Directives & Pseudo-Opcodes — Lab Manual, Chapter Seven

To write assembly language programs you need to know just a little more than the language of the microprocessor. The assembler has its own language above and beyond machine instructions. These additional statements, the assembler directives and pseudo-opcodes, let you create symbolic names for objects, perform assembly time computations, and help you write portable applications. This chapter discusses many of the advanced features provided by MASM 6.x and how you can use them to ease the assembly language programming process.

Writing in pure assembly language isn't much fun. Seemingly simple tasks, like writing the famous "Hello world" program take considerable effort in assembly language. Far more than you would like if you're used to high level languages like Pascal and C. A simple print statement in pure assembly language could take hundreds, or even thousands, of lines of assembly code. Although DOS and BIOS simplify this somewhat, it's still quite a bit more work than using the WRITELN in Pascal. The UCR Standard Library for 80x86 Assembly Language Programmers was developed at the University of California, Riverside, to explicitly reduce the pain of transition from a HLL to assembly. The UCR StdLib provides many high level functions comparable to those found in the C programming languages. Even if you are not familiar with the C programming language, you will find the UCR Standard Library easy to learn and much easier to use than pure assembly language for most programming tasks. Since this chapter presents the last of the tools necessary for you to start writing full featured assembly language programs, it's a great place to introduce you to the UCR Standard Library so you won't suffer too much frustration when writing your assembly language programs.

7.1 Assembly Language Statements

MASM generally expects one assembly language source statement per line of source code. Each assembly language statement consists of one to four *fields:* the *label* field, the *mnemonic* field, the *operand* field, and the *comment* field. Each field is optional. In fact, MASM allows completely blank lines when you leave out all four fields. How you organize these fields in your source code is, perhaps, the primary factor controlling the readability of your code.

MASM is a *free-form* assembler. This means that you do not have to place the fields in a source statement in specific columns[1]. In general, as long as the label field (if present) is the first field on the line, the mnemonic is the second, the operand is third, and the comment field is last, MASM is happy. So a correct MASM statement takes the form:

```
Label         mnemonic  operand     ;comment
```

The amount of white space before and after each field is insignificant to MASM. Consider the following examples:

7.1 What are the four fields of an assembly language statement?

7.2 Which fields are optional in an assembly language statement?

7.3 Why do we use a fixed format source statement when MASM allows free-format statements?

1. Some older assemblers require each field to begin in a specific column. Very few modern assemblers require this.

7.2 The Location Counter

The assembler uses an internal variable, the *location counter* to keep track of the current offset into a segment. The location counter corresponds to the 80x86's *instruction pointer* (IP) register. For simple assembly language programs, the location counter value MASM associates with a statement is the same value the IP register will contain when the CPU executes that instruction. MASM uses the location counter to convert symbolic names into numeric offsets and to determine the position of code within your programs. Since understanding the effects of the location counter on your program can make a difference in the performance and even the correctness of your programs, you should know what the location counter is and how MASM uses it.

Whenever you begin a new segment within a program MASM automatically associates a location counter value with that segment and initializes the location counter to zero. As the assembler emits instructions to the object code file it associates the current location counter value with each instruction. Therefore, the first instruction in a new segment will have the location counter value zero associated with it. As the assembler processes 80x86 machine instructions and MASM pseudo-opcodes MASM increases the value of the location counter by the length of each instruction it processes. So if the first instruction in a segment is two bytes long the location counter value associated with the next instruction is two.

7.4 What CPU register most closely corresponds to the location counter?

____

7.5 If the first instruction in a segment is two bytes long and the second instruction is three bytes long, what is the value of the location counter at the beginning of the third instruction?

____

If you use the "$" symbol within an expression, MASM substitutes the current location counter value *at the beginning of the instruction, before emitting any code,* for the "$" symbol within the expression. For example, the following MOV instruction loads AX with the offset of the MOV instruction:

```
mov       ax, offset $
```

7.6 Given that a short JMP instruction is two bytes long, what will the instruction "JMP $+2" do in your program?

____

____

____

If you make an assembly listing (see Laboratory Exercise #1) you can see the value of the location counter for each instruction in your program. ML created the following example listing file[2]:

```
                  ; Demonstration of location counter values
                  ; in an assembly listing. Assemble this
                  ; code with the /Fl command line option.

0000              cseg       segment
0000              MyProc     proc
0000 50                      push     ax
0001 B0 00                   mov      al, 0
0003 B8 0000                 mov      ax, 0
0006 8B D8                   mov      bx, ax
0008 8B 87 1234              mov      ax, 1234h[bx]
```

2. Most of the assembled listings appearing in this manual have been edited to remove unnecessary information and to format the listing so that it fits properly on these pages. Actual assembly listings produced by the ML program may be slightly different.

```
000C EB 00                jmp      $+2
000E 58                   pop      ax
000F C3                   ret
0010           MyProc     endp
0010           cseg       ends
                          end
```

6.258 Label, mnemonic, operand, comment

The first column is the location counter for the current segment. The next set of hexadecimal numbers are the object code bytes emitted for that instruction. Individual bytes are output to the code stream to successive addresses in memory. The “MOV AL, 0” instruction above, for example, outputs the value B0h to location 0001 and 00h to location 0002. If a word value appears in the output list (i.e., a four digit hexadecimal value) then MASM outputs the L.O. byte first and H.O. byte second according to the 80x86’ little endian organization. For example, the “MOV ax, 1234h[bx]” instruction above outputs 8Bh to location 0008, 87h to location 0009, 34h to location 000Ah, and 12h to location 000Bh.

6.259 All fields are optional.

7.7 What is the opcode for the “PUSH AX” instruction above?

7.8 How many bytes long is the “jmp $+2” instruction above?

The value of the location counter can make a difference in the execution time of your programs. The 80x86 CPUs, when fetching instruction opcodes from memory, always fetch one, two, four, or eight bytes depending on the size of the processor. So, for example, if you are using a 64-bit Pentium processor and you jump to an instruction whose location counter valueis one less than an even multiple of eight, the CPU will spend one memory cycle fetching a single byte. It will need to spend a second memory cycle fetching the second byte of that instruction. If your code had jumped to an address that was an even multiple of eight bytes, the first memory cycle would have fetched eight bytes. Therefore, executing the first instruction (assuming it is longer than one byte) requires only one memory access rather than two.

6.260 To make programs easier to read.

The even directive adjusts the location counter value so that it contains an even value. If the location counter value is already even, the even directive leaves it alone. If the location counter value is odd, even emits a zero byte to the current segment if it is a data segment, it emits no-operation instructions if it is a code segment. The even directive is great for aligning data on an even byte (word) boundary. As such, you can use it to align branch targets on 8086, 80186, 80286, and 80386sx processors (which are all 16-bit processors). The following listing shows how the even directive operates:

```
                     ; Example demonstrating the EVEN directive.

0000                 dseg          segment

                     ; Force an odd location counter within
                     ; this segment:

0000 00              i             byte      0

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

0001 0000            j             word      0

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

                                   even
0004 0000            k             word      0

                     ; Note that even has no effect if we’re
```

```
                      ;  already at an even address.

                                     even
0006 0000             l              word      0
0008                  dseg           ends

0000                  cseg           segment
                                     assume    ds:dseg
0000                  procedure      proc
0000 8B 07                           mov       ax, [bx]
0002 A2 0000 R                       mov       i, al
0005 8B D8                           mov       bx, ax

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

                                     even
0008 8B D9                           mov       bx, cx

                      ; Since we're already at an even address,
                      ; the following EVEN directive has no
                      ; effect.

                                     even
000A 8B D0                           mov       dx, ax
000C C3                              ret
000D                  procedure      endp
000D                  cseg           ends
                                     end
```

7.9 What value does MASM insert before the "k" variable in the data segment above?

____

7.10 MASM will need to insert a byte before "MOV BX, CX" instruction above. What 80x86 instruction does this byte correspond to?

____

Unfortunately, `even` doesn't solve the alignment problems on 32 and 64 bit processors. Fortunately, MASM provides a second directive, `align`, that lets you adjust the location counter value so it is an even multiple of any power of two. The `align` directive uses the syntax:

align *expression*

The value of *expression* must be a power of two (e.g., 2, 4, 8, 16).

Like the `even` directive, `align` emits zeros or no-operation instructions to fill up any vacant space Since `align` lets you choose values that correspond to processor sizes and cache line sizes, you can easily align your code no matter which processor you're using.

7.11 If you want your code to be aligned optimally to produce the fastest code for all members of the 80x86 family, what operand would you use for the ALIGN directive? Why?

____

____

____

____

Some of the best places to put align directives are just before a procedure, just before the target of a `jmp` that is not part of a sequence, and at the beginning of a loop that will execute many times.

6.261 IP

7.12 Although the ALIGN directive will output NOPs to your code segment, it is not a particularly good idea to insert ALIGNs between arbitrary assembly instructions. Why? (Hint: what happens when the CPU executes the code in this "empty" space?)

