4

The Macro Assembler

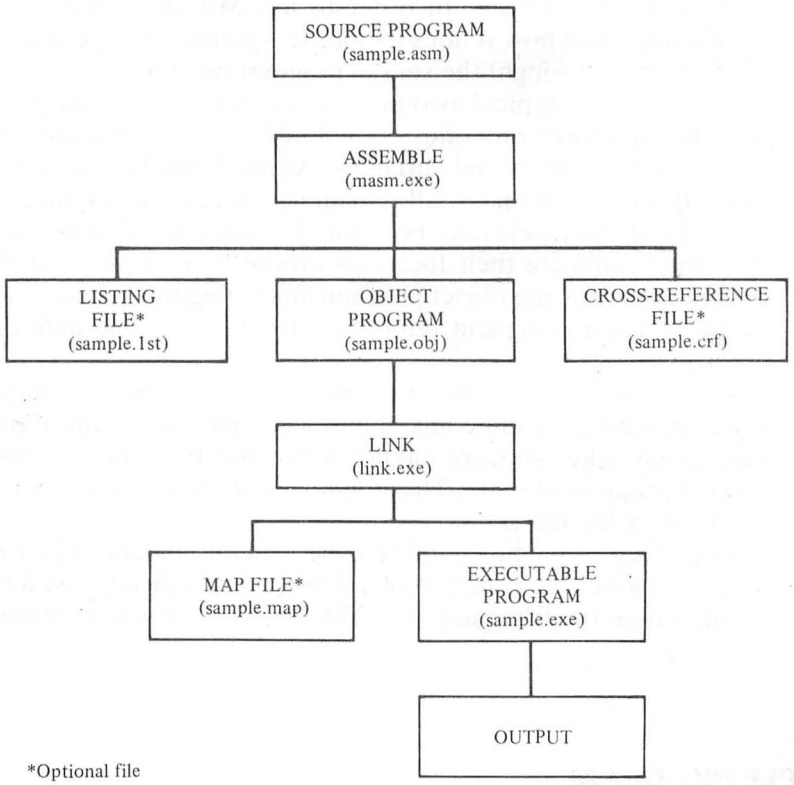
This chapter covers the entire process of assembling, linking, and executing assembly language programs. We will examine each of the files created at assembly time, as well as files created by the linker. Arithmetic and boolean operators will be introduced, along with the OFFSET and PTR operators that clarify the locations and sizes of memory variables. Last of all, we will present a debugging workshop, with examples of some common syntax errors found in assembly programs.

Two types of programs are typically written in assembly language: EXE and COM programs. Throughout most of the book, we will concentrate solely on EXE programs; they are more fully supported by the Macro Assembler and may be debugged using Microsoft CODEVIEW. To get the maximum benefit from this and later chapters, take time out from reading to assemble and run the sample programs. In programming, you learn by doing.

THE ASSEMBLY PROCESS

Figure 4-1 shows the process of assembling, linking, and executing an assembly program. (For the sake of simplicity, we will refer to all of these steps as the

Figure 4-1 The complete assemble–link–execute cycle.



assembly process.) The steps are very similar to those used with high-level languages such as COBOL, FORTRAN, C, or Pascal. A *source program* is a collection of text characters making up a program that follows the rules of assembly language. It must be converted to machine language before it can be run.

Assembly Step. An *assembler* is a program designed to read an assembly language source program as input and generate an object program as output. The assembler may also create a *listing* file, which lists both the source statements and the machine code, and a *cross-reference* file with an alphabetized listing of all labels and identifiers in the program. An *object program* is a machine language interpretation of the original source program, lacking certain information needed by DOS before it may be executed.

Link Step. The *linker* supplied with the Macro Assembler reads an object program and generates an *executable* program. On the IBM-PC, an executable file

has a file extension of EXE. The executable program can be executed just by typing its name.

A Two-Pass Assembler. In order to use MASM effectively, we should know something about how it does its job. It is called a *two-pass* assembler because it reads (passes through) the source program two times.

Let us use a typical two-pass assembler as an example. In the first pass through the source program, symbol tables are constructed, with the names and locations of all labels and variables. As much machine code is generated as possible. In the second pass, all remaining references to symbol locations must be resolved. If any labels have been forward-referenced on the first pass, the assembler must calculate their locations on the second pass. At the same time, the assembler writes the object program and listing file to disk. (A *forward reference* occurs when a statement refers to a label or variable defined later in the program.)

We may also call the object program a *program module,* implying that multiple program modules can be linked into a complete program. Even after the second pass, some addresses are unknown because they refer to labels outside of the current program module. These *unresolved externals,* as they are called, will be resolved by the *linker.*

The Macro Assembler follows the general guidelines just outlined. The two separate passes are transparent unless you specifically ask for a listing file to be generated on the first pass. The MASM *User's Guide* explains how to do this.

A SAMPLE PROGRAM

Let's write and assemble a program that exchanges the contents of two variables. First, the source program is created using a text editor (Figure 4-2). The format is free-form, so labels, comments, and instructions may be typed in any column position. All text typed to the right of a semicolon (;) is ignored. MASM is *case insensitive,* meaning that it doesn't recognize the difference between uppercase and lowercase letters.

