “.MODEL” directive, however, blank operand fields imply a near type.
12. The mnemonic “instr” is optional. You may also place a statement label on a line by itself. The assembler assigns the location counter of the next instruction in the program to the symbol.
The loop instruction decrements the cx register and transfers control to the instruction labelled by Loop1 until cx becomes zero.

Inside a procedure, statement labels are local. That is, the scope of statement labels inside a procedure are visible only to code inside that procedure. If you want to make a symbol global to a procedure, place two colons after the symbol name. In the example above, if you needed to refer to Loop1 outside of the enclosing procedure, you would use the code:

```
mov cx, 25
Loop1::
mov ax, cx
call PrintInteger
loop Loop1
```

Generally, far symbols are the targets of jump and call instructions. The most common method programmers use to create a far label is to place far in the operand field of a proc directive. Symbols that are simply constants are normally defined with the equ directive. You can also declare symbols with different types using the equ and extrn/extern/externdef directives. An explanation of the extrn directives appears in the section “Managing Large Programs” on page 425.

If you declare a numeric constant using an equate, MASM assigns the type abs (absolute, or constant) to the system. Text and string equates are given the type text. You can also assign an arbitrary type to a symbol using the equ directive, see “Type Operators” on page 392 for more details.

### 8.11.2 Label Values

Whenever you define a label using a directive or pseudo-opcode, MASM gives it a type and a value. The value MASM gives the label is usually the current location counter value. If you define the symbol with an equate the equate’s operand usually specifies the symbol’s value. When encountering the label in an operand field, as with the loop instruction above, MASM substitutes the label’s value for the label.

### 8.11.3 Type Conflicts

Since the 80x86 supports strongly typed symbols, the next question to ask is “What are they used for?” In a nutshell, strongly typed symbols can help verify proper operation of your assembly language programs. Consider the following code sections:

```
DSEG segment public 'DATA'
.
.
I byte ?
.
.
DSEG ends
CSEG segment public 'CODE'
.
.
mov ax, I
.
.
CSEG ends
end
```

The mov instruction in this example is attempting to load the ax register (16 bits) from a byte sized variable. Now the 80x86 microprocessor is perfectly capable of this operation. It would load the al register from the memory location associated with I and load the ah register from the next successive memory location (which is probably the L.O. byte of some other variable). However, this probably wasn’t the original intent. The person who
wrote this code probably forgot that I is a byte sized variable and assumed that it was a word variable – which is definitely an error in the logic of the program.

MASM would never allow an instruction like the one above to be assembled without generating a diagnostic message. This can help you find errors in your programs, particularly difficult-to-find errors. On occasion, advanced assembly language programmers may want to execute a statement like the one above. MASM provides certain coercion operators that bypass MASM’s safety mechanisms and allow illegal operations (see “Coercion” on page 390).

8.12 Address Expressions

An address expression is an algebraic expression that produces a numeric result that MASM merges into the displacement field 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 specified operand. For example, the following instruction fills the immediate data fields of the mov instruction with zeros:

\[
\text{mov ax, 0}
\]

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:

\[
\begin{align*}
\text{Value} & \text{ equ 0} \\
\text{mov ax, Value}
\end{align*}
\]

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 \textit{ax} from location \textit{Var} and add one to it:

\[
\text{mov ax, Var1+1}
\]

Instead, this instruction loads the \textit{al} register with the byte stored at the address of \textit{Var1} plus one and then loads the \textit{ah} register with the byte stored at the address of \textit{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!

8.12.1 Symbol Types and Addressing Modes

Consider the following instruction:

\[
\text{jmp Location}
\]

Depending on how the label \textit{Location} is defined, this jmp instruction will perform one of several different operations. If you’ll look back at the chapter on the 80x86 instruction set, you’ll notice that the jmp instruction takes several forms. As a recap, they are

\[
\begin{align*}
\text{jmp label} & \quad \text{(short)} \\
\text{jmp label} & \quad \text{(near)} \\
\text{jmp label} & \quad \text{(far)} \\
\text{jmp reg} & \quad \text{(indirect near, through register)} \\
\text{jmp mem/reg} & \quad \text{(indirect near, through memory)} \\
\text{jmp mem/reg} & \quad \text{(indirect far, through memory)}
\end{align*}
\]

Notice that MASM uses the same mnemonic (\textit{jmp}) for each of these instructions; how does it tell them apart? The secret lies with the operand. If the operand is a statement label within the current segment, the assembler selects one of the first two forms depend-
ing on the distance to the target instruction. If the operand is a statement label within a different segment, then the assembler selects jmp (far) label. If the operand following the jmp instruction is a register, then MASM uses the indirect near jmp and the program jumps to the address in the register. If a memory location is selected, the assembler uses one of the following jumps:

- NEAR if the variable was declared with word/sword/dw
- FAR if the variable was declared with dword/sdword/dd

An error results if you’ve used byte/sbyte/db, qword/dq, or tbyte/dt or some other type.

If you’ve specified an indirect address, e.g., jmp [bx], the assembler will generate an error because it cannot determine if bx is pointing at a word or a dword variable. For details on how you specify the size, see the section on coercion in this chapter.

### 8.12.2 Arithmetic and Logical Operators

MASM recognizes several arithmetic and logical operators. The following tables provide a list of such operators:

#### Table 36: Arithmetic Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+expr</td>
<td>Positive (unary)</td>
</tr>
<tr>
<td>-</td>
<td>-expr</td>
<td>Negation (unary)</td>
</tr>
<tr>
<td>+</td>
<td>expr + expr</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>expr - expr</td>
<td>Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>expr * expr</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>expr / expr</td>
<td>Division</td>
</tr>
<tr>
<td>MOD</td>
<td>expr MOD expr</td>
<td>Modulo (remainder)</td>
</tr>
<tr>
<td>[ ]</td>
<td>expr [ expr ]</td>
<td>Addition (index operator)</td>
</tr>
</tbody>
</table>

#### Table 37: Logical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHR</td>
<td>expr SHR expr</td>
<td>Shift right</td>
</tr>
<tr>
<td>SHL</td>
<td>expr SHL expr</td>
<td>Shift left</td>
</tr>
<tr>
<td>NOT</td>
<td>NOT expr</td>
<td>Logical (bit by bit) NOT</td>
</tr>
<tr>
<td>AND</td>
<td>expr AND expr</td>
<td>Logical AND</td>
</tr>
<tr>
<td>OR</td>
<td>expr OR expr</td>
<td>Logical OR</td>
</tr>
<tr>
<td>XOR</td>
<td>expr XOR expr</td>
<td>Logical XOR</td>
</tr>
</tbody>
</table>

#### Table 38: Relational Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>expr EQ expr</td>
<td>True (0FFh) if equal, false (0) otherwise</td>
</tr>
<tr>
<td>NE</td>
<td>expr NE expr</td>
<td>True (0FFh) if not equal, false (0) otherwise</td>
</tr>
<tr>
<td>LT</td>
<td>expr LT expr</td>
<td>True (0FFh) if less, false (0) otherwise</td>
</tr>
<tr>
<td>LE</td>
<td>expr LE expr</td>
<td>True (0FFh) if less or equal, false (0) otherwise</td>
</tr>
<tr>
<td>GT</td>
<td>expr GT expr</td>
<td>True (0FFh) if greater, false (0) otherwise</td>
</tr>
<tr>
<td>GE</td>
<td>expr GE expr</td>
<td>True (0FFh) if greater or equal, false (0) otherwise</td>
</tr>
</tbody>
</table>
You must not confuse these operators with 80x86 instructions! The addition operator adds two values together, their sum becomes an operand to an instruction. This addition is performed when assembling the program, not at run time. If you need to perform an addition at execution time, use the `add` or `adc` instructions.

You're probably wondering “What are these operators used for?” The truth is, not much. The addition operator gets used quite a bit, the subtraction somewhat, the comparisons once in a while, and the rest even less. Since addition and subtraction are the only operators beginning assembly language programmers regularly employ, this discussion considers only those two operators and brings up the others as required throughout this text.

The addition operator takes two forms: `expr+expr` or `expr[expr]`. For example, the following instruction loads the accumulator, not from memory location `COUNT`, but from the very next location in memory:

```
mov al, COUNT+1
```

The assembler, upon encountering this statement, will compute the sum of `COUNT`’s address plus one. The resulting value is the memory address for this instruction. As you may recall, the `mov al, memory` instruction is three bytes long and takes the form:

```
Opcode | L. O. Displacement Byte | H. O. Displacement Byte
```

The two displacement bytes of this instruction contain the sum `COUNT+1`.

The `expr[expr]` form of the addition operation is for accessing elements of arrays. If `AryData` is a symbol that represents the address of the first element of an array, `AryData[5]` represents the address of the fifth byte into `AryData`. The expression `AryData+5` produces the same result, and either could be used interchangeably, however, for arrays the `expr[expr]` form is a little more self documenting. One trap to avoid: `expr1[expr2][expr3]` does not automatically index (properly) into a two dimensional array for you. This simply computes the sum `expr1+expr2+expr3`.

The subtraction operator works just like the addition operator, except it computes the difference rather than the sum. This operator will become very important when we deal with local variables in Chapter 11.

Take care when using multiple symbols in an address expression. MASM restricts the operations you can perform on symbols to addition and subtraction and only allows the following forms:

<table>
<thead>
<tr>
<th>Expression:</th>
<th>Resulting type:</th>
</tr>
</thead>
<tbody>
<tr>
<td>reloc + const</td>
<td>Reloc, at address specified.</td>
</tr>
<tr>
<td>reloc - const</td>
<td>Reloc, at address specified.</td>
</tr>
<tr>
<td>reloc - reloc</td>
<td>Constant whose value is the number of bytes between the first and second operands. Both variables must physically appear in the same segment in the current source file.</td>
</tr>
</tbody>
</table>

Reloc stands for relocatable symbol or expression. This can be a variable name, a statement label, a procedure name, or any other symbol associated with a memory location in the program. It could also be an expression that produces a relocatable result. MASM does not allow any operations other than addition and subtraction on expressions whose resulting type is relocatable. You cannot, for example, compute the product of two relocatable symbols.

The first two forms above are very common in assembly language programs. Such an address expression will often consist of a single relocatable symbol and a single constant (e.g., “var + 1”). You won’t use the third form very often, but it is very useful once in a while. You can use this form of an address expression to compute the distance, in bytes, between two points in your program. The `procsize` symbol in the following code, for example, computes the size of Proc1:
Proc1 proc near
push ax
push bx
push cx
mov cx, 10
lea bx, SomeArray
mov ax, 0
ClrArray: mov [bx], ax
add bx, 2
loop ClrArray
pop cx
pop bx
pop ax
ret
Proc1 endp

procsize = $ - Proc1

“$” is a special symbol MASM uses to denote the current offset within the segment (i.e.,
the location counter). It is a relocatable symbol, as is Proc1, so the equate above computes
the difference between the offset at the start of Proc1 and the end of Proc1. This is the
length of the Proc1 procedure, in bytes.

The operands to the operators other than addition and subtraction must be constants
or an expression yielding a constant (e.g., “$-Proc1” above produces a constant value).
You’ll mainly use these operators in macros and with the conditional assembly directives.

8.12.3 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:

\[
\text{type PTR expression}
\]

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:

\[
\text{mov ax, word ptr I}
\]
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:

\[
\begin{align*}
\text{mov} & \quad \text{word ptr DBL, bx} \\
\text{mov} & \quad \text{word ptr DBL+2, ds}
\end{align*}
\]

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 fine. 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.\(^{13}\)

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:

\[
\text{mov} \quad [bx], \ 5
\]

Unfortunately, the assembler has no way of telling whether bx points at a byte, word, or double word item in memory.\(^{14}\) The value of the immediate operand isn’t of any use. Even though five 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:

\[
\begin{align*}
\text{mov} & \quad \text{byte ptr [bx], 5} \quad ;\text{For a byte variable} \\
\text{mov} & \quad \text{word ptr [bx], 5} \quad ;\text{For a word variable} \\
\text{mov} & \quad \text{dword ptr [bx], 5} \quad ;\text{For a dword variable}
\end{align*}
\]

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? Well, quit complaining and remember the textequ directive. With the equate directive you can substitute a long string like “word ptr” for a short symbol. You’ll find equates like the following in many programs, including several in this text:

\[
\begin{align*}
\text{byp} & \quad \text{textequ <byte ptr>} \quad ;\text{Remember, “bp” is a reserved symbol!} \\
\text{wp} & \quad \text{textequ <word ptr>} \\
\text{dp} & \quad \text{textequ <dword ptr>} \\
\text{np} & \quad \text{textequ <near ptr>} \\
\text{fp} & \quad \text{textequ <far ptr>}
\end{align*}
\]

With equates like the above, you can use statements like the following:

---

\(^{13}\) The situation when you force a near call to a far procedure is even worse. See the exercises for more details.

\(^{14}\) Actually, you can use the assume directive to tell MASM what bx is pointing at. See the MASM reference manuals for details.
8.12.4 Type Operators

The “xxxx ptr” coercion operator is an example of a type operator. MASM expressions possess two major attributes: a value and a type. The arithmetic, logical, and relational operators change an expression’s value. The type operators change its type. The previous section demonstrated how the ptr operator could change an expression’s type. There are several additional type operators as well.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTR</td>
<td>byte ptr expr</td>
<td>Coerce expr to point at a byte.</td>
</tr>
<tr>
<td></td>
<td>word ptr expr</td>
<td>Coerce expr to point at a word.</td>
</tr>
<tr>
<td></td>
<td>dword ptr expr</td>
<td>Coerce expr to point at a dword.</td>
</tr>
<tr>
<td></td>
<td>qword ptr expr</td>
<td>Coerce expr to point at a qword.</td>
</tr>
<tr>
<td></td>
<td>tbyte ptr expr</td>
<td>Coerce expr to point at a byte.</td>
</tr>
<tr>
<td></td>
<td>near ptr expr</td>
<td>Coerce expr to a near value.</td>
</tr>
<tr>
<td></td>
<td>far ptr expr</td>
<td>Coerce expr to a far value.</td>
</tr>
<tr>
<td>short</td>
<td>short expr</td>
<td>expr must be within ±128 bytes of the current jmp instruction (typically a JMP instruction). This operator forces the JMP instruction to be two bytes long (if possible).</td>
</tr>
<tr>
<td>this</td>
<td>this type</td>
<td>Returns an expression of the specified type whose value is the current location counter.</td>
</tr>
<tr>
<td>seg</td>
<td>seg label</td>
<td>Returns the segment address portion of label.</td>
</tr>
<tr>
<td>offset</td>
<td>offset label</td>
<td>Returns the offset address portion of label.</td>
</tr>
<tr>
<td>.type</td>
<td>type label</td>
<td>Returns a byte that indicates whether this symbol is a variable, statement label, or structure name. Superseded by opattr.</td>
</tr>
<tr>
<td>opattr</td>
<td>opattr label</td>
<td>Returns a 16 bit value that gives information about label.</td>
</tr>
<tr>
<td>length</td>
<td>length variable</td>
<td>Returns the number of array elements for a single dimension array. If a multi-dimension array, this operator returns the number of elements for the first dimension.</td>
</tr>
<tr>
<td>lengthof</td>
<td>lengthof variable</td>
<td>Returns the number of items in array variable.</td>
</tr>
<tr>
<td>type</td>
<td>type symbol</td>
<td>Returns an expression whose type is the same as symbol and whose value is the size, in bytes, for the specified symbol.</td>
</tr>
<tr>
<td>size</td>
<td>size variable</td>
<td>Returns the number of bytes allocated for single dimension array variable. Useless for multi-dimension arrays. Superseded by sizeof.</td>
</tr>
<tr>
<td>sizeof</td>
<td>sizeof variable</td>
<td>Returns the size, in bytes, of array variable.</td>
</tr>
<tr>
<td>low</td>
<td>low expr</td>
<td>Returns the L.O. byte of expr.</td>
</tr>
<tr>
<td>lowword</td>
<td>lowword expr</td>
<td>Returns the L.O. word of expr.</td>
</tr>
<tr>
<td>high</td>
<td>high expr</td>
<td>Returns the H.O. byte of expr.</td>
</tr>
<tr>
<td>highword</td>
<td>highword expr</td>
<td>Returns the H.O. word of expr.</td>
</tr>
</tbody>
</table>
The short operator works exclusively with the jmp instruction. Remember, there are two jmp direct near instructions, one that has a range of 128 bytes around the jmp, one that has a range of 32,768 bytes around the current instruction. MASM will automatically generate a short jump if the target address is up to 128 bytes before the current instruction. This operator is mainly present for compatibility with old MASM (pre-6.0) code.