6.262 5

____

____

____

The following listing demonstrates the use of the align directive:

6.263 Fall through to the following instruction.

```
                    ; Example demonstrating the align
                    ; directive.

0000                dseg           segment

                    ; Force an odd location counter
                    ; within this segment:

0000 00             i              byte      0

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

0001 0000           j              word      0

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

                                   align     2
0004 0000           k              word      0

                    ; Force odd address again:

0006 00             k_odd          byte      0

                    ; Align the next entry on a double
                    ; word boundary.

                                   align     4
0008 00000000       l              dword     0

                    ; Align the next entry on a quad
                    ; word boundary:

                                   align     8
0010                RealVar        real8     3.14159
400921F9F01B866E

                    ; Start the following on a paragraph
                    ; boundary:

                                   align     16
0020 00000001 00000002
                    Table          dword     1,2,3,4,5
00000003 00000004
00000005
0034                dseg           ends
                                   end
```

6.264 50h

6.265 2

The align directive has one important limitation: it cannot align data to a block any larger than the alignment specified in the segment directive. Since the segment directive supports byte, word, dword, para, and page alignment options, the maximum operand for align is going to be 256 (page alignment). The allowable operands, therefore, are as follows:

- Align and even are illegal if the segment alignment is byte,
- Even and align 2 is legal if the segment alignment is word,
- Even, align 2,and align 4 are legal if the segment alignment is double word,
- Even, align 2, align4, align 8, and align 16 are legal if the segment alignment is paragraph, and
- Even and align with operands 2, 4, 8, 16, 32, 64, 128, and 256 are legal if the segment alignment is page.

7.13 Given that cache lines are 16 bytes on the 80486, what would be a good operand to use for the ALIGN directive before each of the procedures in your program?

____

7.3 Symbols

One of the primary benefits to an assembler like MASM is that it lets you use symbolic names in place of numeric values. Although MASM allows symbolic names to take many different forms, your symbols should always take the following form:

- The symbol should begin with an alphabetic character. When interfacing with the C programming language, you may need to begin certain symbols with an underscore as well. You should not begin symbols with an underscore unless you need to make that symbol available to a C program.
- After the first character, a symbol may contain alphabetic characters, numeric characters, and underscore characters.
- MASM allows any number of symbols in an identifier. Only the first 31 are significant, however. If two unique symbols contain the same characters up to the 32nd character, MASM thinks they are the same symbol.
- In general, MASM symbols are not case sensitive. However, if you are interfacing your code to the C programming language, you may need to use the option directive or a command line parameter to specify case sensitivity. The following option operands let you specify case sensitivity:

```
option    CASEMAP:NONE                ;Symbols are case sensitive
option    CASEMAP:NOTPUBLIC           ;Public symbols are case
                                      ; sensitive, locals are not.
option    CASEMAP:ALL                 ;Symbols are case insensitive.
```

Case sensitivity is a touchy issue with many programmers. Some (very) strongly believe that it's a good idea to not only have case sensitivity, but to use it wherever possible as well. Others feel that if you cannot tell the difference between two identifiers when they are spoken, the programming language shouldn't differentiate them either. This text adopts the pragmatic approach of using totally unique symbols for different objects and making sure that the case is proper for each usage. In general, this is a good policy to adopt since you will be able to interface with high level languages yet avoid the confusion that occurs when you have two symbols whose only difference is alphabetic case. The "option casemap:notpublic" directive is probably the best choice for all around assembly language programs.

There are other restrictions on symbols in your assembly language programs. For example, you cannot use one of MASM's reserved words as a symbol. See the textbook or the MASM reference manual for a list of MASM's reserved words.

7.14 Taking advantage of case insensitivity in a program is generally a bad idea. Why does MASM even need to support this?

____

____

Symbols in an assembly language program have two major attributes associated with them: a *value* and a *type*. The allowable types include byte, word, dword, qword, tbyte, real4, real8, real10, near, far, text, segment, abs (absolute or constant), and other types. A symbol's declaration determines its type. Statement labels (those followed by one or two colors) are always *near* symbols. Near procedure names are also near symbols. Likewise, far procedure names are always far typed symbols. Variables you declare with the data definition directives (db, byte, sword, real4, dq, etc.) all take on their respective types. Symbols declared with textequ are textual symbols, and other constants declared with equ or "=" and an literal constant operand are of type absolute. Symbols appearing in a segment directive are symbols of type segment[3] Whenever you create an assembly listing, MASM prints out a *symbol table* at the end of the listing. This symbol table provides the type and value information for each symbol in the program. The following example shows you what the symbol table looks like:

6.266 0

6.267 NOP

6.268 ALIGN 4 because aligning on double word boundaries is best for 80386 and later processors. Although not necessary for 80286 and before, you still get the best performance.

6.269 You have execute all those NOPs.

```
                      ; Program with symbols of various types.

0000                  dseg          segment
0000 00               i             byte      0
0001 0000             j             word      0
0003 00000000         k             dword     0
0007                  l             qword     0
0000000000000000
000F                  dseg          ends

0000                  cseg          segment
0000                  MyProc        proc      near
0000 90                             nop
0001 90               MyLbl:        nop
0002 90               MyLbl2::      nop
0003 C3                             ret
0004                  MyProc        endp

0004                  FarProc       proc      far
0004 90                             nop
0005 CB                             ret
0006                  FarProc       endp
0006                  cseg          ends

= 0001                Value1        =         1
= 0002                Value2        =         2
= 0002                Value3        equ       2
= 2                   Value4        equ       <2>
= 2                   Value5        textequ   <2>
                                    end

Segments and Groups:

N a m e           Size   Length     Align      Combine       Class

cseg . . . .      16 Bit 0006        Para       Private
dseg . . . .      16 Bit 000F        Para       Private

Procedures, parameters and locals:

N a m e                             Type       Value         Attr

FarProc . . . . . . . . . . . .P Far            0004          cseg
                                               Length= 0002 Public
MyProc . . . . . . . . . . . . . P Near         0000          cseg
                                               Length= 0004 Public
MyLbl . . . . . . . . . . . . . L Near          0001          cseg

Symbols:
```

3. Symbols appearing in a group directive are also symbols of type segment.

```
N a m e                            Type       Value         Attr

MyLbl2 . . . . . . . . . . . . . .LNear         0002          cseg
Value1 . . . . . . . . . . . . . .Number        0001h
Value2 . . . . . . . . . . . . . .Number        0002h
Value3 . . . . . . . . . . . . . .Number        0002h
Value4 . . . . . . . . . . . . . .Text          2
Value5 . . . . . . . . . . . . . .Text          2
i . . . . . . . . . . . . . . . . Byte          0000          dseg
j . . . . . . . . . . . . . . . . Word          0001          dseg
k . . . . . . . . . . . . . . . . DWord         0003          dseg
l . . . . . . . . . . . . . . . . QWord         0007          dseg
```

(The "Attr" field above is just the segment address in this example.)

7.15 What is the type of symbol Value5 above?

____

7.16 Which type above corresponds to "absolute"

____

7.17 What is the offset of MyLbl2 in the above example?

____

The value of a symbol is usually the value of the location counter and segment address at the beginning of the statement on which the symbol lies. Textual, macro, segment, and absolute typed symbols are the obvious exceptions. The value of a textual symbol is simply the text in the operand field of the `textequ` statement. The value of a segment symbol is the paragraph address of the corresponding segment. The value of an absolute symbol is whatever value appears in the operand field of the `equ` directive.

If a symbol's type is not segment, textual, or absolute, then the value associated with that symbol consists of the two components of a segmented address: the offset and the segment portion. You can use the `offset` and `seg` operators to extract these two values, e.g.,

```
mov     di, seg MySymbol
mov     es, di
mov     di, offset MySymbol
```

Note that the `seg` and `offset` operators always return a constant (abs type). Therefore, the first and third instructions above always use the immediate addressing mode.

7.18 Why can't you execute an instruction of the form:

MOV ES, seg MySymbol

____

____

____

____

If a segment name appears in the operand field of an instruction, MASM automatically returns a constant corresponding to the segment's paragraph address. If `cseg` is a symbol of type segment, the following two statements are legal and produce exactly the same results:

```
mov     ax, seg cseg
mov     ax, cseg
```

MASM is a strongly typed assembler. That is, as much as possible it insists that the types of operands to an instruction agree. For example, the following instruction is illegal because it mixes eight and sixteen bit operands in the same instruction: 6.270 6.15: ALIGN 16

```
            mov     bl, ax
```

Similarly, if MyVar is of type word (perhaps you've declared it using the word directive), then the following is also illegal because the operand sizes do not match:

```
            mov     bl, MyVar
```

Although it is never possible to move a sixteen bit register into an eight bit register[4], moving a memory location into an eight bit register is always possible. Even if a variable is 16 bits, you could move at least eight of those bits into the eight bit register. Moving a portion of a variable into a register is a very common operation. For example, it is often the case that you want to load a 16 bit register from a double word variable (e.g., a pointer). Since the assembler checks the types of the operands, it wouldn't normally allow you to do this. Fortunately, MASM provides several *coercion* operators to let you change the type of a symbol. The "*type* ptr" operator does this, where *type* represents one of the keywords byte, word, dword, near, far, etc. If MyVar above was a sixteen bit variable, the following statement would let you load the L.O. byte of MyVar into bl: 6.271 To interface with case sensitive languages like C.

```
            mov     bl, byte ptr MyVar
```

7.19 If MyVar is a byte variable, what will "MOV AX, WORD PTR MyVar" do?

____

____

____

The this operand is also useful when defining symbols. This returns the address of the current byte in memory (i.e., the location counter value). If used within an instruction, this corresponds to the first byte of that instruction. The this operand takes the form "this *type*" where *type* is one of MASM's data types (described above). Generally, you would use the this operand with an equ directive as follows:

```
BSymbol     equ     this byte
```

This assigns the current location counter (and segment value) to BSymbol and sets its type to *byte*. Note, by the way, that the statement above is identical to:

```
BSymbol     equ     byte ptr $
```

Remember, "$" returns the current value of the location counter.