The this operator forms an expression with the specified type whose value is the current location counter. The instruction mov bx, this word, for example, will load the bx register with the value 8B1Eh, the opcode for mov bx, memory. The address this word is the address of the opcode for this very instruction! You mostly use this operator with the equ directive to give a symbol some type other than constant. For example, consider the following statement:

```
HERE equ this near
```

This statement assigns the current location counter value to HERE and sets the type of HERE to near. This, of course, could have been done much easier by simply placing the label HERE: on the line by itself. However, the this operator with the equ directive does have some useful applications, consider the following:

```
WArray equ this word
BArray byte 200 dup (?)
```

In this example the symbol BArray is of type byte. Therefore, instructions accessing BArray must contain byte operands throughout. MASM would flag a mov ax, BArray+8 instruction as an error. However, using the symbol WArray lets you access the same exact memory locations (since WArray has the value of the location counter immediately before encountering the byte pseudo-opcode) so mov ax, WArray+8 accesses location BArray+8. Note that the following two instructions are identical:

```
mov ax, word ptr BArray+8
mov ax, WArray+8
```

The seg operator does two things. First, it extracts the segment portion of the specified address, second, it converts the type of the specified expression from address to constant. An instruction of the form mov ax, seg symbol always loads the accumulator with the constant corresponding to the segment portion of the address of symbol. If the symbol is the name of a segment, MASM will automatically substitute the paragraph address of the segment for the name. However, it is perfectly legal to use the seg operator as well. The following two statements are identical if dseg is the name of a segment:

```
mov ax, dseg
mov ax, seg dseg
```

Offset works like seg, except it returns the offset portion of the specified expression rather than the segment portion. If VAR1 is a word variable, mov ax, VAR1 will always load the two bytes at the address specified by VAR1 into the ax register. The mov ax, offset VAR1 instruction, on the other hand, loads the offset (address) of VAR1 into the ax register. Note that you can use the lea instruction or the mov instruction with the offset operator to load the address of a scalar variable into a 16 bit register. The following two instructions both load bx with the address of variable J:

```
mov bx, offset J
lea bx, J
```

The lea instruction is more flexible since you can specify any memory addressing mode, the offset operator only allows a single symbol (i.e., displacement only addressing). Most programmers use the mov form for scalar variables and the lea instruction for other addressing modes. This is because the mov instruction was faster on earlier processors.

One very common use for the seg and offset operators is to initialize a segment and pointer register with the segmented address of some object. For example, to load es:di with the address of SomeVar, you could use the following code:

```
mov di, seg SomeVar
mov es, di
mov di, offset SomeVar
```
Since you cannot load a constant directly into a segment register, the code above copies the segment portion of the address into \texttt{di} and then copies \texttt{di} into \texttt{es} before copying the offset into \texttt{di}. This code uses the \texttt{di} register to copy the segment portion of the address into \texttt{es} so that it will affect as few other registers as possible.

\texttt{Opattr} returns a 16 bit value providing specific information about the expression that follows it. The \texttt{.type} operator is an older version of \texttt{opattr} that returns the L.O. eight bits of this value. Each bit in the value of these operators has the following meaning:

\begin{table}[h]
\centering
\begin{tabular}{|c|l|}
\hline
Bit(s) & Meaning                                                                 \\
\hline
0     & References a label in the code segment if set.                           \\
1     & References a memory variable or relocatable data object if set.          \\
2     & Is an immediate (absolute/constant) value if set.                        \\
3     & Uses direct memory addressing if set.                                    \\
4     & Is a register name, if set.                                             \\
5     & References no undefined symbols and there is no error, if set.          \\
6     & Is an SS: relative reference, if set.                                   \\
7     & References an external name.                                            \\
8-10  & 000 - no language type                                                   \\
     & 001 - C/C++ language type                                               \\
     & 010 - SYSCALL language type                                             \\
     & 011 - STDCALL language type                                             \\
     & 100 - Pascal language type                                              \\
     & 101 - FORTRAN language type                                             \\
     & 110 - BASIC language type                                               \\
\hline
\end{tabular}
\caption{OPATTR/.TYPE Return Value}
\end{table}

The language bits are for programmers writing code that interfaces with high level languages like C++ or Pascal. Such programs use the simplified segment directives and MASM’s HLL features.

You would normally use these values with MASM’s conditional assembly directives and macros. This allows you to generate different instruction sequences depending on the type of a macro parameter or the current assembly configuration. For more details, see “Conditional Assembly” on page 397 and “Macros” on page 400.

The \texttt{size}, \texttt{sizeof}, \texttt{length}, and \texttt{lengthof} operators compute the sizes of variables (including arrays) and return that size and their value. You shouldn’t normally use \texttt{size} and \texttt{length}. The \texttt{sizeof} and \texttt{lengthof} operators have superceded these operators. \texttt{Size} and \texttt{length} do not always return reasonable values for arbitrary operands. MASM 6.x includes them to remain compatible with older versions of the assembler. However, you will see an example later in this chapter where you can use these operators.

The \texttt{sizeof variable} operator returns the number of bytes directly allocated to the specified variable. The following examples illustrate the point:

\begin{verbatim}
a1 byte ? ; SIZEOF(a1) = 1
a2 word ? ; SIZEOF(a2) = 2
a4 dword ? ; SIZEOF(a4) = 4
a8 real18 ? ; SIZEOF(a8) = 8
ary0 byte 10 dup (0) ; SIZEOF(ary0) = 10
ary1 word 10 dup (10 dup (0)) ; SIZEOF(ary1) = 200
\end{verbatim}

You can also use the \texttt{sizeof} operator to compute the size, in bytes, of a structure or other data type. This is very useful for computing an index into an array using the formula from Chapter Four:

\[
\text{Element \_Address := base \_address + index*Element \_Size}
\]

You may obtain the element size of an array or structure using the \texttt{sizeof} operator. So if you have an array of structures, you can compute an index into the array as follows:
.286 ;Allow 80286 instructions.
s struct
<some number of fields>
s ends

array s 16 dup ({}) ;An array of 16 “s” elements

imul bx, I, sizeof s ;Compute BX := I * elementsize
mov al, array[bx].fieldname

You can also apply the sizeof operator to other data types to obtain their size in bytes. For example, sizeof byte returns 1, sizeof word returns two, and sizeof dword returns 4. Of course, applying this operator to MASM’s built-in data types is questionable since the size of those objects is fixed. However, if you create your own data types using typedef, it makes perfect sense to compute the size of the object using the sizeof operator:

integer typedef word
Array integer 16 dup (?)

imul bx, bx, sizeof integer

In the code above, sizeof integer would return two, just like sizeof word. However, if you change the typedef statement so that integer is a dword rather than a word, the sizeof integer operand would automatically change its value to four to reflect the new size of an integer.

The lengthof operator returns the total number of elements in an array. For the Array variable above, lengthof Array would return 16. If you have a two dimensional array, lengthof returns the total number of elements in that array.

When you use the lengthof and sizeof operators with arrays, you must keep in mind that it is possible for you to declare arrays in ways that MASM can misinterpret. For example, the following statements all declare arrays containing eight words:

A1 word 8 dup (?)
A2 word 1, 2, 3, 4, 5, 6, 7, 8
; Note: the "\" is a “line continuation” symbol. It tells MASM to append
; the next line to the end of the current line.
A3 word 1, 2, 3, 4, \ 5, 6, 7, 8
A4 word 1, 2, 3, 4
word 5, 6, 7, 8

Applying the sizeof and lengthof operators to A1, A2, and A3 produces sixteen (sizeof) and eight (lengthof). However, sizeof(A4) produces eight and lengthof(A4) produces four. This happens because MASM thinks that the arrays begin and end with a single data declaration. Although the A4 declaration sets aside eight consecutive words, just like the other three declarations above, MASM thinks that the two word directives declare two separate arrays rather than a single array. So if you want to initialize the elements of a large array or a multidimensional array and you also want to be able to apply the lengthof and sizeof operators to that array, you should use A3’s form of declaration rather than A4’s.

The type operator returns a constant that is the number of bytes of the specified operand. For example, type(word) returns the value two. This revelation, by itself, isn’t particularly interesting since the size and sizeof operators also return this value. However, when you use the type operator with the comparison operators (eq, ne, le, lt, gt, and ge), the comparison produces a true result only if the types of the operands are the same. Consider the following definitions:
Integer typedef word
J word ?
K sword ?
L integer ?
M word ?

byte type (J) eq word ;value = 0FFh
byte type (J) eq sword ;value = 0
byte type (J) eq type (L) ;value = 0FFh
byte type (J) eq type (M) ;value = 0FFh
byte type (L) eq integer ;value = 0FFh
byte type (K) eq dword ;value = 0

Since the code above typedef'd Integer to word, MASM treats integers and words as the same type. Note that with the exception of the last example above, the value on either side of the eq operator is two. Therefore, when using the comparison operations with the type operator, MASM compares more than just the value. Therefore, type and sizeof are not synonymous. E.g.,

byte type (J) eq type (K) ;value = 0
byte (sizeof J) equ (sizeof K) ;value = 0FFh

The type operator is especially useful when using MASM’s conditional assembly directives. See “Conditional Assembly” on page 397 for more details.

The examples above also demonstrate another interesting MASM feature. If you use a type name within an expression, MASM treats it as though you’d entered “type(name)” where name is a symbol of the given type. In particular, specifying a type name returns the size, in bytes, of an object of that type. Consider the following examples:

Integer typedef word
s struct
d dword ?
w word ?
b byte ?
s ends

byte word ;value = 2
byte sword ;value = 2
byte byte ;value = 1
byte dword ;value = 4
byte s ;value = 7
byte word eq word ;value = 0FFh
byte word eq sword ;value = 0
byte b eq dword ;value = 0
byte s eq byte ;value = 0
byte word eq Integer ;value = 0FFh

The high and low operators, like offset and seg, change the type of expression from whatever it was to a constant. These operators also affect the value of the expression – they decompose it into a high order byte and a low order byte. The high operator extracts bits eight through fifteen of the expression, the low operator extracts and returns bits zero through seven. Highword and lowword extract the H.O. and L.O. 16 bits of an expression (see Figure 8.7).

You can extract bits 16-23 and 24-31 using expressions of the form low(highword(expr)) and high(highword(expr))\textsuperscript{15}, respectively.

8.12.5 Operator Precedence

Although you will rarely need to use a complex address expression employing more than two operands and a single operator, the need does arise on occasion. MASM supports a simple operator precedence convention based on the following rules:

- MASM executes operators of a higher precedence first.

\textsuperscript{15} The parentheses make this expression more readable, they are not required.
Directives and Pseudo Opcodes

• Operators of an equal precedence are left associative and evaluate from left to right.
• Parentheses override the normal precedence.

Table 41: Operator Precedence

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(Highest)</strong></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>length, lengthof, size, sizeof, ( ), [ ] ,&lt; &gt;</td>
</tr>
<tr>
<td>2</td>
<td>. (structure field name operator)</td>
</tr>
<tr>
<td>3</td>
<td>CS: DS: ES: FS: GS: SS: (Segment override prefixes)</td>
</tr>
<tr>
<td>4</td>
<td>ptr offset set type opattr this</td>
</tr>
<tr>
<td>5</td>
<td>high, low, highword, lowword</td>
</tr>
<tr>
<td>6</td>
<td>+ - (unary)</td>
</tr>
<tr>
<td>7</td>
<td>* / mod shl shr</td>
</tr>
<tr>
<td>8</td>
<td>+ - (binary)</td>
</tr>
<tr>
<td>9</td>
<td>eq ne lt le gt ge</td>
</tr>
<tr>
<td>10</td>
<td>not</td>
</tr>
<tr>
<td>11</td>
<td>and</td>
</tr>
<tr>
<td>12</td>
<td>or xor</td>
</tr>
<tr>
<td>13</td>
<td>short .type</td>
</tr>
<tr>
<td><strong>(Lowest)</strong></td>
<td></td>
</tr>
</tbody>
</table>

Parentheses should only surround expressions. Some operators, like sizeof and lengthof, require type names, not expressions. They do not allow you to put parentheses around the name. Therefore, “(sizeof X)” is legal, but “sizeof(X)” is not. Keep this in mind when using parentheses to override operator precedence in an expression. If MASM generates an error, you may need to rearrange the parentheses in your expression.

As is true for expressions in a high level language, it is a good idea to always use parentheses to explicitly state the precedence in all complex address expressions (complex meaning that the expression has more than one operator). This generally makes the expression more readable and helps avoid precedence related bugs.

8.13 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 – 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 8086 version if they have an 8086 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, see “Macros” on page 400.

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.

8.13.1 IF Directive

The # directive uses the following syntax:

```
if expression
<sequence of statements>
else ;This is optional!
<sequence of statements>
endif
```

Page 398
MASM evaluates expression. If it is a non-zero value, then MASM will assemble the statements between the if and else directives (or endif, if the else isn’t present). If the expression evaluates to zero (false) and an else section is present, MASM will assemble the statements between the else directive and the endif directive. If the else section is not present and expression evaluates to false, then MASM will not assemble any of the code between the if and endif directives.

The important thing to remember is that expression has to be an expression that MASM can evaluate at assembly time. That is, it must evaluate to a constant. Manifest constants (equates) and values that MASM’s type operators produce are commonly found in if directive expressions. For example, suppose you want to assemble code for two different processors as described above. You could use statements like the following:

```assembly
Processor = 80386 ; Set to 8086 for 8086-only code
...
if Processor eq 80386
  shl ax, 4 ; Must be 80386 processor.
else
  mov cl, 4
  shl cl, 4
endif
```

There are other ways to accomplish this same thing. MASM provides built-in variables that tell you if you are assembling code for some specific processor. More on that later.

---

### 8.13.2 IFE directive

The ife directive is used exactly like the if directive, except it assembles the code after the ife directive only if the expression evaluates to zero (false), rather than true (non-zero).

---

### 8.13.3 IFDEF and IFNDEF

These two directives require a single symbol as the operand. Ifdef will assemble the associated code if the symbol is defined, Ifndef will assemble the associated code if the symbol isn’t defined. Use else and endif to terminate the conditional assembly sequences.

These directives are especially popular for including or not including code in an assembly language program to handle certain special cases. For example, you could use statements like the following to include debugging statements in your code:

```assembly
ifdef DEBUG
  ; Place debugging statements here
endif
```

To activate the debugging code, simply define the symbol DEBUG somewhere at the beginning of your program (before the first ifdef referencing DEBUG). To automatically eliminate the debugging code, simply delete the definition of DEBUG. You may define DEBUG using a simple statement like:

```assembly
DEBUG = 0
```

Note that the value you assign to DEBUG is unimportant. Only the fact that you have defined (or have not defined) this symbol is important.

---

### 8.13.4 IFB, IFNB

These directives, useful mainly in macros (see “Macros” on page 400) check to see if an operand is blank (ifb) or not blank (ifnb). Consider the following code:
Blank textequ <>
NotBlank textequ <not blank>
ifb Blank
<this code will assemble>
endif
ifb NotBlank
<this code will not>
endif

The ifnb works in an opposite manner to ifb. That is, it would assemble the statements above that ifb does not and vice versa.

8.13.5 IFIDN, IFDIF, IFIDNI, and IFDIFI

These conditional assembly directives take two operands and process the associated code if the operands are identical (ifidn), different (ifdif), identical ignoring case (ifidni), or different ignoring case (ifdifi). The syntax is

ifidn op1, op2
<statements to assemble if <op1> = <op2>>
endif
ifdif op1, op2
<statements to assemble if <op1> ≠ <op2>>
endif
ifidni op1, op2
<statements to assemble if <op1> = <op2>>
endif
ifdifi op1, op2
<statements to assemble if <op1> ≠ <op2>>
endif

The difference between the IFxxx and IFxxxI statements above is that the IFxxxI statements ignore differences in alphabetic case when comparing operands.

8.14 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 final “.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 file like high level language procedures and loops let you repetitively execute sequences of high level language statements.

8.14.1 Procedural Macros

The following sequence defines a macro:

name macro [parameter1 {parameter2 {,...}}}]
<statements>
endm
Name must be a valid and unique symbol in the source file. You will use this identifier to invoke the macro. The (optional) parameter names are placeholders for values you specify when you invoke the macro; the braces above denote the optional items, they should not actually appear in your source code. These parameter names are local to the macro and may appear elsewhere in the program.

Example of a macro definition:

```
COPY     macro Dest, Source
    mov    ax, Source
    mov    Dest, ax
endm
```

This macro will copy the word at the source address to the word at the destination address. The symbols Dest and Source are local to the macro and may appear elsewhere in the program.

Note that MASM does not immediately assemble the instructions between the macro and endm directives when MASM encounters the macro. Instead, the assembler stores the text corresponding to the macro into a special table (called the symbol table). MASM inserts these instructions into your program when you invoke the macro.

To invoke (use) a macro, simply specify the macro name as a MASM mnemonic. When you do this, MASM will insert the statements between the macro and endm directives into your code at the point of the macro invocation. If your macro has parameters, MASM will substitute the actual parameters appearing as operands for the formal parameters appearing in the macro definition. MASM does a straight textual substitution, just as though you had created text equates for the parameters.

Consider the following code that uses the COPY macro defined above:

```
call  SetUpX
copy  Y, X
add   Y, 5
```

This program segment will issue a call to SetUpX (which, presumably, does something to the variable X) then invokes the COPY macro, that copies the value in the variable X into the variable Y. Finally, it adds five to the value contained in variable Y.

Note that this instruction sequence is absolutely identical to:

```
call  SetUpX
mov   ax, X
mov   Y, ax
add   Y, 5
```

In some instances using macros can save a considerable amount of typing in your programs. For example, suppose you want to access elements of various two dimensional arrays. As you may recall, the formula to compute the row-major address for an array element is

```
element address = base address + (First Index * Row Size + Second Index) * element size
```

Suppose you want write some assembly code that achieves the same result as the following C code:

```
int a[16][7], b[16][7], x[7][16];
int i,j;

for (i=0; i<16; i = i + 1)
    for (j=0; j < 7; j = j + 1)
        x[j][i] = a[i][j]*b[15-i][j];
```

The 80x86 code for this sequence is rather complex because of the number of array accesses. The complete code is
This is a lot of code for only five C/C++ statements! If you take a close look at this code, you'll notice that a large number of the statements simply compute the index into the three arrays. Furthermore, the code sequences that compute these array indices are very similar. If they were exactly the same, it would be obvious we could write a macro to replace the three array index computations. Since these index computations are not identical, one might wonder if it is possible to create a macro that will simplify this code. The answer is yes; by using macro parameters it is very easy to write such a macro. Consider the following code:

```
NDX2 macro Index1, Index2, RowSize
    imul bx, Index1, RowSize
    add bx, Index2
    add bx, bx
endm

mov I, 0 ;Initialize I loop index with zero.
ForILp:
    cmp I, 16 ;Is I less than 16?
    jnl ForIDone ;If so, fall into body of I loop.

mov J, 0 ;Initialize J loop index with zero.
ForJLp:
    cmp J, 7 ;Is J less than 7?
    jnl ForJDone ;If so, fall into body of J loop.
    imul bx, I, 7 ;Compute index for a[i][j].
    add bx, J
    add bx, bx ;Element size is two bytes.
    mov ax, A[bx] ;Get a[i][j]

    mov bx, 15 ;Compute index for b[15-I][j].
    sub bx, I
    imul bx, I
    add bx, J
    add bx, bx ;Element size is two bytes.
    imul ax, b[bx] ;Compute a[i][j] * b[16-i][j]

    imul bx, J, 16 ;Compute index for X[J][I]
    add bx, I
    add bx, bx
    mov X[bx], ax ;Store away result.
    inc J ;Next loop iteration.
    jmp ForJLp

ForJDone:
    inc I ;Next I loop iteration.
    jmp ForILp

ForIDone:
    ;Done with nested loop.
```

This is a lot of code for only five C/C++ statements! If you take a close look at this code, you'll notice that a large number of the statements simply compute the index into the three arrays. Furthermore, the code sequences that compute these array indices are very similar. If they were exactly the same, it would be obvious we could write a macro to replace the three array index computations. Since these index computations are not identical, one might wonder if it is possible to create a macro that will simplify this code. The answer is yes; by using macro parameters it is very easy to write such a macro. Consider the following code:
Directives and Pseudo Opcodes

mov    bx, 15 ;Compute index for b[15-I][j].
sub    bx, I
NDX2   bx, J, 7
imul   ax, b[bx] ;Compute a[i][j] * b[15-i][j]
NDX2   J, I, 16
mov    X[bx], ax ;Store away result.
inc    J ;Next loop iteration.
jmp    ForJLp
ForJDone: inc    I ;Next I loop iteration.
jmp    ForILp
ForIDone: ;Done with nested loop.

One problem with the NDX2 macro is that you need to know the row size of an array (since it is a macro parameter). In a short example like this one, that isn’t much of a problem. However, if you write a large program you can easily forget the sizes and have to look them up or, worse yet, “remember” them incorrectly and introduce a bug into your program. One reasonable question to ask is if MASM could figure out the row size of the array automatically. The answer is yes.

MASM’s length operator is a holdover from the pre-6.0 days. It was supposed to return the number of elements in an array. However, all it really returns is the first value appearing in the array’s operand field. For example, (length a) would return 16 given the definition for a above. MASM corrected this problem by introducing the lengthof operator that properly returns the total number of elements in an array. (Lengthof a), for example, properly returns 112 (16 * 7). Although the (length a) operator returns the wrong value for our purposes (it returns the column size rather than the row size), we can use its return value to compute the row size using the expression (lengthof a)/(length a). With this knowledge, consider the following two macros:

; LDAX- This macro loads ax with the word at address Array[Index1][Index2]  
; Assumptions: You’ve declared the array using a statement like  
; Array word Colsize dup (RowSize dup (?))  
; and the array is stored in row major order.  
; If you specify the (optional) fourth parameter, it is an 80x86  
; machine instruction to substitute for the MOV instruction that  
; loads AX from Array[ bx ].
LDAX    macro Array, Index1, Index2, Instr
    imul   bx, Index1, (lengthof Array) / (length Array)  
    add    bx, Index2  
    add    bx, bx  
    ; See if the caller has supplied the fourth operand.
    ifb    <Instr>
    mov    ax, Array[ bx ]  
    endif
    else
    instr   ax, Array[ bx ]  
    endif
   endif
endm

; STAX- This macro stores ax into the word at address Array[Index1][Index2]  
; Assumptions: Same as above
STAX    macro Array, Index1, Index2
    imul   bx, Index1, (lengthof Array) / (length Array)  
    add    bx, Index2  
    add    bx, bx  
    mov    Array[ bx ], ax  
endm

With the macros above, the original program becomes:
As you can plainly see, the code for the loops above is getting shorter and shorter by using these macros. Of course, the entire code sequence is actually longer because the macros represent more lines of code that they save in the original program. However, that is an artifact of this particular program. In general, you’d probably have more than three array accesses; furthermore, you can always put the LDAX and STAX macros in a library file and automatically include them anytime you’re dealing with two dimensional arrays. Although, technically, your program might actually contain more assembly language statements if you include these macros in your code, you only had to write those macros once. After that, it takes very little effort to include the macros in any new program.

We can shorten this code sequence even more using some additional macros. However, there are a few additional topics to cover before we can do that, so keep reading.

### 8.14.2 Macros vs. 80x86 Procedures

Beginning assembly language programmers often confuse macros and procedures. A procedure is a single section of code that you call from various points in the program. A macro is a sequence of instructions that MASM replicates in your program each time you use the macro. Consider the following two code fragments:

```assembly
; Proc_1 procedure
Proc_1 proc near
    mov ax, 0
    mov bx, ax
    mov cx, 5
    ret
Proc_1 endp

; Macro_1 macro
Macro_1 macro
    mov ax, 0
    mov bx, ax
    mov cx, 5
endm

call Proc_1
call Proc_1
...
call Macro_1
...
call Macro_1
```

```assembly
; Proc_1 procedure
Proc_1 proc near
    mov ax, 0
    mov bx, ax
    mov cx, 5
    ret
Proc_1 endp
```
Although the macro and procedure produce the same result, they do it in different ways. The procedure definition generates code when the assembler encounters the proc directive. A call to this procedure requires only three bytes. At execution time, the 80x86:

- encounters the call instruction,
- pushes the return address onto the stack,
- jumps to Proc_1,
- executes the code therein,
- pops the return address off the stack, and
- returns to the calling code.

The macro, on the other hand, does not emit any code when processing the statements between the macro and endm directives. However, upon encountering Macro_1 in the mnemonic field, MASM will assemble every statement between the macro and endm directives and emit that code to the output file. At run time, the CPU executes these instructions without the call/ret overhead.

The execution of a macro expansion is usually faster than the execution of the same code implemented with a procedure. However, this is another example of the classic speed/space trade-off. Macros execute faster by eliminating the call/return sequence. However, the assembler copies the macro code into your program at each macro invocation. If you have a lot of macro invocations within your program, it will be much larger than the same program that uses procedures.

Macro invocations and procedure invocations are considerably different. To invoke a macro, you simply specify the macro name as though it were an instruction or directive. To invoke a procedure you need to use the call instruction. In many contexts it is unfortunate that you use two separate invocation mechanisms for such similar operations. The real problem occurs if you want to switch a macro to a procedure or vice versa. It might be that you’ve been using macro expansion for a particular operation, but now you’ve expanded the macro so many times it makes more sense to use a procedure. Maybe just the opposite is true, you’ve been using a procedure but you want to expand the code in-line to improve it’s performance. The problem with either conversion is that you will have to find every invocaiton of the macro or procedure call and modify it. Modifying the procedure or macro is easy, but locating and changing all the invocations can be quite a bit of work. Fortunately, there is a very simple technique you can use so procedure calls share the same syntax as macro invocation. The trick is to create a macro or a text equate for each procedure you write that expands into a call to that procedure. For example, suppose you write a procedure ClearArray that zeros out arrays. When writing the code, you could do the following:

```
ClearArray    textequ  <call $$ClearArray>
$$ClearArray  proc  near
  ...
$$ClearArray  endm
```

To call the ClearArray procedure, you’d simply use a statement like the following:

```
  ...
  <Set up parameters for ClearArray>
  ClearArray
  ...
```

If you ever change the $$ClearArray procedure to a macro, all you need to do is name it ClearArray and dispose of the textequ for the procedure. Conversely, if you already have a macro and you want to convert it to a procedure, Simply name the procedure $$procname and create a text equate that emits a call to this procedure. This allows you to use the same invocation syntax for procedures or macros.

This text won’t normally use the technique described above, except for the UCR Standard Library routines. This is not because this isn’t a good way to invoke procedures. Some people have trouble differentiating macros and procedures, so this text will use...
explicit calls to help avoid that confusion. Standard Library calls are an exception because using macro invocations is the standard way to call these routines.

### 8.14.3 The LOCAL Directive

Consider the following macro definition:

```assembly
LJE macro Dest
    jne SkipIt
    jmp Dest
SkipIt:
    endm
```

This macro does a “long jump if equal”. However, there is one problem with it. Since MASM copies the macro text verbatim (allowing, of course, for parameter substitution), the symbol `SkipIt` will be redefined each time the LJE macro appears. When this happens, the assembler will generate a multiple definition error. To overcome this problem, the `local` directive can be used to define a `local` symbol within the macro. Consider the following macro definition:

```assembly
LJE macro Dest
    local SkipIt
    jne SkipIt
    jmp Dest
SkipIt:
    endm
```

In this macro definition, `SkipIt` is a local symbol. Therefore, the assembler will generate a new copy of `SkipIt` each time you invoke the macro. This will prevent MASM from generating an error.

The `local` directive, if it appears within your macro definition, must appear immediately after the `macro` directive. If you need multiple local symbols, you can specify several of them in the `local` directive’s operand field. Simply separate each symbol with a comma:

```assembly
IFDEFEQUAL macro a, b
    local ElsePortion, Done
    mov ax, a
    cmp ax, b
    jne ElsePortion
    inc bx
    jmp Done
ElsePortion: dec bx
Done:
    endm
```

### 8.14.4 The EXITM Directive

The `exitm` directive immediately terminates the expansion of a macro, exactly as though MASM encountered `endm`. MASM ignores all text from the `exitm` directive to the `endm`.

You’re probably wondering why anyone would ever use the `exitm` directive. After all, if MASM ignores all text between `exitm` and `endm`, why bother sticking an `exitm` directive into your macro in the first place? The answer is conditional assembly. Conditional assembly can be used to conditionally execute the `exitm` directive, thereby allowing further macro expansion under certain conditions, consider the following:

```assembly
Bytes macro Count
    byte Count
if Count eq 0
    exitm
endif
byte Count dup (?)
endm
```
Of course, this simple example could have been coded without using the `exitm` directive (the conditional assembly directive is all we require), but it does demonstrate how the `exitm` directive can be used within a conditional assembly sequence to control its influence.

8.14.5 Macro Parameter Expansion and Macro Operators

Since MASM does a textual substitution for macro parameters when you invoke a macro, there are times when a macro invocation might not produce the results you expect. For example, consider the following (admittedly dumb) macro definition:

```plaintext
Index = 8

Problem- This macro attempts to load AX with the element of a word
array specified by the macro’s parameter. This parameter
must be an assembly-time constant.

Problem macro Parameter
mov ax, Array[Parameter*2]
endm.

Problem Index+2
```

When MASM expands the first invocation of `Problem` above, it produces the instruction:

```plaintext
mov ax, Array[2*2]
```

Okay, so far so good. This code loads element two of `Array` into `ax`. However, consider the expansion of the second invocation to `Problem`, above:

```plaintext
mov ax, Array[Index+2*2]
```

Because MASM’s address expressions support operator precedence (see “Operator Precedence” on page 396), this macro expansion will not produce the correct result. It will access the sixth element of `Array` (at index 12) rather than the tenth element at index 20.

The problem above occurs because MASM simply replaces a formal parameter by the actual parameter’s text, not the actual parameter’s value. This pass by name parameter passing mechanism should be familiar to long-time C and C++ programmers who use the `#define` statement. If you think that macro (pass by name) parameters work just like Pascal and C’s pass by value parameters, you are setting yourself up for eventual disaster.

One possible solution, that works well for macros like the above, is to put parentheses around macro parameters that occur within expressions inside the macro. Consider the following code:

```plaintext
Problem macro Parameter
mov ax, Array[(Parameter)*2]
endm.

Problem Index+2
```

This macro invocation expands to

```plaintext
mov ax, Array[(Index+2)*2]
```

This produces the expected result.

Textual parameter substitution is but one problem you’ll run into when using macros. Another problem occurs because MASM has two types of assembly time values: numeric and text. Unfortunately, MASM expects numeric values in some contexts and text values in others. They are not fully interchangeable. Fortunately, MASM provides a set of operators that let you convert between one form and the other (if it is possible to do so). To
understand the subtle differences between these two types of values, look at the following statements:

```plaintext
Numeric = 10+2
Textual textequ <10+2>
```

MASM evaluates the numeric expression “10+2” and associates the value twelve with the symbol `Numeric`. For the symbol `Textual`, MASM simply stores away the string “10+2” and substitutes it for `Textual` anywhere you use it in an expression.

In many contexts, you could use either symbol. For example, the following two statements both load `ax` with twelve:

```assembly
mov ax, Numeric ;Same as mov ax, 12
mov ax, Textual ;Same as mov ax, 10+2
```

However, consider the following two statements:

```assembly
mov ax, Numeric*2 ;Same as mov ax, 12*2
mov ax, Textual*2 ;Same as mov ax, 10+2*2
```

As you can see, the textual substitution that occurs with text equates can lead to the same problems you encountered with textual substitution of macro parameters.

MASM will automatically convert a text object to a numeric value, if the conversion is necessary. Other than the textual substitution problem described above, you can use a text value (whose string represents a numeric quantity) anywhere MASM requires a numeric value.

Going the other direction, numeric value to text value, is not automatic. Therefore, MASM provides an operator you can use to convert numeric data to textual data: the “%” operator. This `expansion` operator forces an immediate evaluation of the following expression and then it converts the result of the expression into a string of digits. Look at these invocations of the `Problem` macro:

```plaintext
Problem 10+2 ;Parameter is “10+2”
Problem %10+2 ;Parameter is “12”
```

In the second example above, the text expansion operator instructs MASM to evaluate the expression “10+2” and convert the resulting numeric value to a text value consisting of the digits that represent the value twelve. Therefore, these two macro expand into the following statements (respectively):

```assembly
mov ax, Array[10+2*2] ;Problem 10+2 expansion
mov ax, Array[12*2] ;Problem %10+2 expansion
```

MASM provides a second operator, the `substitution` operator that lets you expand macro parameter names where MASM does not normally expect a symbol. The substitution operator is the ampersand (“&”) character. If you surround a macro parameter name with ampersands inside a macro, MASM will substitute the parameter’s text regardless of the location of the symbol. This lets you expand macro parameters whose names appear inside other identifiers or inside literal strings. The following macro demonstrates the use of this operator:

```assembly
DebugMsg macro Point, String
  Msg&String& byte "At point &Point&: &String&"
endm

DebugMsg 5, <Assertion fails>
```

The macro invocation immediately above produces the statement:

```assembly
Msg5 byte "At point 5: Assertion failed"
```

Note how the substitution operator allowed this macro to concatenate “Msg” and “5” to produce the label on the byte directive. Also note that the expansion operator lets you expand macro identifiers even if they appear in a literal string constant. Without the ampersands in the string, MASM would have emitted the statement:
Another important operator active within macros is the **literal character operator**, the exclamation mark ("!"). This symbol instructs MASM to pass the following character through without any modification. You would normally use this symbol if you need to include one of the following symbols as a character within a macro:

```
! & > %
```

For example, had you really wanted the string in the `DebugMsg` macro to display the ampersands, you would use the definition:

```asm
DebugMsg macro Point, String
Msg&String& byte "At point !&Point!&: !&String!&"
endm
```

"Debug 5, <Assertion fails>" would produce the following statement:

```asm
Msg5 byte "At point &Point&: &String&"
```

Use the "<" and ">" symbols to delimit text data inside MASM. The following two invocations of the `PutData` macro show how you can use these delimiters in a macro:

```asm
PutData macro TheName, TheData
PD_&TheName& byte TheData
endm

PutData MyData, 5, 4, 3 ;Emits "PD_MyData byte 5"
PutData MyData, <5, 4, 3> ;Emits "PD_MyData byte 5, 4, 3"
```

You can use the text delimiters to surround objects that you wish to treat as a single parameter rather than as a list of multiple parameters. In the `PutData` example above, the first invocation passes four parameters to `PutData` (PutData ignores the last two). In the second invocation, there are two parameters, the second consisting of the text 5, 4, 3.