Most programmers use the "THIS *type*" form in EQU directives and the "$" form as operands to instructions. However, the two are mostly interchangeable. The following statement is perfectly legal:

```
            mov     ax, this word
```

7.20 What will the instruction above do?

____

____

4. It is possible, using sign extension, to move an eight-bit register into a sixteen bit register.

More often than not, programmers use the "`this` *type*" with the `equ` directive to generate two symbols for the same address, allowing them to easily access the data at that address as two separate types. Consider the following code sequence:

```
WPtr        equ     this word
DPtr        dword   0
             .
             .
             .
            mov     WPtr, di
            mov     WPtr+2, es
             .
             .
            les     di, DPtr
```

7.21 What would you add to the above if you needed to access DPtr as a sequence of bytes in addition to words and dwords?

____

____

7.4 Literal Constants

MASM lets you specify five different types of literal (non-symbolic) constants: integers, reals, strings, text, and BCD values. The first four types you will frequently use in a typical assembly language program. BCD operations do not occur very often, we will not consider them in this laboratory manual.

MASM lets you specify integer constants in one of four different forms: binary, octal[5], decimal, and hexadecimal. An integer constant is one that begins with a decimal digit and is followed by a string of decimal digits or A…F (for hexadecimal constants). To specify the radix for an integer constant that is not the current default, MASM requires a suffix of "b" or "B" for binary, "t", "T", "d" or "D" for decimal, or "h" or "H" for hexadecimal. Examples:

```
10110b          1234                    1234d          1234h
```

7.22 Why isn't "ABCDh" a valid hexadecimal constant?

____

You can change the default radix using the`.radix` directive. The single operand to this directive must contain a value in the range 2…16. Until the next `.radix` directive, all integer constants in your program without a radix suffix ("b", "d", "t", or "h") will use the specified base. You can restore decimal as the default base using the `.radix 10` directive.

7.23 If the current default radix is base 16 (hexadecimal) and you use a constant of the form "12d" MASM treats this as the hex constant "12Dh". How do you specify the decimal constant 12?

____

MASM lets you specify string constants by surrounding the desired text with either a pair of quotation marks or a pair of apostrophes. If you need to include an apostrophe or quote within a string, the easiest solution is to use the other character as the delimiter for the string, e.g.,

```
"It's got an apostrophe in it."
'He said "How are you?"'
```

5. We will ignore the octal base throughout this text.

MASM also allows you to double up an apostrophe or quote within a string delimited by that same character, just like Pascal and some other high level languages:

6.272 Text

```
"He said "How are you?""
```

7.24 If you need to include *both* apostrophes and quotes within a string, how could you do this?

____

____

Text equates let you perform textual substitutions during assembly. These involve the use of the `textequ` and `equ` assembler directives[6]. You can use the `textequ` directive to define a text constant as follows:

6.273 Number

```
symbol        textequ             <textual data>
```

The angle brackets ("<" and ">", also know as the less than and greater than symbols) must surround the textual data you wish to define. When MASM encounters *symbol* after processing the `textequ` directive above, it simply substitutes the text between the angle brackets for that symbol.

6.274 2

7.25 Suppose you have the text equate "Var textequ <[bx+6]>" within your program. If you write the statement "mov ax, var" in your code, what statement will this actually produce?

____

____

Please note that the following two equates are *not* equivalent:

6.275 You cannot move immediate values into a segment register.

```
item          equu      $+2
item2         textequ   <$+2>
```

The first equate computes the value of the location counter plus two and assigns this value to `item`. The textual equal, on the other hand, simply substitutes the string "$+2" everywhere it sees the symbol `item2`. Since the value of the location counter will probably be different for each usage of `item2`, it will not produce the same result at the item equate. Look at the object code produced in the following listing:

```
0000                  cseg          segment
0000 = 0002           equ1          equ        $+2
     = $+2            equ2          equ        <$+2>
0000                  MyProc        proc
0000 B8 0000                        mov        ax, 0
0003 8D 1E 0002 R                   lea        bx, equ1
0007 8D 1E 0009 R                   lea        bx, equ2
000B 8D 1E 0002 R                   lea        bx, equ1
000F 8D 1E 0011 R                   lea        bx, equ2
0013                  MyProc        endp
0013                  cseg          ends
                                    end
```

6.276 It will load MyuVar into AL and the byte following MyVar into AH.

6.277 Load the opcode of the MOV AX, displacement into AX.

6. Microsoft supports the EQU directive for compatibility reasons. They suggest that, for compatibility with future versions of MASM, that you always use the TEXTEQU directive for textual equates.

7.26 In the listing above, why does "LEA BX, equ1" always produce the same opcode bytes while "LEA BX, equ2" does not?

Real constants take the same form as their HLL counterparts (see the textbook) and are required by certain MASM directives (.e.g, `real4`) and some 80x87 machine instructions.

7.5 Procedures

MASM's `proc` and `endp` directives let you define procedures in an assembly language program. Although the `proc` and `endp` directives are not strictly necessary in an assembly language program, they do simplify assembly language programs and you should always use them when creating procedures. The basic syntax for the `proc` directive is

```
ProcName        proc        operand(s)
```

Where *ProcName* is the name of the procedure you wish to define and the operand field is either blank or contains the keyword `near` or `far`. If either keyword is present, then the procedure will be a near or far procedure, depending upon the operand. If the operand field is blank, then the procedure usually defaults to a near procedure unless you've placed a `.model` directive in the source file. If you have, then the default depends upon the operand of the `.model` directive. See the MASM Programmer's Guide for more information on `.model`.

The choice of near or far as an operand to the `proc` directive has two immediate effects on your program. First, any `call` instruction that references such a procedure automatically becomes a near or far call depending on the type of the procedure. Second, MASM automatically converts any `ret` instructions within the procedure to `retn` or `retf` as appropriate.

7.27 If you want to force a far return from a near procedure, what instruction could you use to do this?

7.28 If you wanted to create a near procedure named "MyProc", what would the PROC statement look like?

MASM uses the `endp` directive to mark the end of a procedure. Unlike HLLs, MASM will not automatically issue a `ret` instruction immediately before an `endp` directive. It is your responsibility to put an instruction that changes the flow of control before the `endp` if you do not want to execute whatever follows the procedure upon hitting the `endp` directive. The `endp` directive requires a label in the label field that must match the label in the corresponding `proc` directive. The syntax is the following:

```
ProcName        endp
```

All statement labels (those with a ":" suffix) within a procedure are *local* to that procedure. This means that you cannot reference these labels from outside that procedure and any attempt to do so will produce an "undefined symbol" error. If you need to reference a statement label from outside the procedure, use a double colon ("::") after the label you want to be global. E.g.,

```
ProcWGlbls     proc
               mov       cx, 10
GlobalLbl::    loop      GlobalLbl
               ret
ProcWGlbls     endp
```

Please note that only statement labels are local to a procedure. Most other symbols including those declared with `equ`, `byte`, `word`, etc., are global to the procedure and you may reference them from other parts of your program.

7.29 How would you rewrite ProcWGlbls if you did not want "GlobalLbl" to be a global symbol?

____

____

____

6.278 BPtr equ this byte

6.279 It needs to begin with a decimal digit, e.g., "0".

7.6 Address Expressions

Anywhere MASM allows a symbol or numeric value (e.g., a displacement in an instruction), it will allow an address expression. An address expression is an algebraic expression that MASM computes at assembly time. If this expression appears in the displacement field of an instruction, then MASM computes the result of that expression and places the result in the displacement field of the instruction's opcode.

6.280 "12T"

Address expressions allow the following arithmetic, logical, and relational operators:

Table 16: Arithmetic Operators

Operator	Syntax	Description
+	+*expr*	Positive (unary)
-	-*expr*	Negation (unary)
+	*expr* + *expr*	Addition
-	*expr* - *expr*	Subtraction
*	*expr* * *expr*	Multiplication
/	*expr* / *expr*	Division
MOD	*expr* MOD *expr*	Modulo (remainder)
[ ]	*expr* [*expr*]	Addition (index operator)

6.281 Whichever delimiter you use to surround the characters in the string, double that character up in the string.

6.282 mov ax, [bx+6]

Table 17: Logical Operators

Operator	Syntax	Description
SHR	*expr* SHR *expr*	Shift right
SHL	*expr* SHL *expr*	Shift left
NOT	NOT *expr*	Logical (bit by bit) NOT
AND	*expr* AND *expr*	Logical AND
OR	*expr* OR *expr*	Logical OR
XOR	*expr* XOR *expr*	Logical XOR

Table 18: Relational Operators

Operator	Syntax	Description
EQ	*expr* EQ *expr*	True if equal
NE	*expr* NE *expr*	True if not equal
LT	*expr* LT *expr*	True if less than
LE	*expr* LE *expr*	True if less than or equal
GT	*expr* GT *expr*	True if greater than
GE	*expr* GE *expr*	True if greater than or equal

MASM generates zero for false and 0FFFFFFFFh for true.

7.30 What will the instruction "MOV AL, X+1" do?

____

____

Although the addition and subtraction operators are the most often used operators, the others have their uses as well. For example, suppose you have a word array containing 256 elements that you want to index using ASCII characters. If you wanted to initialize element "A" to 1250 you could use the following instruction:

```
mov     Array["A"*2], 1250
```

This is far more readable, and understandable, than the corresponding code that does not use an address expression:

```
mov     Array[130], 1250
```

7.31 Suppose this array contained double word elements rather than word elements. How could you initialize the element at index "A" to 1250 in this case (assume you are on an 80386 processor)?

____

____

The logical and relational operators have some obvious uses with the conditional assembly statements, the following example generates an error if the "ShortProc" procedure is longer than 16 bytes:

```
ShortProc     proc      near
               .
               .
               .
ShortProc     endp
SPLen         equ       $-ShortProc

              if        SPLen GT 16
              .err
              endif
```

The ($-ShortProc)" operand computes the length of the procedure. Note that we could have placed the length computation directly into the IF directive as follows:

```
if      ($ - ShortProc) GT 16
```

Some languages, like Pascal, use *length prefixed* strings where the first byte of a character array contains the length of the string that follows. Counting up the characters in a string can be a real chore, especially if the string is long or if you change it often. However, by using address expressions you can have MASM automatically compute the length for you:

6.283 "$+2" is computed only once at the point of the equ directive. The textual equate, however, substitutes "$+2" at each occurrence of *equ2*, which causes a computation of "$+2" at that point.

```
LenPrefixed   byte      EndStr-$-1
              byte      "This is my string of characters."
EndStr        equ       this byte
```

7.32 What is the value assigned to the "EndStr" symbol above?

The logical operators are useful on occasion as well. Suppose you have the following equate which is a bit mask for converting upper case to lower case:

```
CaseBit       equ       20h
```

You could convert upper to lower case with the single instruction:

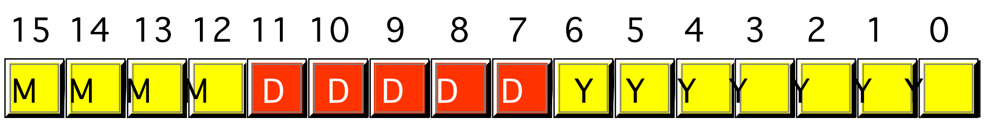
```
              or        al, CaseBit
```

The opposite operation, converting lower to upper case requires ANDing with 5Fh rather than 20h. You can convert 20h to 5Fh by using the logical NOT operator. So you can still use the CaseBit symbol with an AND instruction as follows:

6.284 RETF

```
              and       al, not CaseBit
```

As a final example, consider the DATE data type from Chapter Two:



If you have three symbols Month, Day, and Year, equated to appropriate values, you can pack them into a single word taking the above format with the statement:

6.285 MyProc proc near

```
ThisDate      word      (Month shl 12) + (Day shl 7) + Year
```

There are many restrictions on the operators you can use with certain symbol types. For example, MASM will not allow you to compute "($ MOD 15)" because it doesn't know the final value of "$"[7]. On the other hand, it can compute the distance between two *relocatable* objects like a statement label and "$", which is why "EndStr-$" is acceptable to the assembler. To sort out all the crazy details, please see the MASM Programmer's Guide.

7.7 Type Operators

The MASM type operators let you coerce the type of one operand to another type or return some intrinsically useful formation about that operand. The following table lists some of the commonly used type operators MASM provides (see the textbook for a more complete list):

6.286 Remove the second colon from the GlobalLbl symbol.

7. Only the linker will know this value.

Table 19: Common Type Operators

Operator	Syntax	Description
`PTR`	byte ptr *expr* word ptr *expr* dword ptr *expr* qword ptr *expr* tbyte ptr *expr* near ptr *expr* far ptr *expr*	Coerce *expr* to point at a byte. Coerce *expr* to point at a word. Coerce *expr* to point at a dword. Coerce *expr* to point at a qword. Coerce *expr* to point at a tbyte. Coerce *expr* to a near value. Coerce *expr* to a far value.
`this`	this *type*	Returns an expression of the specified type whose value is the current location counter.
`seg`	seg *label*	Returns the segment address portion of *label*.
`offset`	offset *label*	Returns the offset address portion of *label*.
`lengthof`	lengthof *variable*	Returns the number of items in *variable*.
`sizeof`	sizeof *variable*	Returns the size, in bytes, of *variable*.

You've already see examples of the first four operators in this chapter. There is no need to discuss them further here.

The `lengthof` operator returns the total number of elements in an array, assuming you've defined the array with a single statement using the `dup` operator. For example, if you've defined an array as follows:

```
MyArray        word       64 dup (?)
```

then `mov cx, lengthof MyArray` loads `cx` with 64. Note that the number of bytes in the array are irrelevant. `Lengthof` would return 64 for `MyArray` even if it had been a byte or dword array.

One advantage to using the `lengthof` operator is that you can set up your code to automatically adjust to the size of the array. If you wanted to initialize each element of MyArray to 0, you could use the following loop:

```
               mov        cx, lengthof MyArray
               mov        ax, 0
               lea        bx, MyArray
FillLp:        mov        [bx], ax
               add        bx, 2
               loop       FillLp
```

If you change the size of the array to 256 at some future date, you will not have to modify the code. It will automatically adjust to the new size of the array and it would still work correctly.

The `sizeof` operator returns a constant that gives the number of bytes in an array or structure. This operator is quite useful when computing indexes into arrays and performing other computations that depend on the size of some data structure. For example, suppose you want to create an array of structures. You could use the following declarations to easily accomplish this:

```
MyStruct       struct
Field1         byte       ?
Field2         word       ?
Field3         dword      ?
MyStruct       ends

MyArray        byte       64 * (sizeof MyStruct) dup (?)
```

Note that this code sequence uses the `byte` pseudo-opcode to reserve storage for the array since the `sizeof` operator returns the size of an object in bytes. Please be aware that using the `lengthof` operator on MyArray returns the number of elements in this *byte* array. The assembler thinks that you've created a byte array, not a MyStruct array.

6.287 Load AL with the byte following memory location X.

7.33 How could you create a 16 x 16 two dimensional array of MyStruct using the sizeof operator and the above technique?

Keep in mind that this isn't the best way to allocate an array of structures. The standard way, of course, is to use the following declaration:

```
MyArray       MyStruct  64 cup ({})
```

The sizeof operation is very useful for computing data structure size when using the UCR Standard Library Malloc routine. For example, if you wanted to allocate storage for a single instance of MyStruct on the heap, you could use the following code:

```
              mov       cx, sizeof MyStruct
              malloc
              jc        InsufficientMemory
```

6.288 mov Array["A"*4], 1250

If you wanted to allocate an array of MyStruct, say 64 elements, you could use the following code:

```
              mov       cx, 64 * sizeof MyStruct
              malloc
              jc        InsufficientMemory
```

This provides a reasonable example of where you really can use the sizeof operator to allocate an array of structures. Malloc always requires the size in bytes, so the sizeof operator is quite useful for creating such arrays.

7.8 Conditional Assembly

Conditional assembly is a very powerful feature provided by MASM. Conditional assembly, as its name implies, lets you choose whether or not to assemble certain statements into your object module depending on some condition that exists at assembly time. Two common uses for conditional assembly are to include special debugging statements in your program that you can easily remove after debugging and providing a way of assembling different code depending on the processor available.

6.289 if ($ mod 15) NE 0
byte (15 - ($ mod 15)) dup (?)
endif

The ifdef and ifndef directives are probably the most commonly used conditional assembly directives. They use the following syntax:

```
ifdef   label                                   ifndef            label
  .                                               .
  .                                               .
  .                                               .
endif                                           endif
```

The operand to these two directives must be a single symbol.

If the symbol is defined in the current source file *before* encountering the ifdef statement, MASM will effectively ignore the ifdef and endif statements and assemble all the code between them. If the symbol has not been defined at that point, MASM will ignore all the statements between the ifdef and the endif statements. Ifndef works in a similar fashion except it assembles the instructions if the symbol is *not* defined at that point.

Consider the example mentioned earlier, that of including debug code in your source files. Consider the following sequence:

```
print
byte      "Calling 'MySub' from 'ThisProc'",cr,lf,0
call      MySub
print
byte      "Returning from 'MySub' to 'ThisProc'",cr,lf,0
```

This debugging code traces the execution of the program.

There is a major problem with inserting statements like this in your code. It's quite possible that you never execute this sequence of instructions during normal operations. So you might not ever see these messages and, therefore, *you could forget to remove them from your code*. Later, when someone uses your program they might cause the program to execute this sequence of instructions producing some embarrassing diagnostic messages on their screen.

Even if you always do remember to remove the debugging statements, there is a minor problem. What happens if at a later date you want to see if the program calls MySub ? Then you'd have to put these diagnostic messages back into your code. Now consider the following example:

```
ifdef     debug
print
byte      "Calling 'MySub' from 'ThisProc'",cr,lf,0
emdif

call      MySub

ifdef     debug
print
byte      "Returning from 'MySub' to 'ThisProc'",cr,lf,0
endif
```

With these conditional assembly statements in your program you will get the diagnostics assembled only if there is a symbol "debug" that appears earlier in your program. So by placing a single symbol, "debug", at the beginning of your program you can automatically turn on all debugging statements[8]. Likewise, by removing the debug statement, you can automatically disable all the debugging statements. Note that MASM ignores any value debug might have. Ifdef/ifndef only tests to see if the symbol is defined. You could use the following statement to define debug in your program:

```
debug          equu        0
```

The following code sample shows how MASM handles the ifdef directive. In this example there are two symbols that ifdef directives check: debug1 and debug2. In this instance debug1 has a definition but debug2 does not. Note that MASM does not emit any code (check the location counter value!) for the statements surrounded by the ifdef debug2 and corresponding endif statements.

```
                     ; 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.

 = 0000              DEBUG1        =          0

 0000                cseg          segment.
0000              DummyProc        proc
                                   ifdef      DEBUG2
                                   print
                                   byte       "In DummyProc"
                                   byte       cr,lf,0
                                   endif
0000 C3                            ret
0001                 DummyProc     endp
```