The last macro operator of interest is the ";;" operator. This operator begins a macro comment. MASM normally copies all text from the macro into the body of the program during assembly, including all comments. However, if you begin a comment with ";;" rather than a single semicolon, MASM will not expand the comment as part of the code during macro expansion. This increases the speed of assembly by a tiny amount and, more importantly, it does not clutter a program listing with copies of the same comment (see “Controlling the Listing” on page 424 to learn about program listings).

**Table 42: Macro Operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>Text substitution operator</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Literal text operator</td>
</tr>
<tr>
<td>!</td>
<td>Literal character operator</td>
</tr>
<tr>
<td>%</td>
<td>Expression operator</td>
</tr>
<tr>
<td>;;</td>
<td>Macro comment</td>
</tr>
</tbody>
</table>

### 8.14.6 A Sample Macro to Implement For Loops

Remember the for loops and matrix operations used in a previous example? At the conclusion of that section there was a brief comment that we could “improve” that code even more using macros, but the example had to wait. With the description of macro operators out of the way, we can now finish that discussion. The macros that implement the for loop are
First, three macros that let us construct symbols by concatenating others. This is necessary because this code needs to expand several components in text equates multiple times to arrive at the proper symbol.

MakeLbl - Emits a label created by concatenating the two parameters passed to this macro.

MakeLbl macro FirstHalf, SecondHalf
&FirstHalf&&SecondHalf&:
endm

jgDone macro FirstHalf, SecondHalf
jg &FirstHalf&&SecondHalf&
endm

jmpLoop macro FirstHalf, SecondHalf
jmp &FirstHalf&&SecondHalf&
endm

ForLp - This macro appears at the beginning of the for loop. To invoke this macro, use a statement of the form:

ForLp LCV, StartVal, StopVal

Note: "FOR" is a MASM reserved word, which is why this macro doesn't use that name.

ForLp macro LCV, Start, Stop

; We need to generate a unique, global symbol for each for loop we create. This symbol needs to be global because we will need to reference it at the bottom of the loop. To generate a unique symbol, this macro concatenates "FOR" with the name of the loop control variable and a unique numeric value that this macro increments each time the user constructs a for loop with the same loop control variable.

ifndef $$For&LCV& ;; Symbol = $$FOR concatenated with LCV
$$For&LCV& = 0 ;; If this is the first loop w/LCV, use zero, otherwise increment it.
else
$$For&LCV& = $$For&LCV& + 1
endif

; Emit the instructions to initialize the loop control variable:

mov ax, Start
mov LCV, ax

; Output the label at the top of the for loop. This label takes the form
;
$$FOR LCV x
; where LCV is the name of the loop control variable and X is a unique number that this macro increments for each for loop that uses the same loop control variable.

MakeLbl $$For&LCV&, %$$For&LCV&

; Okay, output the code to see if this for loop is complete.
; The jgDone macro generates a jump (if greater) to the label the Next macro emits below the bottom of the for loop.

mov ax, LCV
cmp ax, Stop
jgDone $$Next&LCV&, $$For&LCV&
endm

; The Next macro terminates the for loop. This macro increments the loop control variable and then transfers control back to the label at the top of the for loop.

Next macro LCV
inc LCV
jmpLoop $$For&LCV&, $$For&LCV&
MakeLbl $$Next&LCV&, $$For&LCV&
endm
With these macros and the LDAX/STAX macros, the code from the array manipulation example presented earlier becomes very simple. It is

```
ForLp I, 0, 15
ForLp J, 0, 6
ldax A, I, J ;Fetch A[I][J]
mov bx, 15 ;Compute 16-I.
sub bx, I
ldax b, bx, J, imul ;Multiply in B[15-I][J].
stax x, J, I ;Store to X[J][I]
Next J
Next I
```

Although this code isn’t quite as short as the original C/C++ example, it’s getting pretty close!

While the main program became much simpler, there is a question of the macros themselves. The ForLp and Next macros are extremely complex! If you had to go through this effort every time you wanted to create a macro, assembly language programs would be ten times harder to write if you decided to use macros. Fortunately, you only have to write (and debug) a macro like this once. Then you can use it as many times as you like, in many different programs, without having to worry much about its implementation.

Given the complexity of the For and Next macros, it is probably a good idea to carefully describe what each statement in these macros is doing. However, before discussing the macros themselves, we should discuss exactly how one might implement a for/next loop in assembly language. This text fully explores the for loop a little later, but we can certainly go over the basics here. Consider the following Pascal for loop:

```
for variable := StartExpression to EndExpression do
    Some_Statement;
```

Pascal begins by computing the value of StartExpression. It then assigns this value to the loop control variable (variable). It then evaluates EndExpression and saves this value in a temporary location. Then the Pascal for statement enters the loop’s body. The first thing the loop does is compare the value of variable against the value it computed for EndExpression. If the value of variable is greater than this value for EndExpression, Pascal transfers to the first statement after the for loop, otherwise it executes Some_Statement. After the Pascal for loop executes Some_Statement, it adds one to variable and jumps back to the point where it compares the value of variable against the computed value for EndExpression. Converting this code directly into assembly language yields the following code:

```
;Note: This code assumes StartExpression and EndExpression are simple variables. ;If this is not the case, compute the values for these expression and place ;them in these variables.
mov ax, StartExpression
mov Variable, ax
ForLoop:
mov ax, Variable
cmp ax, EndExpression
jg ForDone
<Code for Some_Statement>
inc Variable
jmp ForLoop
ForDone:
```

To implement this as a set of macros, we need to be able to write a short piece of code that will write the above assembly language statements for us. At first blush, this would seem easy, why not use the following code?

```
ForLp macro Variable, Start, Stop
mov ax, Start
mov Variable, ax
ForLoop:
mov ax, Variable
cmp ax, Stop
jg ForDone
```

Page 411
These two macros would produce correct code – exactly once. However, a problem develops if you try to use these macros a second time. This is particularly evident when using nested loops:

```
ForLp I, 1, 10
ForLp J, 1, 10
.  .  .
Next J
Next I
```

The macros above emit the following 80x86 code:

```
mov ax, 1 ;The ForLp I, 1, 10
mov I, ax ; macro emits these
ForLoop:
  mov ax, I ; statements.
  cmp ax, 10 ;
  jg ForDone ;
  mov ax, 1 ;The ForLp J, 1, 10
  mov J, ax ; macro emits these
ForLoop:
  mov ax, J ; statements.
  cmp ax, 10 ;
  jg ForDone ;
  inc J ;The Next J macro emits these
  jmp ForLp ; statements.
Next:
  inc I ;The Next I macro emits these
  jmp ForLp ; statements.
ForDone:
```

The problem, evident in the code above, is that each time you use the ForLp macro you emit the label “ForLoop” to the code. Likewise, each time you use the Next macro, you emit the label “ForDone” to the code stream. Therefore, if you use these macros more than once (within the same procedure), you will get a duplicate symbol error. To prevent this error, the macros must generate unique labels each time you use them. Unfortunately, the local directive will not work here. The local directive defines a unique symbol within a single macro invocation. If you look carefully at the code above, you’ll see that the ForLp macro emits a symbol that the code in the Next macro references. Likewise, the Next macro emits a label that the ForLp macro references. Therefore, the label names must be global since the two macros can reference each other’s labels.

The solution the actual ForLp and Next macros use is to generate globally known labels of the form “$$For” + “variable name” + “some unique number.” and “$$Next” + “variable name” + “some unique number”. For the example given above, the real ForLp and Next macros would generate the following code:

```
mov ax, 1 ;The ForLp I, 1, 10
mov I, ax ; macro emits these
$$ForI0:
  mov ax, I ; statements.
  cmp ax, 10 ;
  jg $$NextI0 ;
  mov ax, 1 ;The ForLp J, 1, 10
  mov J, ax ; macro emits these
$$ForJ0:
  mov ax, J ; statements.
  cmp ax, 10 ;
```

The problem, evident in the code above, is that each time you use the ForLp macro you emit the label “ForLoop” to the code. Likewise, each time you use the Next macro, you emit the label “ForDone” to the code stream. Therefore, if you use these macros more than once (within the same procedure), you will get a duplicate symbol error. To prevent this error, the macros must generate unique labels each time you use them. Unfortunately, the local directive will not work here. The local directive defines a unique symbol within a single macro invocation. If you look carefully at the code above, you’ll see that the ForLp macro emits a symbol that the code in the Next macro references. Likewise, the Next macro emits a label that the ForLp macro references. Therefore, the label names must be global since the two macros can reference each other’s labels.

The solution the actual ForLp and Next macros use is to generate globally known labels of the form “$$For” + “variable name” + “some unique number.” and “$$Next” + “variable name” + “some unique number”. For the example given above, the real ForLp and Next macros would generate the following code:

```
mov ax, 1 ;The ForLp I, 1, 10
mov I, ax ; macro emits these
$$ForI0:
  mov ax, I ; statements.
  cmp ax, 10 ;
  jg $$NextI0 ;
  mov ax, 1 ;The ForLp J, 1, 10
  mov J, ax ; macro emits these
$$ForJ0:
  mov ax, J ; statements.
  cmp ax, 10 ;
```
Directives and Pseudo Opcodes

8.14.7 Macro Functions

A macro function is a macro whose sole purpose is to return a value for use in the operand field of some other statement. Although there is the obvious parallel between procedures and functions in a high level language and procedural macros and functional macros, the analogy is far from perfect. Macro functions do not let you create sequences of code that emit some instructions that compute a value when the program actually executes. Instead, macro functions simply compute some value at assembly time that MASM can use as an operand.

A good example of a macro function is the Date function. This macro function packs a five bit day, four bit month, and seven bit year value into 16 bits and returns that 16 bit value as the result. If you needed to create an initialized array of dates, you could use code like the following:

```assembly
DateArray word Date(2, 4, 84)
word Date(1, 1, 94)
word Date(7, 20, 60)
word Date(7, 19, 69)
word Date(6, 18, 74)
```

The Date function would pack the data and the word directive would emit the 16 bit packed value for each date to the object code file. You invoke macro functions by using their name where MASM expects a text expression of some sort. If the macro function requires any parameters, you must enclose them within parentheses, just like the parameters to Date, above.

Macro functions look exactly like standard macros with two exceptions: they do not contain any statements that generate code and they return a text value via an operand to the exitm directive. Note that you cannot return a numeric value with a macro function. If you need to return a numeric value, you must first convert it to a text value.

The following macro function implements Date using the 16 bit date format given in Chapter One (see "Bit Fields and Packed Data" on page 28):
The text expansion operator (“%”) is necessary in the operand field of the `exitm` directive because macro functions always return textual data, not numeric data. The expansion operator converts the numeric value to a string of digits acceptable to `exitm`.

One minor problem with the code above is that this function returns garbage if the date isn’t legal. A better design would generate an error if the input date is illegal. You can use the “.err” directive and conditional assembly to do this. The following implementation of `Date` checks the month, day, and year values to see if they are somewhat reasonable:

```assembly
Date macro month, day, year
local Value

if (month gt 12) or (month lt 1) or \
  (day gt 31) or (day lt 1) or \
  (year gt 99) (year lt 1)

  .err
  exitm <0> ;;Must return something!
endif

Value = (month shl 12) or (day shl 7) or year

exitm %Value
endm
```

With this version, any attempt to specify a totally outrageous date triggers the assembly of the “.err” directive that forces an error at assembly time.

### 8.14.8 Predefined Macros, Macro Functions, and Symbols

MASM provides four built-in macros and four corresponding macro functions. In addition, MASM also provides a large number of predefined symbols you can access during assembly. Although you would rarely use these macros, functions, and variables outside of moderately complex macros, they are essential when you do need them.

<table>
<thead>
<tr>
<th>Name</th>
<th>operands</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>substr</code></td>
<td>string, start, length</td>
<td>NewStr substr Oldstr, 1, 3</td>
<td>Returns a string consisting of the characters from start to start+length in the string operand. The length operand is optional. If it is not present, MASM returns all characters from position start through the end of the string.</td>
</tr>
<tr>
<td><code>instr</code></td>
<td>start, string, substr</td>
<td>Pos instr 2, OldStr, &lt;ax&gt;</td>
<td>Searches for “substr” within “string” starting at position “start.” The starting value is optional. If it is missing, MASM begins searching for the string from position one. If MASM cannot find the substring within the string operand, it returns the value zero.</td>
</tr>
<tr>
<td><code>sizestr</code></td>
<td>string</td>
<td>StrSize sizestr OldStr</td>
<td>Returns the size of the string in the operand field.</td>
</tr>
<tr>
<td><code>catstr</code></td>
<td>string, string, ...</td>
<td>NewStr catstr OldStr, &lt;$&gt;$</td>
<td>Creates a new string by concatenating each of the strings appearing in the operand field of the catstr macro.</td>
</tr>
</tbody>
</table>
The `substr` and `catstr` macros return text data. In some respects, they are similar to the `textequ` directive since you use them to assign textual data to a symbol at assembly time. The `instr` and `sizestr` are similar to the "=" directive insofar as they return a numeric value.

The `catstr` macro can eliminate the need for the `MakeLbl` macro found in the `ForLp` macro. Compare the following version of `ForLp` to the previous version (see “A Sample Macro to Implement For Loops” on page 409).

```masm
ForLp macro LCV, Start, Stop
local ForLoop
ifndef $$For&LCV&
$$For&LCV& = 0
else
$$For&LCV& = $$For&LCV& + 1
endif
mov ax, Start
mov LCV, ax
; Due to bug in MASM, this won’t actually work. The idea is sound, though
; Read on for correct solution.
ForLoop textequ @catstr($$For&LCV&, %$$For&LCV&)
&ForLoop&:
    mov ax, LCV
cmp ax, Stop
jgDone $$Next&LCV&, %$$For&LCV&
endm
```

MASM also provides macro function forms for `catstr`, `instr`, `sizestr`, and `substr`. To differentiate these macro functions from the corresponding predefined macros, MASM uses the names `@catstr`, `@instr`, `@sizestr`, and `@substr`. The following equivalences between these operations:

```plaintext
Symbol     catstr  String1, String2, ...
Symbol     textequ @catstr(String1, String2, ...)
Symbol     substr  SomeStr, 1, 5
Symbol     textequ @substr(SomeStr, 1, 5)
Symbol     instr   1, SomeStr, SearchStr
Symbol     =      @substr(1, SomeStr, SearchStr)
Symbol     sizestr SomeStr
Symbol     =      @sizestr(SomeStr)
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>@substr</td>
<td>string, start, length&lt;br&gt;Returns: text data</td>
<td>ifiddn @substr(parm, 1, 4), &lt;[bx]&gt;</td>
</tr>
<tr>
<td>@instr</td>
<td>start, string, substr&lt;br&gt;Returns: numeric data</td>
<td>if @instr(parm,&lt;bx&gt;)</td>
</tr>
<tr>
<td>@sizestr</td>
<td>string&lt;br&gt;Returns: numeric data</td>
<td>byte @sizestr(SomeStr)</td>
</tr>
<tr>
<td>@catstr</td>
<td>string, string,...&lt;br&gt;Returns: text data</td>
<td>jg @catstr($$Next&amp;LCV&amp;, %$$For&amp;LCV&amp;)</td>
</tr>
</tbody>
</table>

The last example above shows how to get rid of the `jgDone` and `jmpLoop` macros in the `ForLp` macro. A final, improved, version of the `ForLp` and `Next` macros, eliminating the three support macros and working around the bug in MASM might look something like the following:

Table 44: MASM Predefined Macro Functions
MASM also provides a large number of built-in variables that return information about the current assembly. The following table describes these built-in assembly time variables.

### Table 45: MASM Predefined Assembly Time Variables

<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>Description</th>
<th>Return result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date &amp; Time Information</td>
<td>@Date</td>
<td>Returns the date of assembly.</td>
<td>Text value</td>
</tr>
<tr>
<td></td>
<td>@Time</td>
<td>Returns a string denoting the time of assembly.</td>
<td>Text value</td>
</tr>
</tbody>
</table>
### Environment Information

<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>Description</th>
<th>Return result</th>
</tr>
</thead>
<tbody>
<tr>
<td>@CPU</td>
<td>Returns a 16 bit value whose bits determine the active processor directive. Specifying the .8086, .186, .286, .386, .486, and .586 directives enable additional instructions in MASM. They also set the corresponding bits in the @cpu variable. Note that MASM sets all the bits for the processors it can handle at any one given time. For example, if you use the .386 directive, MASM sets bits zero, one, two, and three in the @cpu variable.</td>
<td>Bit 0 - 8086 instrs permissible. Bit 1 - 80186 instrs permissible. Bit 2 - 80286 instrs permissible. Bit 3 - 80386 instrs permissible. Bit 4 - 80486 instrs permissible. Bit 5 - Pentium instrs permissible. Bit 6 - Reserved for 80686 (?). Bit 7 - Protected mode instrs okay. Bit 8 - 8087 instrs permissible. Bit 10 - 80287 instrs permissible. Bit 11 - 80386 instrs permissible. (bit 11 is also set for 80486 and Pentium instr sets).</td>
<td></td>
</tr>
<tr>
<td>@Environ</td>
<td>@Environ(name) returns the text associated with DOS environment variable name. The parameter must be a text value that evaluates to a valid DOS environment variable name.</td>
<td>Text value</td>
<td></td>
</tr>
<tr>
<td>@Interface</td>
<td>Returns a numeric value denoting the current language type in use. Note that this information is similar to that provided by the opattr attribute. The H.O. bit determines if you are assembling code for MS-DOS/Windows or OS/2. This directive is mainly useful for those using MASM’s simplified segment directives. Since this text does not deal with the simplified directives, further discussion of this variable is unwarranted.</td>
<td>Bits 0-2 000- No language type 001- C 010- SYSCALL 011- STDCALL 100- Pascal 101- FORTRAN 110- BASIC Bit 7 0- MS-DOS or Windows 1- OS/2</td>
<td></td>
</tr>
<tr>
<td>@Version</td>
<td>Returns a numeric value that is the current MASM version number multiplied by 100. For example, MASM 6.11’s @version variable returns 611.</td>
<td>Numeric value</td>
<td></td>
</tr>
<tr>
<td>@FileCur</td>
<td>Returns the current source or include file name, including any necessary pathname information.</td>
<td>Text value</td>
<td></td>
</tr>
<tr>
<td>@FileName</td>
<td>Returns the current source file name (base name only, no path information). If in an include file, this variable returns the name of the source file that included the current file.</td>
<td>Text value</td>
<td></td>
</tr>
<tr>
<td>@Line</td>
<td>Returns the current line number in the source file.</td>
<td>Numeric value</td>
<td></td>
</tr>
</tbody>
</table>
Although there is insufficient space to go into detail about the possible uses for each of these variables, a few examples might demonstrate some of the possibilities. Other uses of these variables will appear throughout the text; however, the most impressive uses will be the ones you discover.

The @CPU variable is quite useful if you want to assemble different code sequences in your program for different processors. The section on conditional assembly in this chapter described how you could create a symbol to determine if you are assembling the code for an 80386 and later processor or a stock 8086 processor. The @CPU symbol provides a symbol that will tell you exactly which instructions are allowable at any given point in your program. The following is a rework of that example using the @CPU variable:

```assembly
if @CPU and 100b ;Need an 80286 or later processor
    shl ax, 4 ; for this instruction.
else ;Must be 8086 processor.
    mov cl, 4
    shl ax, cl
endif
```

You can use the @Line directive to put special diagnostic messages in your code. The following code would print an error message including the line number in the source file of the offending assertion, if it detects an error at run-time:

```assembly
mov ax, ErrorFlag
cmp ax, 0
je NoError
mov ax, @Line ;Load AX with current line #
call PrintError ;Go print error message and Line #
jmp Quit ;Terminate program.
```

### 8.14.9 Macros vs. Text Equates

Macros, macro functions, and text equates all substitute text in a program. While there is some overlap between them, they really do serve different purposes in an assembly language program.
Text equates perform a single text substitution on a line. They do not allow any parameters. However, you can replace text anywhere on a line with a text equate. You can expand a text equate in the label, mnemonic, operand, or even the comment field. Furthermore, you can replace multiple fields, even an entire line with a single symbol.

Macro functions are legal in the operand field only. However, you can pass parameters to macro functions making them considerably more general than simple text equates.

Procedural macros let you emit sequences of statements (with text equates you can emit, at most, one statement).

8.14.10 Macros: Good and Bad News

Macros offer considerable convenience. They let you insert several instructions into your source file by simply typing a single command. This can save you an incredible amount of typing when entering huge tables, each line of which contains some bizarre, but repeated calculation. It's useful (in certain cases) for helping make your programs more readable. Few would argue that ForLp I,1,10 is not more readable than the corresponding 80x86 code. Unfortunately, it's easy to get carried away and produce code that is inefficient, hard to read, and hard to maintain.

A lot of so-called “advanced” assembly language programmers get carried away with the idea that they can create their own instructions via macro definitions and they start creating macros for every imaginable function under the sun. The COPY macro presented earlier is a good example. The 80x86 doesn’t support a memory to memory move operation. Fine, we’ll create a macro that does the job for us. Soon, the assembly language program doesn’t look like 80x86 assembly language at all. Instead, a large number of the statements are macro invocations. Now this may be great for the programmer who has created all these macros and intimately understands their operation. To the 80x86 programmer who isn’t familiar with those macros, however, it’s all gibberish. Maintaining a program someone else wrote, that contains “new” instructions implemented via macros, is a horrible task. Therefore, you should rarely use macros as a device to create new instructions on the 80x86.

Another problem with macros is that they tend to hide side effects. Consider the COPY macro presented earlier. If you encountered a statement of the form COPY VAR1,VAR2 in an assembly language program, you’d think that this was an innocuous statement that copies VAR2 to VAR1. Wrong! It also destroys the current contents of the ax register leaving a copy of the value in VAR2 in the ax register. This macro invocation doesn’t make this very clear. Consider the following code sequence:

```
mov ax, 5
copy Var2, Var1
mov Var1, ax
```

This code sequence copies Var1 into Var2 and then (supposedly) stores five into Var1. Unfortunately, the COPY macro has wiped out the value in ax (leaving the value originally contained in Var1 alone), so this instruction sequence does not modify Var1 at all!

Another problem with macros is efficiency. Consider the following invocations of the COPY macro:

```
copy Var3, Var1
copy Var2, Var1
copy Var0, Var1
```

These three statements generate the code:

```
mov ax, Var1
mov Var3, ax
mov ax, Var1
mov Var2, ax
mov ax, Var1
mov Var0, ax
```
Clearly, the last two `mov ax,Var1` instructions are superfluous. The `ax` register already contains a copy of `Var1`, there is no need to reload `ax` with this value. Unfortunately, this inefficiency, while perfectly obvious in the expanded code, isn't obvious at all in the macro invocations.

Another problem with macros is complexity. In order to generate efficient code, you can create extremely complex macros using conditional assembly (especially `ifb`, `ifidn`, etc.), repeat loops (described a little later), and other directives. Unfortunately, these macros are small programs all on their own. You can have bugs in your macros just as you can have bugs in your assembly language program. And the more complex your macros become, the more likely they'll contain bugs that will, of course, become bugs in your program when invoking the macro.

Overusing macros, especially complex ones, produces hard to read code that is hard to maintain. Despite the enthusiastic claims of those who love macros, the unbridled use of macros within a program generally causes more bugs than it helps to prevent. If you're going to use macros, go easy on them.

There is a good side to macros, however. If you standardize on a set of macros and document all your programs as using these macros, they may help make your programs more readable. Especially if those macros have easily identifiable names. The UCR Standard Library for 80x86 Assembly Language Programmers uses macros for most library calls. You'll read more about the UCR Standard Library in the next chapter.

### 8.15 Repeat Operations

Another macro format (at least by Microsoft's definition) is the repeat macro. A repeat macro is nothing more than a loop that repeats the statements within the loop some specified number of times. There are three types of repeat macros provided by MASM: `repeat/rept`, `for/irp`, and `forc/irpc`. The `repeat/rept` macro uses the following syntax:

```
repeat  expression
<statements>
endm
```

Expression must be a numeric expression that evaluates to an unsigned constant. The `repeat` directive duplicates all the statements between `repeat` and `endm` that many times. The following code generates a table of 26 bytes containing the 26 uppercase characters:

```
ASCIICode  =  'A'
repeat  26
byte    ASCIICode
ASCIICode  =  ASCIICode+1
endm
```

The symbol `ASCIICode` is assigned the ASCII code for “A”. The loop repeats 26 times, each time emitting a byte with the value of `ASCIICode`. Also, the loop increments the `ASCIICode` symbol on each repetition so that it contains the ASCII code of the next character in the ASCII table. This effectively generates the following statements:

```
byte    'A'
byte    'B'
...
byte    'Y'
byte    'Z'
ASCIICode  =  27
```

Note that the `repeat` loop executes at assembly time, not at run time. Repeat is not a mechanism for creating loops within your program; use it for replicating sections of code within your program. If you want to create a loop that executes some number of times within your program, use the `loop` instruction. Although the following two code sequences produce the same result, they are not the same:
; Code sequence using a run-time loop:
    mov    cx, 10
AddLp:   add    ax, [bx]
    add    bx, 2
    loop   AddLp

; Code sequence using an assembly-time loop:
    repeat 10
    add    ax, [bx]
    add    bx, 2
    endm

The first code sequence above emits four machine instructions to the object code file. At assembly time, the 80x86 CPU executes the statements between AddLp and the loop instruction ten times under the control of the loop instruction. The second code sequence above emits 20 instructions to the object code file. At run time, the 80x86 CPU simply executes these 20 instructions sequentially, with no control transfer. The second form will be faster, since the 80x86 does not have to execute the loop instruction every third instruction. On the other hand, the second version is also much larger because it replicates the body of the loop ten times in the object code file.

Unlike standard macros, you do not define and invoke \texttt{repeat} macros separately. MASM emits the code between the \texttt{repeat} and \texttt{endm} directives upon encountering the \texttt{repeat} directive. There isn't a separate invocation phase. If you want to create a \texttt{repeat} macro that can be invoked throughout your program, consider the following:

\begin{verbatim}
REPTMacro macro Count
    repeat Count
        <statements>
    endm
endm
\end{verbatim}

By placing the \texttt{repeat} macro inside a standard macro, you can invoke the \texttt{repeat} macro anywhere in your program by invoking the \texttt{REPTMacro} macro. Note that you need two \texttt{endm} directives, one to terminate the \texttt{repeat} macro, one to terminate the standard macro.

\texttt{Rept} is a synonym for \texttt{repeat}. \texttt{Repeat} is the newer form, MASM supports \texttt{Rept} for compatibility with older source files. You should always use the \texttt{repeat} form.

\section{The FOR and FORC Macro Operations}

Another form of the \texttt{repeat} macro is the \texttt{for} macro. This macro takes the following form:

\begin{verbatim}
for parameter,<item1 {,item2 {,item3 {,...}}}>
    <statements>
endm
\end{verbatim}

The angle brackets are required around the items in the operand field of the \texttt{for} directive. The braces surround optional items, the braces should not appear in the operand field.

The \texttt{for} directive replicates the instructions between \texttt{for} and \texttt{endm} once for each item appearing in the operand field. Furthermore, for each iteration, the first symbol in the operand field is assigned the value of the successive items from the second parameter. Consider the following loop:

\begin{verbatim}
for    value,<0,1,2,3,4,5>
    byte   value
endm
\end{verbatim}

This loop emits six bytes containing the values zero, one, two, ..., five. It is absolutely identical to the sequence of instructions:
Remember, the for loop, like the repeat loop, executes at assembly time, not at run time.

For’s second operand need not be a literal text constant; you can supply a macro parameter, macro function result, or a text equate for this value. Keep in mind, though, that this parameter must expand to a text value with the text delimiters around it.

irp is an older, obsolete, synonym for for. MASM allows irp to provide compatibility with older source code. However, you should always use the for directive.

The third form of the loop macro is the forc macro. It differs from the for macro in that it repeats a loop the number of times specified by the length of a character string rather than by the number of operands present. The syntax for the forc directive is

```plaintext
forc parameter,<string>
<statements>
endm
```

The statements in the loop repeat once for each character in the string operand. The angle brackets must appear around the string. Consider the following loop:

```plaintext
forc value,<012345>
byte value
endm
```

This loop produces the same code as the example for the for directive above.

Irpc is an old synonym for forc provided for compatibility reasons. You should always use forc in your new code.

### 8.17 The WHILE Macro Operation

The while macro lets you repeat a sequence of code in your assembly language file an indefinite number of times. An assembly time expression, that while evaluates before emitting the code for each loop, determines whether it repeats. The syntax for this macro is

```plaintext
while expression
<Statements>
endm
```

This macro evaluates the assembly-time expression; if this expression’s value is zero, the while macro ignores the statements up to the corresponding endm directive. If the expression evaluates to a non-zero value (true), then MASM assembles the statements up to the endm directive and reevaluates the expression to see if it should assemble the body of the while loop again.

Normally, the while directive repeats the statements between the while and endm as long as the expression evaluates true. However, you can also use the exitm directive to prematurely terminate the expansion of the loop body. Keep in mind that you need to provide some condition that terminates the loop, otherwise MASM will go into an infinite loop and continually emit code to the object code file until the disk fills up (or it will simply go into an infinite loop if the loop does not emit any code).

### 8.18 Macro Parameters

Standard MASM macros are very flexible. If the number of actual parameters (those supplied in the operand field of the macro invocation) does not match the number of for-
mal parameters (those appearing in the operand field of the macro definition), MASM won’t necessarily complain. If there are more actual parameters than formal parameters, MASM ignores the extra parameters and generates a warning. If there are more formal parameters than actual parameters, MASM substitutes the empty string (“<>”) for the extra formal parameters. By using the #b and #nb conditional assembly directives, you can test this last condition. While this parameter substitution technique is flexible, it also leaves open the possibility of error. If you want to require that the programmer supply exactly three parameters and they actually supply less, MASM will not generate an error. If you forget to test for the presence of each parameter using #b, you could generate bad code. To overcome this limitation, MASM provides the ability to specify that certain macro parameters are required. You can also assign a default value to a parameter if the programming doesn’t supply one. Finally, MASM also provides facilities to allow a variable number of macro arguments.

If you want to require a programmer to supply a particular macro parameters, simply put “.req” after the macro parameter in the macro definition. At assembly time, MASM will generate an error if that particular macro is missing.

```
Needs2Parms macro parm1:req, parm2:req
    ...
endm

Needs2Parms ax ;Generates an error.
Needs2Parms ;Generates an error.
Needs2Parms ax, bx ;Works fine.
```

Another possibility is to have the macro supply a default value for a macro if it is missing from the actual parameter list. To do this, simply use the “:=<text>” operator immediately after the parameter name in the formal parameter list. For example, the int 10h BIOS function provides various video services. One of the most commonly used video services is the ah=0eh function that outputs the character in al to the video display. The following macro lets the caller specify which function they want to use, and defaults to function 0eh if they don’t specify a parameter:

```
Video macro service := <0eh>
    mov ah, service
    int 10h
endm
```

The last feature MASM’s macros support is the ability to process a variable number of parameters. To do this you simply place the operator “.vararg” after the last formal parameter in the parameter list. MASM associates the first $n$ actual parameters with the corresponding formal parameters appearing before the variable argument, it then creates a text equate of all remaining parameters to the formal parameter suffixed with the “.vararg” operator. You can use the for macro to extract each parameter from this variable argument list. For example, the following macro lets you declare an arbitrary number of two dimensional arrays, all the same size. The first two parameters specify the number of rows and columns, the remaining optional parameters specify the names of the arrays:

```
MkArrays macro NumRows:req, NumCols:req, Names:vararg
    for AryName, Names
        &AryName& word NumRows dup (NumCols dup (?))
    endm
endm

MkArrays 8, 12, A, B, X, Y
```
8.19 Controlling the Listing

MASM provides several assembler directives that are useful for controlling the output of the assembler. These directives include `echo`, `%out`, `title`, `subttl`, `page`, `.list`, `.nolist`, and `.xlist`. There are several others, but these are the most important.

8.19.1 The ECHO and %OUT Directives

The `echo` and `%out` directives simply print whatever appears in its operand field to the video display during assembly. Some examples of `echo` and `%out` appeared in the sections on conditional assembly and macros. Note that `%out` is an older form of `echo` provided for compatibility with old source code. You should use `echo` in all your new code.

8.19.2 The TITLE Directive

The `title` assembler directive assigns a title to your source file. Only one `title` directive may appear in your program. The syntax for this directive is

```
title text
```

MASM will print the specified text at the top of each page of the assembled listing.

8.19.3 The SUBTTL Directive

The `subttl` (subtitle) directive is similar to the `title` directive, except multiple subtitles may appear within your source file. Subtitles appear immediately below the title at the top of each page in the assembled listing. The syntax for the `subttl` directive is

```
subttl text
```

The specified text will become the new subtitle. Note that MASM will not print the new subtitle until the next page eject. If you wish to place the subtitle on the same page as the code immediately following the directive, use the `page` directive (described next) to force a page eject.

8.19.4 The PAGE Directive

The `page` directive performs two functions- it can force a page eject in the assembly listing and it can set the width and length of the output device. To force a page eject, the following form of the `page` directive is used:

```
page
```

If you place a plus sign, “+”, in the operand field, then MASM performs a page break, increments the section number, and resets the page number to one. MASM prints page numbers using the format

```
section-page
```

If you want to take advantage of the section number facility, you will have to manually insert page breaks (with a “+” operand) in front of each new section.

The second form of the `page` command lets you set the printer page width and length values. It takes the form:

```
page length, width
```

where length is the number of lines per page (defaults to 50, but 56-60 is a better choice for most printers) and width is the number of characters per line. The default page width is
80 characters. If your printer is capable of printing 132 columns, you should change this value to 132 so your listings will be easier to read. Note that some printers, even if their carriage is only 8-1/2" wide, will print at least 132 columns across in a condensed mode. Typically some control character must be sent to the printer to place it in condensed mode. You can insert such a control character in a comment at the beginning of your source listing.

8.19.5 The .LIST, .NOLIST, and .XLIST Directives

The .list, .nolist, and .xlist directives can be used to selectively list portions of your source file during assembly. .List turns the listing on, .Nolist turns the listing off. .Xlist is an obsolete form of .Nolist for older code.

By sprinkling these three directives throughout your source file, you can list only those sections of code that interest you. None of these directives accept any operands. They take the following forms:

```
.list
.nolist
.xlist
```  

8.19.6 Other Listing Directives

MASM provides several other listing control directives that this chapter will not cover. These let you control the output of macros, conditional assembly segments, and so on to the listing file. Please see the appendices for details on these directives.

8.20 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 defined 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 file is fine. 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 five 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 files into manageable chunks. Then you
assemble 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 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 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 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.

**8.20.1 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 join these modules together when you assemble your file. The following example would include 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 the `include` during assembly, exactly as though you had typed that code 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.

---

16. There is nothing wrong with this, other than the fact that it does not take advantage of separate compilation.
8.20.2 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 specific 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. Extn is an older directive that is a synonym for extern. It provides compatibility with old source files. You should always use the extern directive in new source code.

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. Extn is an older directive that is a synonym for extern. It provides compatibility with old source files. 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:

<table>
<thead>
<tr>
<th>DSEG</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>segment</td>
<td>para public 'data'</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Var1</th>
<th>word ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Var2</td>
<td>word ?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DSEG</th>
<th></th>
<th></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>CSEG</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>segment</td>
<td>para public 'code'</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>assume</th>
<th>cs:cseg, ds:dseg</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Proc1</th>
<th>proc near</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov</td>
<td>ax, Var1</td>
</tr>
<tr>
<td>add</td>
<td>ax, Var2</td>
</tr>
<tr>
<td>mov</td>
<td>Var1, ax</td>
</tr>
<tr>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>

CSEG ends

<table>
<thead>
<tr>
<th>Proc1</th>
<th>endp</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>CSEG</th>
<th>ends</th>
</tr>
</thead>
</table>

| end  |         |

;Module #2:

<table>
<thead>
<tr>
<th>CSEG</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>segment</td>
<td>para public 'code'</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mov</th>
<th>Var1, 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov</td>
<td>Var2, 3</td>
</tr>
<tr>
<td>call</td>
<td>Proc1</td>
</tr>
</tbody>
</table>

| :       |         |

<table>
<thead>
<tr>
<th>CSEG</th>
<th>ends</th>
</tr>
</thead>
</table>

| 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. With MASM 6.x, you would use a command like

ML /c pubs.asm

The “/c” option tells MASM to perform a “compile-only” assembly. That is, it will not try to link the code after a successful assembly. This produces a “pubs.obj” object module.

Next, assemble the file containing the external definitions and link in the code using the MASM command:

ML exts.asm pubs.obj

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. This, as it turns out, is the source of problems which Microsoft solved with the externdef directive.

8.20.3 The EXTERNDEF Directive

The externdef directive is a combination of public and extern all rolled into one. It uses the same syntax as the extern directive, that is, you place a list of name:type entries in the operand field. If MASM does not encounter another definition of the symbol in the current source file, externdef behaves exactly like the extern statement. If the symbol does appear in the source file, then externdef behaves like the public command. With externdef there really is no need to use the public or extern statements unless you feel somehow compelled to do so.

The important benefit of the externdef directive 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 for other programs. In addition to sharing some routines and some variables, suppose you want to share constants and macros as well. The include file mechanism provides a perfect way to handle this. You simply create an include file containing the constants, macros, and externdef definitions and include this file in the module that implements your routines and in the modules that use those routines (see Figure 8.8).

Note that extern and public wouldn’t work in this case because the implementation module needs the public directive and the using module needs the extern directive. You would have to create two separate header files. Maintaining two separate header files that
contain mostly identical definitions is not a good idea. The `externdef` directive provides a solution.

Within your headers files you should create segment definitions that match those in the including modules. Be sure to put the `externdef` directives inside the same segments in which the symbol is actually defined. This associates a segment value with the symbol so that MASM can properly make appropriate optimizations and other calculations based on the symbol’s full address:

```masm
; From "HEADER.A" file:
cseg segment para public 'code'
externdef Routine1:near, Routine2:far
externdef i:word, b:byte, flag:byte
dseg ends
```

This text adopts the UCR Standard Library convention of using an “.a” suffix for assembly language header files. Other common suffixes in use include “.inc” and “.def”.

### 8.21 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 and pgmb.obj exist. Finally, you make changes to pgma.asm and pgmb.asm and assemble the pgma.asm but forget to assemble the 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: nmake. The nmake 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 assembly-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 .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

---

17. 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.
assembly language source files (.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 tabstop. The pgm.exe statement above would probably look something like the following:

```
pgm.exe: pgma.obj pgmb.obj
ml /Fepgm.exe pgma.obj pgmb.obj
```

(The “/Fepgm.exe” option tells MASM to name the executable file “pgm.exe.”)

If you need to execute more than one command to resolve the dependencies, you can place several commands after the dependency statement in the appropriate order. Note that you must indent all commands one tab stop. Nmake.exe 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, pgm.exe depends upon the pgma.obj and pgmb.obj files. Obviously, the .obj files depend upon the source files that generated them. Therefore, before attempting to resolve the dependencies for pgm.exe, nmake.exe will first check out the rest of the make file to see if pgma.obj or pgmb.obj depends on anything. If they do, nmake.exe will resolve those dependencies first. Consider the following make file:

```
pgm.exe: pgma.obj pgmb.obj
  ml /Fepgm.exe pgma.obj pgmb.obj
pgma.obj: pgma.asm
  ml /c pgma.asm
pgmb.obj: pgmb.asm
  ml /c pgmb.asm
```

The nmake.exe program will process the first dependency line it finds in the file. However, the files pgm.exe depends upon themselves have dependency lines. Therefore, nmake.exe will first ensure that pgma.obj and pgmb.obj are up to date before attempting to execute MASM to link these files together. Therefore, if the only change you’ve made has been to pgmb.asm, nmake.exe takes the following steps (assuming pgma.obj exists and is up to date).
1. Nmake.exe 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. Nmake.exe processes the pgma.obj dependency line. It notices that the pgma.obj file is newer than the pgma.asm file, so it does not execute the command following this dependency statement.

3. Nmake.exe processes the pgmb.obj dependency line. It notes that pgmb.obj is older than pgmb.asm (since we just changed the pgmb.asm source file). Therefore, nmake.exe executes the DOS command following on the next line. This generates a new pgmb.obj file that is now up to date.

4. Having process the pgma.obj and pgmb.obj dependencies, nmake.exe now returns its attention to the first dependency line. Since nmake.exe just created a new pgmb.obj file, its date/time stamp will be newer than pgm.exe’s. Therefore, nmake.exe will execute the ml 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 nmake.exe to assembly only those modules absolutely necessary to produce a consistent executable file. In the example above, nmake.exe did not bother to assemble pgma.asm since its object file was already up to date.

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

```
pgm.exe: pgma.obj pgmb.obj
   ml /Fepgm.exe pgma.obj pgmb.obj

pgma.obj: pgma.asm pgm.a
   ml /c pgma.asm

pgmb.obj: pgmb.asm pgm.a
   ml /c pgmb.asm
```

Note that any changes to the pgm.a file will force nmake.exe to reassemble both pgm.asm and pgmb.asm since the pgma.obj and pgmb.obj files both depend upon the pgm.a include file. Leaving include files out of a dependency list is a common mistake programmers make that can produce inconsistent .exe files.

Note that you would not normally need to specify the UCR Standard Library include files nor the Standard Library .lib files in the dependency list. True, your resulting .exe 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 .exe and .obj files and force a reassembly of the entire system.

Nmake.exe, by default, assumes that it will be processing a make file named “makefile”. When you run nmake.exe, it looks for “makefile” in the current directory. If it doesn’t find this file, it complains and terminates. 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 nmake.exe program.

Although this section discusses the nmake program in sufficient detail to handle most projects you will be working on, keep in mind that nmake.exe provides considerable functionality that this chapter does not discuss. To learn more about the nmake.exe program, consult the documentation that comes with MASM.

---

18. 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.
8.22 Sample Program

Here is a single program that demonstrates most of the concepts from this chapter. This program consists of several files, including a makefile, that you can assemble and link using the nmake.exe program. This particular sample program computes “cross products” of various functions. The multiplication table you learned in school is a good example of a cross product, so are the truth tables found in Chapter Two of your textbook. This particular program generates cross product tables for addition, subtraction, division, and, optionally, remainder (modulo). In addition to demonstrating several concepts from this chapter, this sample program also demonstrates how to manipulate dynamically allocated arrays. This particular program asks the user to input the matrix size (row and column sizes) and then computes an appropriate set of cross products for that array.

8.22.1 EX8.MAK

The cross product program contains several modules. The following make file assembles all necessary files to ensure a consistent .EXE file.

```
ex8.exe: ex8.obj geti.obj getarray.obj xproduct.obj matrix.a
   ml ex8.obj geti.obj getarray.obj xproduct.obj

ex8.obj: ex8.asm matrix.a
   ml /c ex8.asm

geti.obj: geti.asm matrix.a
   ml /c geti.asm

getarray.obj: getarray.asm matrix.a
   ml /c getarray.asm

xproduct.obj: xproduct.asm matrix.a
   ml /c xproduct.asm
```

8.22.2 Matrix.A

MATRIX.A is the header file containing definitions that the cross product program uses. It also contains all the externdef statements for all externally defined routines.

```
; MATRIX.A
;
; This include file provides the external definitions
; and data type definitions for the matrix sample program
; in Chapter Eight.
;
; Some useful type definitions:

Integer       typedef word
Char           typedef byte

; Some common constants:

Bell          equ   07; ASCII code for the bell character.