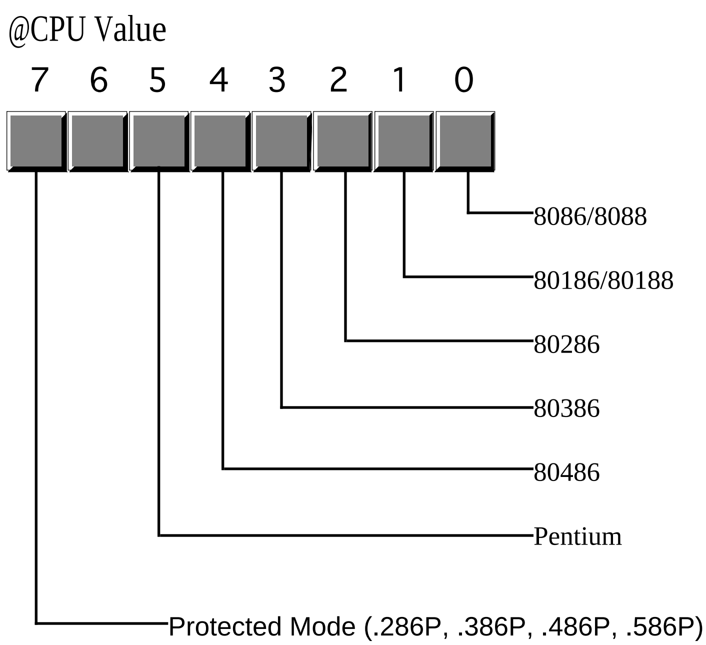
8. ML also has a command line option, /Ddebug, that would let you define this symbol when you assemble the program. This is quite handy if you only need to turn on debugging every now and then.

```
0001                Main          proc
                                  ifdef      DEBUG1
                                  print
0006 43 61 6C 6C 69 6E            byte       "Calling DummyProc"
67 20 44 75 6D 6D
79 50 72 6F 63
0017 0D 0A 00                     byte       cr,lf,0
                                  endif

001A E8 FFE3                      call       DummyProc

                                  ifdef      DEBUG1
                                  print
0022                              byte       "Return from "
52 65 74 75 72 6E                 byte       "DummyProc"
20 66 72 6F 6D 20
44 75 6D 6D 79 50
72 6F 63
0037 0D 0A 00                     byte       cr,lf,0
                                  endif
003A C3                           ret
003B                Main          endp
003B                cseg          ends
                                  end
```

6.290 Ary byte 16 * 16 * sizeof MyStruct dup (?)

Another common problem is developing assembly code that you can assemble for different 80x86 processors. If you write code using 80386 instructions, however, your programs will not run on earlier processors. One alternative is to supply *two* executables. By conditionally assembling one sequence of instructions for 80386 and later processors and another sequence for pre-80386 processors, you can put all your code into a single source file.

MASM provides a predefined symbol, @CPU, that contains certain bits set depending on the CPU type specified by the .8086, .186, .286, .386, .486, and .586 directives. The return value for @CPU is the following (a set bit indicates that the corresponding CPU directive is active):



7.34 If you've specified the .386 directive in your program (and no other processor selection directives appear afterwards), what value will @CPU return?

____

So if you want to assemble code differently depending upon the availability of an 80386 processor, you could use code like the following:

```
dseg          segment
BigVar        dword     ?
               .
               .
               .
dseg          ends

cseg          segment
               .
               .
               .
              if        (@CPU and 1000b) NE 0

; Okay, we've got an 80386 or better, use 32-bit instrs.

              mov       BigVar, 0

              else

; If it's an 80286 or earlier, break the 32 bit operation
; up into two 16 bit operations.

              mov       word ptr BigVar, 0
              mov       word ptr BigVar+2, 0

              endif
```

7.35 What IF directive would you use for the above if you wanted an 80486 or better processor?

____

There are many conditional assembly directives beyond the ones presented here. See the textbook and the MASM Programmer's Guide for more details.

7.9 Macros

Macros are similar to textual equates; they let you replace a single identifier with some text during the assembly process. Macros, however, are far more flexible than simple textual equates because macros support multi-line substitutions and parameters.

A macro *definition* takes the following form:

```
MacroName     macro                   optional parameters

       < sequence of valid MASM statements>

              endm
```

A macro *invocation* takes the form:

```
              MacroName               optional parameters
```

Usually, you do not want to use macros to create new "instructions" for the 80x86. However, there are some times when creating new "instructions" with macros is perfectly reasonable. For example, the 80186 and later processors let you push an immediate value onto the stack. The 8086 and 8088 do not. The following macro provides a "push immediate" instruction that works on all processors:

```
PSHI            macro     value
                if        (@CPU and 10b) NE 0          ;80186 or later
                push      value
                else
                mov       ax, value
                push      ax
                endif
                endm
```

7.36 One big problem with macros is that they often produce *side effects*. A side effect is some computation or operation that takes place that is incidental to the actual operation of the macro and is not obvious from the invocation of the macro. The macro above suffers from a major side effect. What is it?

Here's another example showing how to use macros to allow you to prepare optimized instructions for different processors in the 80x86 family. On the 80286 and later processors, the `shl` instruction allows an immediate value other than one as the second operand. The following `shli` (shift left immediate) macro generates a sequence of `shl operand, 1` instructions for the 8086/8088 and a single instruction for the 80286 and later processors:

```
SHLI            macro     operand, count
                if        (@CPU and 100b) NE 0   ;80286 or later.
                shl       operand, count
                else
                repeat    count
                shl       operand, 1
                endm
                endif
                endm
```

The DATE data type provides a good example of how you could use macros to simplify data entry into your program. The following macro requires three operands: a month, day, and a year value. It checks these values to see if they are within a reasonable range and then packs them into a single word as described in Chapter Two:

```
Date            macro     month, day, year
                if        (month eq 0) or (month gt 12)
                echo      Month value is out of range
                .err
                endif

                if        (day eq 0) or (day gt 31)
                echo      Day value is out of range
                .err
                endif

                if        (year ge 100)
                echo      Year value is out of range
                .err
                endif

                word      (month shl 12) + (day shl 7) + year
                endm
```

MASM also provides a directive similar to `struct` (`record`) that lets you create packed data types. However, it will not let you provide the same level of error checking as this macro does[9]. See the MASM Programmer's Guide or the Quick Help on-line help system for more details.

7.10 Managing Large Programs

MASM provides five directives that let you break large programs into smaller pieces that are easier to manage. Of these, you can easily get by with just two: `include` and `externdef`. Therefore, we will concentrate on those two directives here. If you wish to learn about the other three, see the textbook.

Although you probably think you're not going to be writing large programs anytime soon, any time you use the UCR Standard Library (over 23,000 lines of code at last count, and rising) you are working with a big program since you inherit all the code from that project. Even if your own programs never exceed 1,000 lines, knowing how to use separate compilation (or, in the case of MASM, separate assembly) can help you write your assembly language programs faster.

The `include` directive lets you insert a separate file into your source code whenever you run MASM. Although you can use the `include` directive for a variety of purposes, we're going to use it to include important information about symbols that you need to share between modules. The "CONSTS.A" file in the UCR Standard Library is a good example of a simple include file. This file contains various constants and macros that you will often use when writing assembly language programs. Indeed, few of the statements in this include file have anything to do with the Standard Library at all. It contains definitions for symbols like `cr`, `lf`, `exitpgm`, `dos`, and so on.

By including the "CONSTS.A" file in your programs, you save the effort of declaring these constants yourself. Furthermore, by having this file available you are more likely to reuse the same symbols (like `cr` and `lf`) over and over again in all your programs. This makes them more consistent and, therefore, easier to read and understand. Code reuse is an important tool for those who want to write reliable programs as quickly as possible.

The `include` directive uses the syntax:

```
include                 filename
```

During assembly it copies the specified file into your source file at the point of the include directive.

The `externdef` directive is the primary tool you will need to implement separately compiled modules. EXTERNDEF allows you to *import* and *export* names across modules. This directive takes the form:

```
externdef        symbol:type, symbol:type, ...
```

One or more symbols may appear in the operand field. The types are the standard MASM type identifiers: `byte`, `word`, `dword`, `near`, `far`, `abs`, user defined types, etc.

7.37 What form would the EXTERNDEF statement take if you wanted to declare a single symbol "MyExt" of type FAR?

____

To use separate assembly you must do three things. First, you need to create the two (or more) source files that contain the separate modules you want to assemble. Next, you need to communicate the names of routines, variables, and other symbols you wish to share amongst the modules. Finally, you need to merge the separately assembled modules into a single executable file.

When creating your source modules you should attempt to organize your code with as few external dependencies as possible. That is, when you take a big program and split it into separate modules, you should organize each module so that it contains as few external references as possible. For example, if you have several procedures that all share a single array, especially if no other procedures use that array, should all go into the same module. Ideally you should place any set of logically related operations, especially if they share some common routines and data, in the same module. For example, the UCR Standard Library places all the floating point routines in a single module because they share some common data and some common (internal) routines.

9. On the other hand, it does provide many additional features that the macro implementation does not.

6.291 0Fh

On the other hand, if you take this attitude to the extreme you wind up with one big program again. Choosing a module size that is *just right* takes lots of experience. For example, although the floating point package in the UCR Standard Library is rather large, most of the string modules contain only one or two procedures.

7.38 If you have a string routine and an output routine, and both are called from your main program, should you combine them into a single module?

The externdef directive provides the mechanism whereby you can make one module aware of certain names within another module. Remember, if you call a routine that is not in the current module, MASM will generate an undefined symbol error *unless* you tell it about that symbol. The externdef directive is the mechanism you use to tell MASM that a symbol can be found somewhere else.

6.292 if (@CPU and 10000b)

Defining symbols that appear in other modules is only part of the equation. When you assemble a module in MASM, it generally treats all symbols as local to that module. This prevents "name space pollution" that would occur if all symbols in a module were publicly available to all other modules. Were this to occur, you would not be able to reuse the same (local) symbol twice in two separate modules. So usually symbols within one source module are *private* to that module and unavailable to other modules. To export a name to other modules (make it *public*) you use the same directive you use to import a name: externdef. The beauty of using the same directive to import and export public names is that you can place the set of externdef directives in an include file and include this same file in the module that imports the name and the module that exports the name. This simplifies program maintenance since any changes to the name (such as its type) need only be made to a single include file rather than to several different files.

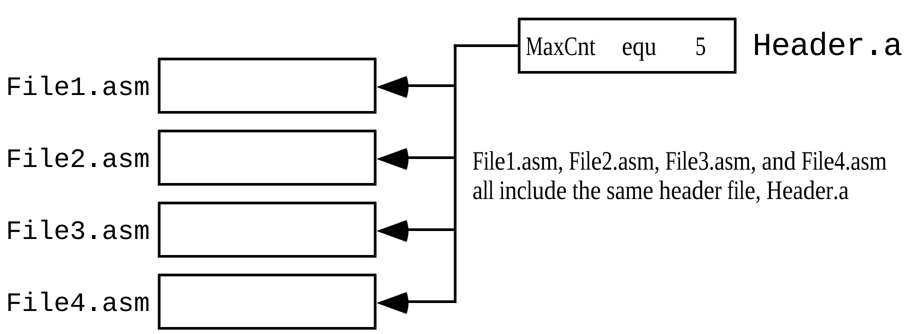
The laboratory section of this manual contains a complete example of a program that uses the externdef directive to share public information between modules.

6.293 It modifies the value of the AX register.

7.11 Project Management with MAKE/NMAKE

Breaking up a project into separate modules will speed up the development process. Only assembling those files that you change can dramatically reduce the time you spend compiling and assembling your project. Unfortunately, breaking up your modules as described in the previous section introduces a problem you don't have with the single source file module: dependencies. Unless properly managed, file dependencies can introduce yet another source of bugs into your programs.

To understand the problem with file dependencies, consider the following modularization of a project:



If you assemble and link together these modules then decide to change some code in File3.asm, it's obvious you must reassemble File3.asm and then relink the object modules to get an updated .EXE file. What is less obvious is what happens when you change a header file like Header.a. Since other modules include the header file *only during assembly*, you must reassemble any module that includes a modified header file.

For example, suppose the four modules above all use the MaxCnt equate to control the number of iterations in various loops. If you assemble those four modules, the value five is going to be embedded into various instructions in the object modules File1.obj, File2.obj, File3.obj, and File4.obj. If you change MaxCnt in Header.a, you will have reassemble all four modules in order to change that constant in each of the object modules.

In a large project it is quite rare than all modules include every header file. In the example above there might actually be ten different modules with only four of them including Header.a. So when you modify a header, it is very easy to forget which files include that header and only reassemble those. Your program would obviously develop problems if three of the modules used the constant MaxCnt equal to eight and one of them used MaxCnt equal to five.

One solution to this problem, of course, is to reassemble all files whenever you modify a header file. Unfortunately, this eliminates the benefits of using MASM's separate assembly and linking facilities. What you really need is a mechanism that automatically assembles any files *dependent* upon Header.a should you make a change to Header.a. The *make* program is the tool that does this for you.

Make is a program management tool. Microsoft provides a version of make, nmake, with MASM that allows you to automatically process files that depend upon one another[10].

The make program requires a source input file containing a sequence of commands. Each command takes the following form:

```
target : <dependency list>
    <DOS commands to execute>
```

The target file is the output file you want to produce. This can be any kind of DOS filename, but for our purposes it will generally be an .OBJ file or an .EXE file. The dependency list is a list of files on which the target file depends. The *dependency line* (the first line above) for File1.asm is

```
file1.obj: file1.asm header.a
```

File1.obj is the target (output) file and it depends only on File1.asm and Header.a. Generally, if you make any changes to the files in the dependency list, you will have to build a new target file.

Nmake.exe, Microsoft's version of the make tool, uses the *file date and time stamps* from MS-DOS to determine whether a file is *out of date*. In the example above nmake compares the date/time stamp of File1.obj against the date/time stamps of file1.asm and header.a. If the date/time on File1.obj is earlier than either of these two files, this means that there have been some changes made to files in the dependency list and something needs to be done about this.

The *something* that nmake does is execute the DOS command(s) that follow the dependency line. This can be any valid DOS command but usually it is the command(s) necessary to bring file1.obj up to date. A typical nmake command for file1.obj takes the form:

```
file1.obj: file1.asm header.a
    ml /c file1.asm
```

Note that the target file on the dependency list must begin in column one. The commands following the dependency line must *not* begin in column one. When nmake determines that a file dependency requires some action, it will execute all commands following the dependency line until it finds another dependency line beginning in column one.

The nmake commands for file2 and file3 are

10. Microsoft's original program was called *make*. However, their original make was incompatible with most programs by that name so when they released a compatible version they called it *nmake*, presumably for *new* make. Borland and many other vendors supply comparable programs that are just called make. Keep in mind, however, that if you use a Microsoft product called make it is probably very old and a bit different than the standard definition of make presented here.

```
file2.obj: file2.asm header.a
    ml /c file2.asm

file3.obj: file3.asm header.a
    ml /c file3.asm
```

6.294 externdef MyExt:far

7.39 What is the nmake command for file4?

A complete make file describes how to build the final .EXE file and any .OBJ (or other) files that the .EXE file depends on. The complete make file for the File1..File4 project is the following:

```
file1.exe: file1.obj file2.obj file3.obj file4.obj
    ml file1.obj file2.obj file3.obj file4.obj

file1.obj: file1.asm header.a
    ml /c file1.asm

file2.obj: file2.asm header.a
    ml /c file2.asm

file3.obj: file3.asm header.a
    ml /c file3.asm

file4.obj: file4.asm header.a
    ml /c file4.asm
```

6.295 No.

Nmake only executes the first dependency line in a make source file[11]. So nmake would compare the date/time of file1.exe against the date and times of the .OBJ files in the dependency list. Now you might think that this would be insufficient. After all, if file1.exe is newer than any of the .OBJ files but you've changed the header.a file, obviously you still need to reassemble and link everything. Fortunately, nmake always performs the *transitive closure* on the dependency list. This means that before comparing the date and time of file1.exe against all the .OBJ files, it makes sure that all the .OBJ files in the dependency list are up to date as well. If there is a dependency line for a given item, nmake executes that command to see if it changes the date/time. In the example above, changing the date/time of the header.a file would cause *all* the .OBJ files to be older than the header.a file, hence nmake would execute all the ML commands associated with the .OBJ targets. This, in turn, would change all the dates and times on the .OBJ files that would cause nmake to execute the "ML file1.obj ..." command to link the new .OBJ files together and produce a new .EXE file.

7.40 Explain what would happen with the above if you just modified the file4.asm file and all the other files were up to date:

The make "language" supports many other features such as macros, variables, and so on. The simple rules presented above, however, are all that are really needed except for the most sophisticated of projects.

11. Actually, you can specify from the nmake command line that it execute other dependency lines as well. However, we'll always use the default which is to execute only the first dependency line in the file.

7.12 The UCR Standard Library

The UCR Standard Library contains several hundred routines you can use to simplify writing assembly language programs. This section will not go into the specifics of any of them, instead it will concentrate on the philosophy of the UCR StdLib and provide some examples of its use. For details on the routines themselves, see the textbook and the UCR StdLib documentation that appears on the diskette accompanying this workbook.

The goal behind the design of the UCR Standard Library was simplicity. There are a few commercial assembly language subroutine packages available in the marketplace. The goal behind those (if you believe their press releases) is efficiency. Those packages were intended for professional assembly language programmers who want to save some development time but are not willing to trade away the reasons for using assembly in the first place. The UCR library is not for these people. The UCR Standard Library exists because students have a hard time learning assembly language. The UCR Standard Library simplifies that learning process by making many operations in assembly language as easy as a HLL like C (especially like C).

Passing parameters between routines has always been a hassle in assembly language. As you'll see in Chapter Nine of the textbook, typical compilers generate a considerable amount of assembly code in order to pass a typical set of parameters to a procedure or function. It's not all that uncommon for there to be more statements setting up and passing parameters than there are statements within the procedure or function itself.

The UCR Standard Library's design goal was to simplify the "glue" code necessary to patch several calls together. The StdLib routines generally expect their parameters in 80x86 registers and they generally return any results there as well. Furthermore, a Standard Library routine that returns a value in the registers generally attempts to return that value in a register which is an input to some other routine that could use that value. More often than not, you can make a long sequence of calls to various StdLib routines without any interleaving 80x86 instructions. This tends to make programs much short, easier to understand, and certainly easier to write. There is, of course, one catch: you've got to learn how to use the UCR Standard Library before you can reap its benefits.

The following code sequence reads a string from the user and prints that string back to the display:

```
getsm     ;Read string from user
puts      ;Print that string
putcr     ;Follow with a new line.
```

7.41 The GETSM routine reads a string from the user and returns a pointer to this string in the ES:DI registers. The ATOI call converts the string pointed at by ES:DI to an integer and returns this integer value in AX. The PUTI routine prints the value in AX as a signed integer.

Write a code sequence that reads a string of text from the user (presumed to be decimal digits, converts this to an integer value, and then prints that integer value back to the display.

_____________________________ _____________________________

_____________________________ _____________________________

The memory management routines are the backbone of the library. Indeed, perhaps as many as a quarter of the routines in the library call the memory management routines directly. For many of the remaining routines, you'll often call the memory manager to allocate buffer space for them.

There are three memory management routines you must deal with: `meminit`, `malloc`, and `free`. `Meminit` initializes the memory management system. You should only call it once and you must call it before you call any other memory management routine or any routine that winds up calling a memory manager routine. The SHELL.ASM file, which you should use as a "starter" for all your programs using the standard library, already contains a call to `meminit` at the beginning of the main program.