; A “Dope Vector” is a structure containing information about arrays that
; a program allocates dynamically during program execution. This particular
; dope vector handles two dimensional arrays. It uses the following fields:
;
; TTL- Points at a zero terminated string containing a description
; of the data in the array.
;
; Func- Pointer to function to compute for this matrix.
```
Directives and Pseudo Opcodes

; Data- Pointer to the base address of the array.
; Dim1- This is a word containing the number of rows in the array.
; Dim2- This is a word containing the number of elements per row in the array.
; ESize- Contains the number of bytes per element in the array.

DopeVec struct
  TTL    dword  
  Func   dword  
  Data   dword  
  Dim1   word   
  Dim2   word   
  ESize  word   
DopeVec ends

; Some text equates the matrix code commonly uses:
Base textequ <es:[di]>
byptextequ <byte ptr>
wptextequ <word ptr>
dptextequ <dword ptr>

; Procedure declarations.
InpSeg segment para public 'input'
  externdef geti:far
  externdef getarray:far
InpSeg ends

cseg segment para public 'code'
  externdef CrossProduct:near
cseg ends

; Variable declarations
dseg segment para public 'data'
  externdef InputLine:byte
dseg ends

; Uncomment the following equates if you want to turn on the debugging statements or if you want to include the MODULO function.
;debug equ 0
;DoMOD equ 0

8.22.3 EX8.ASM

This is the main program. It calls appropriate routines to get the user input, compute the cross product, and print the result.

; Sample program for Chapter Eight.
; Demonstrates the use of many MASM features discussed in Chapter Six
; including label types, constants, segment ordering, procedures, equates,
; address expressions, coercion and type operators, segment prefixes,
Chapter 08

; the assume directive, conditional assembly, macros, listing directives,  
; separate assembly, and using the UCR Standard Library.

; Include the header files for the UCR Standard Library. Note that the  
; "stdlib.a" file defines two segments; MASM will load these segments into  
; memory before "dseg" in this program.

; The ".nolist" directive tells MASM not to list out all the macros for  
; the standard library when producing an assembly listing. Doing so would  
; increase the size of the listing by many tens of pages and would tend to  
; obscure the real code in this program.

; The ".list" directive turns the listing back on after MASM gets past the  
; standard library files. Note that these two directives (".nolist" and  
; ".list") are only active if you produce an assembly listing using MASM’s  
; "/F1" command line parameter.

.nolist
include stdlib.a
includelib stdlib.lib
.list

; The following statement includes the special header file for this  
; particular program. The header file contains external definitions  
; and various data type definitions.

include matrix.a

; The following two statements allow us to use 80386 instructions  
; in the program. The ".386" directive turns on the 80386 instruction  
; set, the "option" directive tells MASM to use 16-bit segments by  
; default (when using 80386 instructions, 32-bit segments are the default).  
; DOS real mode programs must be written using 16-bit segments.

/.386
option segment:use16

dseg segment para public 'data'

Rows integer ? ; Number of rows in matrices
Columns integer ? ; Number of columns in matrices

; Input line is an input buffer this code uses to read a string of text  
; from the user. In particular, the GetWholeNumber procedure passes the  
; address of InputLine to the GETS routine that reads a line of text  
; from the user and places each character into this array. GETS reads  
; a maximum of 127 characters plus the enter key from the user. It zero  
; terminates that string (replacing the ASCII code for the ENTER key with  
; a zero). Therefore, this array needs to be at least 128 bytes long to  
; prevent the possibility of buffer overflow.

; Note that the GetArray module also uses this array.

InputLine char 128 dup (0)

; The following two pointers point at arrays of integers.  
; This program dynamically allocates storage for the actual array data  
; once the user tells the program how big the arrays should be. The  
; Rows and Columns variables above determine the respective sizes of  
; these arrays. After allocating the storage with a call to MALLOC,  
; this program stores the pointers to these arrays into the following  
; two pointer variables.
RowArray    dword    ? ; Pointer to Row values
ColArray    dword    ? ; Pointer to column values.

; ResultArrays is an array of dope vectors(*) to hold the results
; from the matrix operations:
;
; [0]- addition table
; [1]- subtraction table
; [2]- multiplication table
; [3]- division table
;
; [4]- modulo (remainder) table -- if the symbol “DoMOD” is defined.
;
; The equate that follows the ResultArrays declaration computes the number
; of elements in the array. “$” is the offset into dseg immediately after
; the last byte of ResultArrays. Subtracting this value from ResultArrays
; computes the number of bytes in ResultArrays. Dividing this by the size
; of a single dope vector produces the number of elements in the array.
; This is an excellent example of how you can use address expressions in
; an assembly language program.
;
; TheIFDEF DoMOD code demonstrates how easy it is to extend this matrix.
; Defining the symbol "DoMOD" adds another entry to this array. The
; rest of the program adjusts for this new entry automatically.
;
; You can easily add new items to this array of dope vectors. You will
; need to supply a title and a function to compute the matrice’s entries.
; Other than that, however, this program automatically adjusts to any new
; entries you add to the dope vector array.
;
; (*) A “Dope Vector” is a data structure that describes a dynamically
; allocated array. A typical dope vector contains the maximum value for
; each dimension, a pointer to the array data in memory, and some other
; possible information. This program also stores a pointer to an array
; title and a pointer to an arithmetic function in the dope vector.

ResultArrays    DopeVec [AddTbl,Addition], [SubTbl,Subtraction]
DopeVec [MulTbl,Multiplication], [DivTbl,Division]

ifdef DoMOD
DopeVec [ModTbl,Modulo]
endif

; Add any new functions of your own at this point, before the following equate:

RASize = ($-ResultArrays) / (sizeof DopeVec)

; Titles for each of the four (five) matrices.

AddTbl    char   "Addition Table",0
SubTbl    char   "Subtraction Table",0
MulTbl    char   "Multiplication Table",0
DivTbl    char   "Division Table",0

ifdef DoMOD
ModTbl    char   "Modulo (Remainder) Table",0
endif

; This would be a good place to put a title for any new array you create.

dseg    ends
Putting PrintMat inside its own segment demonstrates that you can have multiple code segments within a program. There is no reason we couldn’t have put “PrintMat” in CSEG other than to demonstrate a far call to a different segment.

PrintSeg segment para public 'PrintSeg'

PrintMat- Prints a matrix for the cross product operation.
On Entry:
DS must point at DSEG.
DS:SI points at the entry in ResultArrays for the array to print.
The output takes the following form:

Matrix Title

<- column matrix values ->

^   *------------------------*
 |      |                        |
R      |                        |
o      | Cross Product Matrix   |
w      |       Values           |
|      |                        |
V      |                        |
a      |                        |
l      |                        |
u      |                        |
e      |                        |
s      |                        |
v      |------------------------*

PrintMat proc far
assume ds:dseg

Note the use of conditional assembly to insert extra debugging statements if a special symbol “debug” is defined during assembly. If such a symbol is not defined during assembly, the assembler ignores the following statements:

ifdef debug
print char “In PrintMat”, cr,lf,0
endif

First, print the title of this table. The TTL field in the dope vector contains a pointer to a zero terminated title string. Load this pointer into es:di and call PUTS to print that string.

putcrl
les     di, [si].DopeVec.TTL
puts

Now print the column values. Note the use of PUTISIZE so that each value takes exactly six print positions. The following loop repeats once for each element in the Column array (the number of elements in the column array is given by the Dim2 field in the dope vector).

mov     dx, [si].DopeVec.Dim2        ;# times to repeat the loop.
les     di, ColArray                ;Base address of array.

print    char cr,lf,lf,”        ;Skip spaces to move past the row values.
”,”0     ;row values.

mov      cx, 6            ;Print the value using a
purisize ; minimum of six positions.
add      di, 2            ;Move on to next element.
inc      dx              ;Repeat this loop DIM2 times.
jne      ColValLp
purcr     ;End of column array output
purcr     ;Insert a blank line.

; Now output each row of the matrix. Note that we need to output the
; RowArray value before each row of the matrix.
;
; RowLp is the outer loop that repeats for each row.

RowLp:  mov      Rows, 0     ;Repeat for 0..Dim1-1 rows.
les      di, RowArray    ;Output the current RowArray
mov      bx, Rows        ; value on the left hand side
add      bx, bx          ; of the matrix.
mov      ax, es:[di][bx] ;ES:DI is base, BX is index.
mov      cx, 5            ;Output using five positions.
purisize
print
       char  ": ",0

; ColLp is the inner loop that repeats for each item on each row.

ColLp:  mov      Columns, 0  ;Repeat for 0..Dim2-1 cols.
mov      bx, Rows        ;Compute index into the array
imul     bx, [si].DopeVec.Dim2 ; index := (Rows*Dim2 +
add      bx, Columns     ; columns) * 2
add      bx, bx

; Note that we only have a pointer to the base address of the array, so we
; have to fetch that pointer and index off it to access the desired array
; element. This code loads the pointer to the base address of the array into
; the es:di register pair.

les      di, [si].DopeVec.Data ;Base address of array.
mov      ax, es:[di][bx]     ;Get array element

; The functions that compute the values for the array store an 8000h into
; the array element if some sort of error occurs. Of course, it is possible
; to produce 8000h as an actual result, but giving up a single value to
; trap errors is worthwhile. The following code checks to see if an error
; occurred during the cross product. If so, this code prints "  ****",
; otherwise, it prints the actual value.

cmp      ax, 8000h         ;Check for error value
jne      GoodOutput
print
       char  "  ****",0   ;Print this for errors.
jmp      DoNext

GoodOutput:  mov      cx, 6        ;Use six print positions.
purisize ;Print a good value.

DoNext:  mov      ax, Columns  ;Move on to next array
inc      ax                ; element.
mov      Columns, ax
cmp      ax, [si].DopeVec.Dim2 ;See if we’re done with
jb       ColLp             ; this column.

purcr     ;End each column with CR/LF
mov      ax, Rows          ;Move on to the next row.
ic      ax
mov      Rows, ax
cmp      ax, [si].DopeVec.Dim1 ;Have we finished all the
jb       RowLp             ; rows? Repeat if not done.
ret
PrintMat endp
Chapter 08

PrintSeg ends

cseg segment para public 'code'
assume cs:cseg, ds:dseg

;GetWholeNum- This routine reads a whole number (an integer greater than zero) from the user. If the user enters an illegal whole number, this procedure makes the user re-enter the data.
GetWholeNum proc near
lesi InputLine ;Point es:di at InputLine array.
gets

call Geti ;Get an integer from the line.
jc BadInt ;Carry set if error reading integer.
cmp ax, 0 ;Must have at least one row or column!
jle BadInt
ret

BadInt: print
    char Bell
    char "Illegal integer value, please re-enter",cr,lf,0
    jmp GetWholeNum
GetWholeNum endp

; Various routines to call for the cross products we compute.
; On entry, AX contains the first operand, dx contains the second.
; These routines return their result in AX.
; They return AX=8000h if an error occurs.
;
; Note that the CrossProduct function calls these routines indirectly.
addition proc far
    add ax, dx
    jno AddDone ;Check for signed arithmetic overflow.
    mov ax, 8000h ;Return 8000h if overflow occurs.
AddDone: ret
addition endp

subtraction proc far
    sub ax, dx
    jno SubDone
    mov ax, 8000h ;Return 8000h if overflow occurs.
SubDone: ret
subtraction endp

multiplication proc far
    imul ax, dx
    jno MulDone ;Error if overflow occurs.
    mov ax, 8000h
MulDone: ret
multiplication endp

division proc far
    push cx ;Preserve registers we destroy.
    mov cx, dx
    cwd
    test cx, cx ;See if attempting division by zero.
    je BadDivide
    idiv cx
mov dx, cx ;Restore the munged register.
pop cx
ret
BadDivide: mov ax, 8000h
mov          dx, cx
pop          cx
ret

division     endp

; The following function computes the remainder if the symbol "DoMOD"
; is defined somewhere prior to this point.
ifdef       DoMOD
modulo     proc    far
push        cx
mov          cx, dx
cwd
test        cx, cx   ;See if attempting division by zero.
je           BadDivide
idiv        cx
mov          ax, dx   ;Need to put remainder in AX.
mov          dx, cx   ;Restore the munged registers.
pop          cx
ret

BadMod:     mov        ax, 8000h
mov          dx, cx
pop          cx
ret
modulo      endp
endif

; If you decide to extend the ResultArrays dope vector array, this is a good
; place to define the function for those new arrays.

; The main program that reads the data from the user, calls the appropriate
; routines, and then prints the results.
Main      proc
mov          ax, dseg
mov          ds, ax
mov          es, ax
meminit

; Prompt the user to enter the number of rows and columns:
GetRows:   print
byte      "Enter the number of rows for the matrix:",0
call       GetWholeNum
mov          Rows, ax

; Okay, read each of the row values from the user:

print
char      "Enter values for the row (vertical) array",cr,lf,0

; Malloc allocates the number of bytes specified in the CX register.
; AX contains the number of array elements we want; multiply this value
; by two since we want an array of words. On return from malloc, es:di
; points at the array allocated on the "heap". Save away this pointer in
; the "RowArray" variable.
;
; Note the use of the "wp" symbol. This is an equate to "word ptr" appearing
; in the "matrix.a" include file. Also note the use of the address expression
; "RowArray+2" to access the segment portion of the double word pointer.

mov          cx, ax
shl          cx, 1
malloc
mov          wp RowArray, di
mov wp RowArray+2, es

; Okay, call "GetArray" to read "ax" input values from the user.
; GetArray expects the number of values to read in AX and a pointer
; to the base address of the array in es:di.

print
char "Enter row data:\n",0
mov ax, Rows ;# of values to read.
call GetArray ;ES:DI still points at array.

; Okay, time to repeat this for the column (horizontal) array.

GetCols:
print
byte "Enter the number of columns for the matrix:\n",0
call GetWholeNum ;Get # of columns from the user.
mov Columns, ax ;Save away number of columns.

; Okay, read each of the column values from the user:

print
char "Enter values for the column (horz.) array\n",cr,lf,0

; Malloc allocates the number of bytes specified in the CX register.
; AX contains the number of array elements we want; multiply this value
; by two since we want an array of words. On return from malloc, es:di
; points at the array allocated on the "heap". Save away this pointer in
; the "RowArray" variable.

mov cx, ax ;Convert # Columns to # bytes
shl cx, 1 ; by multiply by two.
malloc ;Get the memory.
mov wp ColArray, di ;Save pointer to the
mov wp ColArray+2, es ;columns vector (array).

; Okay, call "GetArray" to read "ax" input values from the user.
; GetArray expects the number of values to read in AX and a pointer
; to the base address of the array in es:di.

print
char "Enter Column data:\n",0
mov ax, Columns ;# of values to read.
call GetArray ;ES:DI points at column array.

; Okay, initialize the matrices that will hold the cross products.
; Generate RASize copies of the following code.
; The "repeat" macro repeats the statements between the "repeat" and the "endm"
; directives RASize times. Note the use of the Item symbol to automatically
; generate different indexes for each repetition of the following code.
; The "Item = Item+1" statement ensures that Item will take on the values
; 0, 1, 2, ..., RASize on each repetition of this loop.
;
; Remember, the "repeat..endm" macro copies the statements multiple times
; within the source file, it does not execute a "repeat..until" loop at
; run time. That is, the following macro is equivalent to making "RASize"
; copies of the code, substituting different values for Item for each
; copy.
;
; The nice thing about this code is that it automatically generates the
; proper amount of initialization code, regardless of the number of items
; placed in the ResultArrays array.

Item = 0
repeat  RASize
    mov cx, Columns ; Compute the size, in bytes, of the matrix and allocate
    imul cx, Rows ; sufficient storage for the
    add cx, cx ; array.
    mov wp ResultArrays[Item * (sizeof DopeVec)].Data, di
    mov wp ResultArrays[Item * (sizeof DopeVec)].Data+2, es
    mov ax, Rows
    mov ResultArrays[Item * (sizeof DopeVec)].Dim1, ax
    mov ax, Columns
    mov ResultArrays[Item * (sizeof DopeVec)].Dim2, ax
    mov ResultArrays[Item * (sizeof DopeVec)].ESize, 2
    Item = Item+1
endm

; Okay, we’ve got the input values from the user, now let’s compute the addition, subtraction, multiplication, and division tables. Once again, a macro reduces the amount of typing we need to do at this point as well as automatically handling however many items are present in the ResultArrays array.

Element = 0
repeat RASize
    lfs bp, RowArray ; Pointer to row data.
    lgs bx, ColArray ; Pointer to column data.
    lea cx, ResultArrays[element * (sizeof DopeVec)]
    call CrossProduct
    element = element+1
endm

; Okay, print the arrays down here. Once again, note the use of the repeat..endm macro to save typing and automatically handle additions to the ResultArrays array.

Item = 0
repeat RASize
    mov si, offset ResultArrays[item * (sizeof DopeVec)]
    call PrintMat
    Item = Item+1
endm

; Technically, we don’t have to free up the storage malloc’d for each of the arrays since the program is about to quit. However, it’s a good idea to get used to freeing up all your storage when you’re done with it. For example, were you to add code later at the end of this program, you would have that extra memory available to that new code.

les di, ColArray
free
les di, RowArray
free
Item = 0
repeat RASize
    les di, ResultArrays[Item * (sizeof DopeVec)].Data
    free
GETI.ASM contains a routine (geti) that reads an integer value from the user.

; GETI.ASM
; This module contains the integer input routine for the matrix example in Chapter Eight.
.nolist
.include stdlib.a
:list
.include matrix.a

geti proc far
ifdef debug
print char "Inside GETI",cr,lf,0
endif

; First, skip over any leading spaces or commas.
; Note the use of the "byp" symbol to save having to type "byte ptr".
; BYP is a text equate appearing in the macros.a file.
; A "byte ptr" coercion operator is required here because MASM cannot
determine the size of the memory operand (byte, word, dword, etc)
from the operands. I.e., "es:[di]" and " " could be any of these
three sizes.

; Returns carry clear and value in AX if no error.
; Returns carry set if an error occurs.
; This routine leaves ES:DI pointing at the character it fails on when
; converting the string to an integer. If the conversion occurs without
; an error, the ES:DI points at a space, comma, or zero terminating byte.

geti endp
; the loop and then immediately incrementing di, we can increment di before
testing the character in the body of the loop. This makes the loop
slightly more efficient and a lot more elegant.

dec di
SkipSpcs: inc di
cmp byp es:[di], ' '
je SkipSpcs
cmp byp es:[di], ','
jje SkipSpcs

; See if the first non-space/comma character is a decimal digit:

mov al, es:[di]
cmp al, '-' ;Minus sign is also legal in integers.
jne TryDigit
mov al, es:[di+1] ;Get next char, if "-"

TryDigit: isdigit
jne BadGeti ;Jump if not a digit.

; Okay, convert the characters that follow to an integer:

ConvertNum: atoi2 ;Leaves integer in AX
jc BadGeti ;Bomb if illegal conversion.

; Make sure this number ends with a reasonable character (space, comma,
; or a zero byte):

cmp byp es:[di], ' '
jje GoodGeti

cmp byp es:[di], ','
jje GoodGeti
cmp byp es:[di], 0
jje GoodGeti

ifdef debug
print
char "GETI: Failed because number did not end with "
char "a space, comma, or zero byte",cr,lf,0
endif

BadGeti: stc ;Return an error condition.
ret

GoodGeti: clc ;Return no error and an integer in AX
geti ret
endp

InpSeg ends
end

---

8.22.5 GetArray.ASM

GetArray.ASM contains the GetArray input routine. This reads the data for the array
from the user to produce the cross products. Note that GetArray reads the data for a single
dimension array (or one row in a multidimensional array). The cross product program
reads two such vectors: one for the column values and one for the row values in the cross
product. Note: This routine uses subroutines from the UCR Standard Library that appear
in the next chapter.

; GETARRAY.ASM
;
; This module contains the GetArray input routine. This routine reads a
; set of values for a row of some array.
.
.386
option segment:use16

.nolist
include stdlib.a
.list

include matrix.a

; Some local variables for this module:

localdseg segment para public 'LclData'

NumElements word ?
ArrayPtr dword ?
Localdseg ends

InpSeg segment para public 'input'
assume ds:Localdseg

; GetArray-
; Read a set of numbers and store them into an array.
; On Entry:
; es:di points at the base address of the array.
; ax contains the number of elements in the array.
; This routine reads the specified number of array elements
; from the user and stores them into the array. If there
; is an input error of some sort, then this routine makes
; the user reenter the data.

GetArray proc far
pusha ;Preserve all the registers
push ds ; that this code modifies
push es
push fs

ifdef debug
print
char "Inside GetArray, # of input values =",0
puti
putcr
endif

mov cx, Localdseg ;Point ds at our local
mov ds, cx ; data segment.

mov wp ArrayPtr, di ;Save in case we have an
mov wp ArrayPtr+2, es ; error during input.

mov NumElements, ax

; The following loop reads a line of text from the user containing some
; number of integer values. This loop repeats if the user enters an illegal
; value on the input line.
;
; Note: LESI is a macro from the stdlib.a include file. It loads ES:DI
; with the address of its operand (as opposed to les di, InputLine that would
; load ES:DI with the dword value at address InputLine).

RetryLp:
lesi InputLine ;Read input line from user.
gets
mov cx, NumElements ;# of values to read.
lfs si, ArrayPtr ;Store input values here.

; This inner loop reads "ax" integers from the input line. If there is
; an error, it transfers control to RetryLp above.

ReadEachItem: call geti ;Read next available value.
jc       BadGA
mov      fs:[si], ax    ;Save away in array.
add      si, 2         ;Move on to next element.
loop     ReadEachItem  ;Repeat for each element.
pop      fs             ;Restore the saved registers
pop      es             ; from the stack before
pop      ds             ; returning.
popa
ret

; If an error occurs, make the user re-enter the data for the entire
; row:
BadGA:          print
char        "Illegal integer value(s).",cr,lf
char        "Re-enter data:",0
jmp        RetryLp
getArray    endp

InpSeg    ends
end

8.22.6 XProduct.ASM

This file contains the code that computes the actual cross-product.

; XProduct.ASM-
;
;   This file contains the cross-product module.

   .386
   option   segment:use16

   .nolist
   include  stdlib.a
   includelib stdlib.lib
   .list
   include  matrix.a

; Local variables for this module.

dseg      segment para public 'data'
DV        dword   ?
RowNdx    integer  ?
ColNdx    integer  ?
RowCntr   integer  ?
ColCntr   integer  ?
dseg      ends

cseg      segment para public 'code'
assume   ds:dseg

; CrossProduct- Computes the cartesian product of two vectors.
;
; On entry:
;
;     FS:BP-Points at the row matrix.
;     GS:BX-Points at the column matrix.
;     DS:CX-Points at the dope vector for the destination.
;
; This code assume ds points at dseg.
; This routine only preserves the segment registers.

RowMat    textequ <fs:[bp]>
ColMat    textequ <gs:[bx]>
DVP textequ <ds:[bx].DopeVec>

CrossProduct proc near

ifdef debug
print char "Entering CrossProduct routine",cr,lf,0
endif

xchg bx, cx ;Get dope vector pointer
mov ax, DVP.Dim1 ;Put Dim1 and Dim2 values
mov RowCntr, ax ; where they are easy to access.
mov ax, DVP.Dim2
mov ColCntr, ax
xchg bx, cx

; Okay, do the cross product operation. This is defined as follows:
;
; for RowNdx := 0 to NumRows-1 do
; for ColNdx := 0 to NumCols-1 do
; Result[RowNdx, ColNdx] = Row[RowNdx] op Col[ColNdx];

OutsideLp: add RowNdx, 1
mov ax, RowNdx
cmp ax, RowCntr
jge Done

InsideLp: add ColNdx, 1
mov ax, ColNdx
cmp ax, ColCntr
jge OutSideLp

mov di, RowNdx
add di, di
mov ax, RowMat[di]

mov di, ColNdx
add di, di
mov dx, ColMat[di]
push bx ;Save pointer to column matrix.
mov bx, cx ;Put ptr to dope vector where we can
; use it.
call DVP.Func ;Compute result for this guy.
mov di, RowNdx ;Index into array is
imul di, DVP.Dim2 ; (RowNdx*Dim2 + ColNdx) * ElementSize
add di, ColNdx
imul di, DVP.ESize
les bx, DVP.Data ;Get base address of array.
mov es:[bx][di], ax ;Save away result.
pop bx ;Restore ptr to column array.
jmp InsideLp
 Done: ret
CrossProduct endp

cseg ends
end
8.23 Laboratory Exercises

In this set of laboratory exercises you will assemble various short programs, produce assembly listings, and observe the object code the assembler produces for some simple instruction sequences. You will also experiment with a make file to observe how it properly handles dependencies.

8.23.1 Near vs. Far Procedures

The following short program demonstrates how MASM automatically generates near and far call and ret instructions depending on the operand field of the proc directive (this program is on the companion CD-ROM in the chapter eight subdirectory).

Assemble this program with the /Fl option to produce an assembly listing. Look up the opcodes for near and far call and ret instructions in Appendix D. Compare those values against the opcodes this program emits. For your lab report: describe how MASM figures out which instructions need to be near or far. Include the assembled listing with your report and identify which instructions are near or far calls and returns.

; EX8_1.asm (Laboratory Exercise 8.1)

cseg segment para public 'code'
assume cs:cseg, ds:dseg

Procedure1 proc near
; MASM will emit a *far* call to procedure2
; since it is a far procedure.
call Procedure2
;
; Since this return instruction is inside
; a near procedure, MASM will emit a near
; return.
ret

Procedure1 endp

Procedure2 proc far
; MASM will emit a *near* call to procedure1
; since it is a near procedure.
call Procedure1
;
; Since this return instruction is inside
; a far procedure, MASM will emit a far
; return.
ret

Procedure2 endp

Main proc
mov ax, dseg
mov ds, ax
mov es, ax
;
; MASM emits the appropriate call instructions
; to the following procedures.
call Procedure1
call Procedure2

Quit: mov ah, 4ch
8.23.2 Data Alignment Exercises

In this exercise you will compile two different programs using the MASM “/Fl” command line option so you can observe the addresses MASM assigns to the variables in the program. The first program (Ex8_2a.asm) uses the `even` directive to align objects on a word boundary. The second program (Ex8_2b.asm) uses the `align` directive to align objects on different sized boundaries. **For your lab report:** Include the assembly listings in your lab report. Describe what the `even` and `align` directives are doing in the program and comment on how this produces faster running programs.

; EX8_2a.asm
;
; Example demonstrating the EVEN directive.

; dseg segment
; Force an odd location counter within
; this segment:

i byte 0
; This word is at an odd address, which is bad!

j word 0
; Force the next word to align itself on an
; even address so we get faster access to it.

k word 0
; Note that even has no effect if we're already
; at an even address.

l word 0

; The following instruction would normally lie on
; an odd address. The EVEN directive inserts a
; NOP so that it falls on an even address.

moves bx, cx
; Since we're already at an even address, the
; following EVEN directive has no effect.
8.23.3 Equate Exercise

In this exercise you will discover a major difference between a numeric equate and a textual equate (program Ex8_3.asm on the companion CD-ROM). MASM evaluates the operand field of a numeric equate when it encounters the equate. MASM defers evaluation of a textual equate until it expands the equate (i.e., when you use the equate in a program). For your lab report: assemble the following program using MASM’s “/Fl” command line option and look at the object code emitted for the two equates. Explain
why the instruction operands are different even though the two equates are nearly identical.

; Ex8_3.asm
;
; Comparison of numeric equates with textual equates
; and the differences they produce at assembly time.
;
cseg segment
equ1 equ $+2 ; Evaluates "$" at this stmt.
equ2 equ <$+2> ; Evaluates "$" on use.
MyProc proc
    mov ax, 0
    lea bx, equ1
    lea bx, equ2
    lea bx, equ1
    lea bx, equ2
MyProc endp

cseg ends
end

8.23.4 IFDEF Exercise

In this exercise, you will assemble a program that uses conditional assembly and observe the results. The Ex8_4.asm program uses the #ifdef directive to test for the presence of DEBUG1 and DEBUG2 symbols. DEBUG1 appears in this program while DEBUG2 does not. **For your lab report:** assemble this code using the ‘/Fl’ command line parameter. Include the listing in your lab report and explain the actions of the #ifdef directives.

; Ex8_4.asm
;
; Demonstration of IFDEF to control
; debugging features. This code
; assumes there are two levels of
; debugging controlled by the two
; symbols DEBUG1 and DEBUG2. In
; this code example DEBUG1 is
; defined while DEBUG2 is not.

.xlist
.include stdlib.a
.list
.nolistmacro
 список
.listif

DEBUG1 = 0

cseg segment
DummyProc proc
    ifdef DEBUG2
print
byte  "In DummyProc"
byte cr,lf,0
endif
ret
DummyProc endp

Main proc
    ifdef DEBUG1
print
byte  "Calling DummyProc"
byte cr,lf,0
endif

call DummyProc
 ifdef DEBUG1

8.23.5 Make File Exercise

In this exercise you will experiment with a make file to see how nmake.exe chooses which files to reassemble. In this exercise you will be using the Ex8_5a.asm, Ex8_5b.asm, Ex8_5a, and Ex8_5.mak files found in the Chapter Eight subdirectory on the companion CD-ROM. Copy these files to a local subdirectory on your hard disk (if they are not already there). These files contain a program that reads a string of text from the user and prints out any vowels in the input string. You will make minor changes to the .asm and .a files and run the make file and observe the results.

The first thing you should do is assemble the program and create up to date .exe and .obj files for the project. You can do this with the following DOS command:

```
nmake Ex8_5.mak
```

Assuming that the .obj and .exe files were not already present in the current directory, the nmake command above will assemble and link the Ex8_5a.asm and Ex8_5b.asm files producing the Ex8.exe executable.

Using the editor, make a minor change (such as inserting a single space on a line containing a comment) to the Ex8_5a.asm file. Execute the above nmake command. Record what the make file does in your lab report.

Next, make a minor change to the Ex8_5b.asm file. Run the above nmake command and record the result in your lab report. Explain the results.

Finally, make a minor change to the Ex8_5.a file. Run the nmake command and describe the results in your lab report.

For your lab report: explain how the changes to each of the files above affects the make operation. Explain why nmake does what it does. For additional credit: Try deleting (one at a time) the Ex8_5a.obj, Ex8_5b.obj, and Ex8_5.exe files and run the nmake command. Explain why nmake does what it does when you individually delete each of these files.

Ex8_5.mak makefile:

```
ex8_5.exe: ex8_5a.obj ex8_5b.obj
  ml /Feex8_5.exe ex8_5a.obj ex8_5b.obj

ex8_5a.obj: ex8_5a.asm ex8_5.a
  ml /c ex8_5a.asm

ex8_5b.obj: ex8_5b.asm ex8_5.a
  ml /c ex8_5b.asm
```

Ex8_5.a Header File:

```
; Header file for Ex8_5 project.
; This file includes the EXTERNDEF
; directive which makes the PrintVowels
; name public/external. It also includes
; the PrtVowels macro which lets us call
; the PrintVowels routine in a manner
; similar to the UCR Standard Library
```
externdef PrintVowels:near

PrtVowels macro
call PrintVowels
endm

Ex8_5a.asm source file:
; Ex8_5a.asm
;
; Randall Hyde
; 2/7/96
;
; This program reads a string of symbols from the
; user and prints the vowels. It demonstrates the use of
; make files

.xlist
include stdlib.a
includelib stdlib.lib
.list

; The following include file brings in the external
; definitions of the routine(s) in the Lab6x10b
; module. In particular, it gives this module
; access to the "PrtVowels" routine found in
; Lab8_5b.asm.

include Ex8_5.a

cseg segment para public 'code'
Main proc
meminit

; Read a string from the user, print all the vowels
; present in that string, and then free up the memory
; allocated by the GETSM routine:

print
byte "I will find all your vowels"
byte cr,lf
byte "Enter a line of text: ",0
getsm
print
byte "Vowels on input line: ",0
PrtVowels
putcr
free

Quit: ExitPgm
Main endp

cseg ends

sseg segment para stack 'stack'
stk byte 1024 dup ("stack ")
sseg ends

zzzzzsec segment para public 'zzzzzz'
LastBytes byte 16 dup (?)
8.24 Programming Projects

1) Write a program that inputs two 4x4 integer matrices from the user and compute their matrix product. The matrix multiply algorithm (computing $C := A \times B$) is

   ```
   for i := 0 to 3 do
     for j := 0 to 3 do begin
       c[i,j] := 0;
       for k := 0 to 3 do
         c[i,j] := c[i,j] + a[i,k] * b[k,j];
     end;
   
   ```

Feel free to use the ForLp and Next macros from Chapter Six.

2) Modify the sample program (“Sample Program” on page 432) to use the FORLP and NEXT macros provided in the textbook. Replace all for loop simulations in the program with the corresponding macros.

3) Write a program that asks the user to input three integer values, $m$, $p$, and $n$. This program should allocate storage for three arrays: $A[0..m-1, 0..p-1]$, $B[0..p-1, 0..n-1]$, and $C[0..m-1, 0..n-1]$. The program should then read values for arrays $A$ and $B$ from the user. Next, this program should compute the matrix product of $A$ and $B$ using the algorithm:

   ```
   for i := 0 to m-1 do
     for j := 0 to n-1 do begin
       c[i,j] := 0;
       for k := 0 to p-1 do
         c[i,j] := c[i,j] + a[i,k] * b[k,j];
     end;
   
   ```

Finally, the program should print arrays $A$, $B$, and $C$. Feel free to use the ForLp and Next macro given in this chapter. You should also take a look at the sample program (see “Sample Program” on page 432) to see how to dynamically allocate storage for arrays and access arrays whose dimensions are not known until run time.

4) The ForLp and Next macros provide in this chapter only increment their loop control variable by one on each iteration of the loop. Write a new macro, ForTo, that lets you specify an increment constant. Increment the loop control variable by this constant on each iteration of the for loop. Write a program to demonstrate the use of this macro. Hint: you will need to create a global label to pass the increment information to the NEXT macro, or you will need to perform the increment operation inside the ForLp macro.

5) Write a third version for ForLp and Next (see Program #7 above) that lets you specify negative increments (like the for..downto statement in Pascal). Call this macro ForDT (for..downto).

8.25 Summary

This chapter introduced several assembler directives and pseudo-opcodes supported by MASM. This chapter, by no means, is a complete description of what MASM has to offer. It does provide enough information to get you going.

Assembly language statements are free format and there is usually one statement per line in your source file. Although MASM allows free format input, you should carefully structure your source files to make them easier to read.

- See “Assembly Language Statements” on page 355.
MASM keeps track of the offset of an instruction or variable in a segment using the location counter. MASM increments the location counter by one for each byte of object code it writes to the output file.

- See “The Location Counter” on page 357.

Like HLLs, MASM lets you use symbolic names for variables and statement labels. Dealing with symbols is much easier than numeric offsets in an assembly language program. MASM symbols look a whole lot like their HLL with a few extensions.

- See “Symbols” on page 358

MASM provides several different types of literal constants including binary, decimal, and hexadecimal integer constants, string constants, and text constants.

- See “Literal Constants” on page 359.
- See “Integer Constants” on page 360.
- See “String Constants” on page 361.
- See “Text Constants” on page 362.

To help you manipulate segments within your program, MASM provides the `segment/ends` directives. With the segment directive you can control the loading order and alignment of modules in memory.

- See “Segments” on page 366.
- See “Segment Names” on page 367.
- See “Segment Loading Order” on page 368.
- See “Segment Operands” on page 369.
- See “The CLASS Type” on page 374.
- See “Typical Segment Definitions” on page 376.
- See “Why You Would Want to Control the Loading Order” on page 376.

MASM provides the `proc/endp` directives for declaring procedures within your assembly language programs. Although not strictly necessary, the `proc/endp` directives make your programs much easier to read and maintain. The `proc/endp` directives also let you use local statement names within your procedures.

- See “Procedures” on page 365.

Equates let you define symbolic constants of various sorts in your program. MASM provides three directives for defining such constants: “=”, `equ`, and `textequ`. As with HLLs, the judicious use of equates can help make your program easier to read.

- See “Declaring Manifest Constants Using Equates” on page 362.

As you saw in Chapter Four, MASM gives you the ability to declare variables in the data segment using the `byte`, `word`, `dword` and other directives. MASM is a strongly typed assembler and attaches a type as well as a location to variable names (most assemblers only attach a location). This helps MASM locate obscure bugs in your program.

- See “Variables” on page 384.
- See “Label Types” on page 385.
- See “How to Give a Symbol a Particular Type” on page 385.
- See “Label Values” on page 386.
- See “Type Conflicts” on page 386.

MASM supports `address expressions` that let you use arithmetic operators to build constant address values at assembly time. It also lets you override the type of an address value and extract various pieces of information about a symbol. This is very useful for writing maintainable programs.

- See “Address Expressions” on page 387.
- See “Symbol Types and Addressing Modes” on page 387.
- See “Arithmetic and Logical Operators” on page 388.
- See “Coercion” on page 390.
- See “Type Operators” on page 392.
MASM provides several facilities for telling the assembler which segment associates with which segment register. It also gives you the ability to override a default choice. This lets your program manage several segments at once with a minimum of fuss.

- See “Segment Prefixes” on page 377.
- See “Controlling Segments with the ASSUME Directive” on page 377.

MASM provides you with a “conditional assembly” capability that lets you choose which segments of code are actually assembled during the assembly process. This is useful for inserting debugging code into your programs (that you can easily remove with a single statement) and for writing programs that need to run in different environments (by inserting and removing different sections of code).

- See “Conditional Assembly” on page 397.
- See “IF Directive” on page 398.
- See “IFE directive” on page 399.
- See “IFDEF and IFNDEF” on page 399.
- See “IFB, IFNB” on page 399.
- See “IFIDN, IFDIF, IFIDI, and IFDIFI” on page 400.

MASM, living up to its name, provides a powerful macro facility. Macros are sections of code you can replicate by simply placing the macro’s name in your code. Macros, properly used, can help you write shorter, easier to read, and more robust programs. Alas, improperly used, macros produce hard to maintain, inefficient programs.

- See “Macros” on page 400.
- See “Procedural Macros” on page 400.
- See “Macros: Good and Bad News” on page 419.
- See “Repeat Operations” on page 420.

MASM provides several directives you can use to produce “assembled listings” or print-outs of your program with lots of assembler generated (useful!) information. These directives let you turn on and off the listing operation, display information on the display during assembly, and set titles on the output.

- See “Controlling the Listing” on page 424.
- See “The ECHO and %OUT Directives” on page 424.
- See “The .LIST, .NOLIST, and .XLIST Directives” on page 425.
- See “Other Listing Directives” on page 425.

To handle large projects (“Programming in the Large”) requires separate compilation (or separate assembly in MASM’s case). MASM provides several directives that let you merge source files during assembly, separately assemble modules, and communicate procedure and variables names between the modules.

- See “Managing Large Programs” on page 425.
- See “The PUBLIC, EXTERN, and EXTRN Directives” on page 427.
8.26 Questions

1) What is the difference between the following instruction sequences?

\[
\begin{align*}
\text{MOV} & \quad AX, \text{VAR+1} \\
\text{and} & \quad \text{MOV} \quad AX, \text{VAR} \\
\text{INC} & \quad \text{AX}
\end{align*}
\]

2) What is the source line format for an assembly language statement?

3) What is the purpose of the ASSUME directive?

4) What is the location counter?

5) Which of the following symbols are valid?

   a) ThisIsASymbol  b) This_Is_A_Symbol
   c) This.Is.A.Symbol  d) .Is_This_A_Symbol?
   e) ________________  f) @$_?_To_You
   g) 1WayToGo  g) %Hello
   i) F000h  j) ?A_0$1
   k) $1234  l) Hello there

6) How do you specify segment loading order?

7) What is the type of the symbols declared by the following statements?

   a) symbol1 equ 0
   b) symbol2:
   c) symbol3 proc
   d) symbol4 db ?
   e) symbol5 dw ?
   f) symbol6 proc far
   g) symbol7 equ this word
   h) symbol8 equ byte ptr symbol7
   i) symbol9 dd ?
   j) symbol10 macro
   k) symbol11 segment para public 'data'
   l) symbol12 equ this near
   m) symbol13 equ 'ABCD'
   n) symbol14 equ <MOV AX, 0>

8) Which of the symbols in question 7 are not assigned the current location counter value?

9) Explain the purpose of the following operators:

   a) PTR  b) SHORT  c) THIS  d) HIGH  e) LOW
   f) SEG  g) OFFSET

10) What is the difference between the values loaded into the BX register (if any) in the following code sequence?

\[
\begin{align*}
\text{mov} & \quad \text{bx, offset Table} \\
\text{lea} & \quad \text{bx, Table}
\end{align*}
\]

11) What is the difference between the REPEAT macro and the DUP operator?

12) In what order will the following segments be loaded into memory?

\[
\begin{align*}
\text{CSEG} & \quad \text{segment para public 'CODE'} \\
\text{DSEG} & \quad \text{segment para public 'DATA'} \\
\text{ESEG} & \quad \text{segment para public 'CODE'}
\end{align*}
\]
13) Which of the following address expressions do not produce the same result as the others:
   e) Var1*3*5  f) Var1+3+5