The other two routines, `malloc` and `free`, are the workhorses in the standard library. `Malloc` (Memory ALLOCation) allocates a block of memory in the free memory area called the *heap*. To call `malloc` you must pass the number of bytes of data you want. If sufficient storage is available on the heap, `malloc` will return a pointer to the newly allocated block. On input, malloc expects the block size in the cx register, it returns the pointer to the block in the es:di registers.

Generally, the only reason for using a memory allocator like malloc is because you do not need to reserve the block of storage for the entire lifetime of your program. After all, if you needed the storage throughout the execution of the program it would be easier to just declare a suitable array in your data segment. In a typical program you will allocate storage for some object, use that object, and when you are finished with that object return its storage to the free space on the heap so you can reuse it The free routine returns storage back to the free list for use by other objects. To free some storage, you simply pass the address returned by malloc to free in es:di.

```
; Example: The following code sequence reads a line of
; text from the user and prints that line. It MALLOCs
; storage for the string, reads the string, prints it,
; then frees the storage for it.

              mov       cx, 128              ;Need 128 bytes for GETS.
              malloc                         ;Ignore any errors.
              gets                           ;Read the input line.
              puts                           ;Print it.
              putcr                          ;Print a new line.
              free                           ;Free up storage.
```

Allocating storage for gets is such a common operation that there is a separate call, getsm, that allocates the necessary storage. This is a combination of the mov, malloc, and gets calls above.

7.42 Rewrite the code above to use the GETSM routine.

______________________________ ______________________________

______________________________ ______________________________

______________________________ ______________________________

______________________________ ______________________________

6.296 file4.obj: file4.asm header.a

6.297 nmake would assemble file4.asm and then link the object files together.

7.13 The MASM and UCR StdLib Laboratory

In this laboratory you will experiment with many of the assembler directives and some of the UCR Standard Library routines. You will learn how to create separately compiled modules and learn to link the results together. You will also control the loading order of various segments and use CodeView to examine the results.

7.13.1 Before Coming to the Laboratory

Your pre-lab report should contain the following:

- A copy of this lab guide chapter with all the questions answered and corrected.
- A short write-up describing the UCR Standard Library routines you use.

See Chapter Two of this laboratory manual for an example pre-lab report.

Note: your Teaching Assistant or Lab Instructor may elect to give a quiz before the lab begins on the material covered in the laboratory. You will do quite well on that quiz if you've properly prepared for the lab and studied up on the stuff prior to attending the lab. If you simply copy the material from someone else you will do poorly on the quiz and you will probably not finish the lab. Do not take this pre-lab exercise lightly.

7.13.2 Laboratory Exercises

In this lab you will perform the following activities:

- You will learn how to make program listings so you can see the actual opcode bytes MASM emits.
- You will examine how MASM maintains the location counter.
- You will experiment with symbol types and extracting the value of a symbol.
- You will experiment with segment loading order and view the results in CodeView.
- You will use the `proc` and `endp` directives to create near and far procedures and see their effects on `call` and `ret` instructions.
- You will assemble instructions with address expressions and examine the object code MASM produces.
- You will use macros, textual equates, and conditional assembly directives within your program.
- You will build a program consisting of several separately compiled modules, link them together, and produce a single executable file from them.
- You will use a make file to control the assembly of a multi-module project.
- You will call several routines in the UCR Standard Library and learn how to link the library with your program.

❏ Exercise 1: Creating a program listing. For many of the experiments in this laboratory you will need to look at the object code emitted by MASM. For some of the exercises you will need to load the finished program into CodeView and inspect the object code using the memory dump and disassemble commands. For many of the exercises, however, you learn everything you need to know by simply looking at an *assembly listing*. To create an assembly listing with MASM you use the /Fl command line option as follows:

```
ml /c /Fl Lab1_6.asm
```

This produces a file labelled “Lab1_6.lst” that contains your original source code annotated with the location counter value and the opcode bytes for each instruction. Take the following short assembly language program (LAB1_6.ASM on the diskette) and assemble it with the /Fl option then edit the resulting .LST file. Print this file using the MS-DOS *PRINT* command and include this printout with your lab report. Comment on the listing. Be sure to point out the different values of the location counter and the length of each instruction in the listing. Also describe the meaning of the information in the symbol table.

```
cseg          segment
Sample4Lst    proc
              push      ax
              mov       bx, 0
              add       ax, bx
              mov       bx, ax
              mov       ds:[1000h], ax
              pop       ax
              ret
Sample4Lst    endp
cseg          ends
              end
```

❏ Exercise 2: The file “Lab2_6.asm” on the disk accompanying this lab manual contains two procedures. To ensure maximum performance on an 80486 processor these procedures should be double word aligned. Assemble this file and produce an assembly listing. Note the offsets of the procedures within the code segment. Next, modify the `segment` directive and use the `para` alignment operand and then insert two `align 4` directives as described in the program’s comments. Then create an assembly listing of the modified file.

For your lab report: Compare the object code in the two listings. Describe what the addition of the `align` directives does to the object code. Include the listings with your lab report.

6.298 getsm
atoi
puti

For additional credit: Devise an IBM/L program to test the execution time of these two routines. Compare the timing with and without the `align` directives. (Hint: put the procedures in the `%init` section and the calls int the `%do` section.)

❏ Exercise 3: Intel's syntax for assembly language (of which MASM is mostly a superset) is peculiar because it is *strongly typed*. The Lab3_6.asm file on your diskette contains many different types of symbols. Assemble this file and create an assembly listing.

For your lab report: Create an assembly listing with a symbol table printout and include this with your lab report. On the listing, identify the type of each symbol and match it with the corresponding entry in the symbol table. Explain why each symbol has it associated type.

❏ Exercise 4: Equates in an assembly language program are useful for many things. A primary use is to create symbolic constants to help make your program easier to read and understand. The short assembly language program in file lab4_6.asm reads ten integers from the user and then computes the average of those ten numbers. Unfortunately, the literal constant "10" appears throughout this code which makes it difficult to modify this program to work with a different number of input values. Modify this program so that a single equate, *NumItems*, at the beginning of the program controls the number of input values.

6.299 getsm
puts
putcr
free

For your lab report: Include the "before and after" listings of this program. Modify the *NumItems* equate and change the value to 15. Run the program to verify that your change works. Modify the *NumItems* equate and change the value to five. Run the program and verify that this change works. Include print-outs of three program executions (10, 15, and five) in your lab report.

❏ Exercise 5: In the program above MASM and the linker will load the data segment into memory before the code segment. In general, it's much better to put the data segment *after* the code segment in memory. If your program has a bug in it and it decides to write 200 integers to the array rather than ten, having the data segment before the code segment would be a disaster since the program would overwrite itself. First, assemble the program as-is with the /Fi option (for CodeView information) and load the program into CodeView. Single step through the first few instructions of the main program (that set up the `ds` register) to verify that the data segment appears in memory *before* the code segment. Then add the following two statements to your program immediately before `dseg`:

```
cseg            segment     para public 'code'
cseg            ends
```

By adding these two lines to the program (and without touching anything else), you can instruct MASM and LINK to load the code segment before the data segment memory. Modify the program you produced in Exercise #4 to do just that. Reassemble the modified version using the /Fi option and load the file into CodeView. Execute the first few instruction in the main program to determine that `dseg` appears after `cseg` in memory.

For your lab report: Include a screen dump of the two programs in CodeView. Mark up the screen dumps and explain how you know that `dseg` follows `cseg` in memory.

For additional credit: Another way to move dseg after cseg is to physical move dseg below cseg in your source file. Do this and produce an assembly listing. What differences, if any, do you see in the object code that the assembler generates? Is there any advantage to placing the data at the end of the file? At the beginning of the file?

- ❏ Exercise 6: MASM's `proc` and `endp` statements control the generation of code in a couple of different ways. The program lab6_6.asm contains two near procedures. Assemble the code and produce an assembly listing. Then change the procedures to far procedures and produce a second assembly listing.

  For your lab report: Identify all the opcodes that are different in the two listings. Explain their differences.

  For additional credit: Modify one of the return instructions to be `retf` and the other to be `retn`. Modify calls to the procedures to be `call near ptr proc1` and `call far ptr proc2`. Generate a second pair of listings, one with both procedure definitions containing a near operand and the second listing with both procedure definitions containing a far operand. Again compare the opcode differences between the two assemblies. Explain the result.

- ❏ Exercise 7: Remove the `ret` instruction from the (original) PROC1 procedure above. Run the program.

  For your lab report: Describe and explain the result in your lab report.

  For additional credit: Explain what would happen if you removed the `ret` instruction from PROC2, as well.

- ❏ Exercise 8: The program in file lab8_6.asm contains several type conflicts. Assume the addresses and registers are correct, all that's missing are coercion operators (i.e., `word ptr`, `byte ptr`, etc.).

  For your lab report: Assemble the code and determine the lines that need the coercion operators. Supply the necessary type coercion operators to remove all syntax errors. Run the program and explain the results.

- ❏ Exercise 9: The lab9_6.asm program uses the SHLI macro that appears earlier in this chapter.

  For your lab report: Assemble the code with and without the .286 directive present. Produce an assembly listing in both cases. Describe the differences between the two programs.

  For additional credit: Testing the @CPU assembler variable only tells you the processor directive currently active in an assembly. It does *not* check to see if you are actually using the specified processor when you run the program. Look up the CPUIDENT routine in the UCR Standard Library and discuss how you could use this procedure to determine the actual CPU in use at run-time.

- ❏ Exercise 10: In this exercise you will learn how to link together separately assembled modules. There are three source files associated with this exercise: Lab10a_6.asm, Lab10b_6.asm, and Lab10_6.a (these files are available on the diskette). Lab10a_6.asm contains the main program and other assorted routines and data definitions. Lab10b_6.asm contains a separately assembled module that the code in Lab10a_6.asm uses. Lab10_6.a is an include file that contains the necessary `externdef` directives and other goodies to make everything work together.

  The ML command uses the syntax:

  ```
  ML options filename filename filename ...
  ```

  Until now you've only supplied one filename on the command line when using ML. Nonetheless, MASM will let you specify several filenames and it will assemble each file and then link their object modules together if all assemblies were successful. The following ML command will assemble and link the Lab6x10a.asm and Lab6x10b.asm files[12]:

  ```
  ML Lab10a_6.asm Lab10b_6.asm
  ```

  ML produces an .EXE file whose name matches the first filename on the command line. So the command above will produce "Lab10a_6.exe" as its final output.

  Although the ML command above separately assembles the two source files and links them together, this particular example will always assemble both source files. This eliminates one of the major benefits of separate compilation: saving time because you don't have to reassemble all source files in a project. Fortunately, ML provides some options that allow you to assemble your source files at different times and link the result together. The first such option is "/c" or "-c" that stands for *compile only* (no link). If you specify this command line option then ML will assemble the specified source file(s) producing .OBJ output(s), but it will not run the linker

12. Since Lab6x10.a is an include file you do not specify its name on the command line. The other two files automatically include the text of this file when MASM assembles them.

on the resulting output. The following command assembles the Lab6x10b.asm file but does not link it to anything:

```
ML /c Lab10b_6.asm
```

Although we have always included the .ASM suffix on ML command line filenames, they are not the only suffix ML allows. In particular, ML allows .OBJ suffixes as well. If you supply an .OBJ file on the command line, ML does not assemble that file, it simply links the object file in with the rest of the files you specify. So two commands that demonstrate separate compilation are

```
ML /c Lab10b_6.asm
ML Lab10a_6.asm Lab10b_6.obj
```

These two commands produce exactly the same result as the ML command with two .ASM files given earlier. The advantage here is that if you make changes to Lab10a_6.asm but do not make any changes to Lab10b_6.asm, you need only execute the second of the two above commands to get a new, correct, .EXE file. As long as you do not change the Lab10b_6.asm file, there no need to reassemble it. While this may not seem like a substantial savings, imagine what would happen if you have a project with 10 .ASM files and you only change one of the source files. Reassembling one file and then linking the 10 .OBJ files together is going to be faster than assembling and linking all 10 source files.

The first filename on the ML command line need not be an .ASM file. For example, if you make changes to Lab6x10b.asm but do not modify Lab10a_6.asm, you could create a new executable using the ML command:

```
ML Lab10a_6.obj Lab10b_6.asm
```

This command will produce the Lab10a_6.exe executable file since Lab10a_6 is the first filename on the command line.

❏ Exercise 11: Using a make file. Once you begin using separate assembly you will need to use make files to automatically assemble dependent modules. An appropriate make file for the above project is the following (see the Lab10_6.mak file on the diskette):

```
lab10a_6.exe: lab10a_6.obj lab10b_6.obj
    ml lab10a_6.obj lab10b_6.obj

lab10a_6.obj: lab10a_6.asm lab10_6.a
    ml /c lab10a_6.asm

lab10b_6.obj: lab10b_6.asm lab10_6.a
    ml /c lab10b_6.asm
```

Delete any .OBJ and .EXE files associated with this project (generated in exercise 10). If you enter the following command, nmake should assemble and link together the files from scratch:

```
nmake lab6x10.mak
```

After nmake creates the new .EXE file, immediately run nmake again. This time nmake will not reassemble the files. Instead, it will simply report that lab10a_6.exe is up to date. Since none of the dependent files have changed, nmake reports that there is no need to reassemble the source files.

Now, make a slight change to the lab10a_6.asm file, perhaps by adding a blank line or a comment to the file. When you quit the editor, MS-DOS will update the time/date stamp on the file so that it is newer than the other files in the project. Use the above nmake command again. Note that nmake only assembles the lab10a_6.asm file and relinks the files. It does not reassemble the lab10b_6.asm file. Repeat this operation after modifying the lab10b_6.asm file.

Finally, try making a small modification to the lab6x10.a header file. Run nmake and note that it reassembles both files.

For your lab report: Include print-outs of the files, modifications, and DOS sessions running nmake in your lab report. Hand annotate the changes and point out the changes that caused reassembly.

7.14 Sample Program

Here is a single program that demonstrates most of the concepts from Chapter Six. 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 Chapter Six, 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.

7.14.1 EX6.MAK

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

```
ex6:ex6.obj geti.obj getarray.obj xproduct.obj matrix.a
       ml ex6.obj geti.obj getarray.obj xproduct.obj

ex6.obj: ex6.asm matrix.a
       ml /c ex6.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
```

7.14.2 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 Six.
;
; 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.
;
;       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]>

byp             textequ <byte ptr>
wp              textequ <word ptr>
dp              textequ <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
```

7.14.3 EX6.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 Six.
; 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,
; 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
; “/Fl” 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.
;
; The IFDEF 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.

              putcr
              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).

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

              mov       dx, [si].DopeVec.Dim2  ;# of times to repeat the loop.
              les       di, ColArray           ;Base address of array.
ColValLp:     mov       ax, es:[di]            ;Fetch current array element.
              mov       cx, 6                  ;Print the value using a
              putisize                         ; minimum of six positions.
              add       di, 2                  ;Move on to next array element.
              dec       dx                     ;Repeat this loop DIM2 times.
              jne       ColValLp
              putcr                            ;End of column array output
              putcr                            ;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.

              mov       Rows, 0                 ;Repeat for 0..Dim1-1 rows.
RowLp:        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.
              putisize
              print
              char      ": ",0

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

              mov       Columns, 0              ;Repeat for 0..Dim2-1 columns.
ColLp:        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.
              putisize                          ;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.

              putcr                             ;End each column with CR/LF

              mov       ax, Rows                ;Move on to the next row.
              inc       ax
              mov       Rows, ax
              cmp       ax, [si].DopeVec.Dim1  ;Have we finished all the
              jb        RowLp                   ; rows?  Repeat if not done.
              ret
PrintMat      endp
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 procfar
               imul     ax, dx
               jno      MulDone
               mov      ax, 8000h             ;Error if overflow occurs.
MulDone:       ret
multiplication endp

division      proc      far
              push      cx                    ;Preserve registers we destory.

              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:”,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:”,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”,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:”,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,
                imul      cx, Rows               ; of the matrix and allocate
                add       cx, cx                 ; sufficient storage for the
                malloc                           ; 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
Item            =           Item+1
                endm


Quit:           ExitPgm                             ;DOS macro to quit program.
Main            endp

cseg            ends

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

zzzzzzseg       segment     para public 'zzzzzz'
LastBytes       byte        16 dup (?)
zzzzzzseg       ends
                end         Main
```

7.14.4 GETI.ASM

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 Six.

                .nolist
                include     stdlib.a
                .list

                include     matrix.a

InpSeg          segment     para public 'input'

; Geti-On entry, es:di points at a string of characters.
;       This routine skips any leading spaces and comma characters and then
;       tests the first (non-space/comma) character to see if it is a digit.
;       If not, this routine returns the carry flag set denoting an error.
;       If the first character is a digit, then this routine calls the
;       standard library routine "atoi2" to convert the value to an integer.
;       It then ensures that the number ends with a space, comma, or zero
;       byte.
```

```
;
;       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            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.
;
; Also note a cute little trick here; by decrementing di before entering
; 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], ','
                je        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], ' '
                je        GoodGeti
                cmp       byp es:[di], ','
                je        GoodGeti
                cmp       byp es:[di], 0
                je        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
                ret
geti            endp

InpSeg          ends
                end
```

7.14.5 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 multidimension array). The cross product program reads two such vectors: one for the column values and one for the row values in the cross product.

```
; 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
```

7.14.6 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];

                mov       RowNdx, -1             ;Really starts at zero.
```

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

              mov     ColNdx, -1              ;Really starts at zero.
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
```

7.15 Programming Projects

- Program #1: Write any program of your choice that uses at least ten different UCR Standard Library routines. Consult the appendix in your textbook and the electronic documentation on the diskette for details on the various StdLib routines. At least five of the routines you choose should *not* appear in this chapter or in Chapter Six of your textbook. Learn those routines yourself by studying the UCR StdLib documentation.
- Program #2: Write a program that demonstrates the use of each of the format options in the PRINTF StdLib routine.
- Program #3: Rewrite the sample program in the previous section so that it uses the ForLp and Next macros provided in Chapter Six of your textbook in place of all the individual instructions that simulate a FOR loop in this code.
- Program #4: Write a program that inputs two 4x4 integer matrices from the user and compute their matrix product. The matrix multiply algorithm (computing C := A * 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.
- Program #5: Modify the sample program in this chapter to use the FORLP and NEXT macros provided in the textbook. Replace all for loop simulations in the program with the corresponding macros.
- Program #6: 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 Chapter Six. You should also take a look at the sample program (see "Sample Program" on page 278) to see how to dynamically allocate storage for arrays and access arrays whose dimensions are not known until run time.
- Program #7: The ForLp and Next macros provide in Chapter Six 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.
- Program #8: 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).

7.16 Answers to Selected Exercises

2) Label, mnemonic, operand, and comment.

6) The order that segments appear in the source file is the primary method for determining segment loading order. The class operand to the segment directive is the secondary mechanism.

7) a. constant (abs)
h. byte
j. macro
k. segment
m. string (or text)

9) b. SHORT lets you force a one byte JMP displacement.

10)
```
mov     bx, offset Table
lea     bx, Table
```
Generally there is no difference between the values the assembler loads into bx by these two instructions.

12) CSEG, ESEG, then DSEG.