Assembler Directives

A number of standard assembler directives used in the sample program are listed here:

.CODE	Mark the beginning of the code segment
.DATA	Mark the beginning of the data segment
DOSSEG	Use standard segment ordering
END	End of program assembly
ENDP	End of procedure

Figure 4-2 The SAMPLE.ASM program, written for MASM Version 5.0+.

```
page ,132

title  Exchange Two Variables                (SAMPLE.ASM)

dosseg
.model small
.stack 100h

.code

main proc

     mov   ax,@data       ; initialize DS register
     mov   ds,ax
 swap:
     mov   al,value1      ; load the AL register
     xchg  value2,al      ; exchange AL and value2
     mov   value1,al      ; store AL back into value1
     mov   ax,4C00h       ; return to DOS
     int   21h

main endp

.data

value1   db    0Ah
value2   db    14h

end main
```

.MODEL	Determine the program memory model size
PAGE	Set a page format for the listing file
PROC	Begin procedure
.STACK	Set the size of the stack segment
TITLE	Title of listing file

Working from the first line to the last in Figure 4-2, we see the following directives:

PAGE. The PAGE directive is used either to begin a new page in the program listing or to select the number of lines per page and the length of each line. The default number of lines per page is 50.

The default line width is 80, which usually causes lines to wrap around when

the file is printed. A value of 132 for the line width is preferable. Examples of settings for the page length and line width settings are as follows:

Example	Length	Width
PAGE	50	80
PAGE ,132	50	132
PAGE 40	40	80

The PAGE directive may be used anywhere in a listing. It forces the listing file to begin printing on a new page. In the program in Figure 4-2, the PAGE directive accepts the default page length and changes the line length to 132.

TITLE. The TITLE directive identifies the program listing title. Any text typed to the right of this directive is printed at the top of each page in the listing file.

DOSSEG. Microsoft high-level languages have a standard order for the code, data, and stack segments. The DOSSEG directive tells MASM to place the segments in the standard order. One can place the segments in any order in the source program without running into compatibility problems. (The segments are identified by the CODE, STACK, and DATA directives.)

.MODEL. The program in Figure 4-2 uses Microsoft's *small* memory model to determine the maximum size of the code and data. The small model is the most efficient model that may be debugged easily using CODEVIEW.

.STACK. The .STACK directive sets the size of the program stack, which may be any size up to 64K. A size of 100h is more than enough for most programs. The stack segment is addressed by the SS and SP registers.

.CODE. The .CODE directive identifies the part of the program that contains instructions (code). The *code segment,* as it is called, is addressed by the CS and IP registers.

PROC. The PROC directive creates a name and an address for the beginning of a procedure. A *procedure* is a group of instructions that is given a common name. We have called this procedure **main**, but any name up to 31 characters would be valid. Most assembly language programs have at least one procedure, the one that executes first.

ENDP. The ENDP directive marks the end of a procedure. Its name must match the name used by the PROC directive.

.DATA. All variables are defined in the area following the .DATA directive, called the data segment.

END. The END directive terminates assembly of the program, and any statements after it are ignored. The syntax is:

```
END [entrypoint]
```

Entrypoint tells the assembler and linker where the program will begin execution. In a program made up of separately compiled modules, only the first module has an *entrypoint* label after the END directive.

Program Instructions

The first two instructions in the program in Figure 4-2 initialize the DS register:

```
mov   ax,@data
mov   ds,ax
```

The name **@data** is a standard MASM identifier that stands for the location of the data segment. (The actual segment name varies, depending on which memory model is used.) By setting DS equal to the start of the data segment, we make it possible for MASM to locate variables correctly.

The next three instructions exchange the contents of **value1** and **value2**:

```
mov   al,value1
xchg  value2,al
mov   value1,al
```

Assembling and Linking

Before we assemble and link SAMPLE.ASM, the following programs must be in the current directory or DOS path:

MASM.EXE	Microsoft Macro Assembler
LINK.EXE	Linker
CREF.EXE	Cross-reference generator

Step 1. Assemble the Program. We will assume that the sample program has been created using a text editor and saved to disk as SAMPLE.ASM. To assemble it, we type

```
MASM/L/N/C SAMPLE;
```

in either uppercase or lowercase letters. MASM displays a copyright message and begins to read the source program. At the end of assembly, MASM displays statistics on the amount of available free space and the number of errors:

```
> MASM/L/N/C SAMPLE;

Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981-1985, 1987. All rights reserved.

 50592 + 271152 Bytes symbol space free

      0 Warning Errors
      0 Severe Errors
```

In this example, the program assembled successfully and no error messages were displayed. Otherwise, the source program would have to be edited and reassembled until all errors were corrected. There are a number of options available when assembling a program; these are listed in Figure 4-3. A few of the more useful ones are shown here with a source program named SAMPLE:

DOS Command Line	Comment
MASM SAMPLE;	Create only an object file named SAMPLE.OBJ.
MASM/L SAMPLE;	Create a listing file named SAMPLE.LST.
MASM/L/N SAMPLE;	Create a listing file but suppress the symbol tables that are usually printed at the end.
MASM/Z SAMPLE;	Create an object file and display source lines containing errors on the screen during assembly.
MASM/C SAMPLE;	Create a cross-reference file named SAMPLE.CRF that contains the line numbers of all names in the program.
MASM/ZI SAMPLE;	Place a list of symbols and line numbers in the object file for use by the CODEVIEW debugger.

Step 2. Link the Program. In the LINK step, the linker program (LINK.EXE) reads the object file as input (SAMPLE.OBJ), creates an executable file (SAMPLE.EXE), and optionally creates a map file (SAMPLE.MAP):

```
> LINK/M SAMPLE;

Microsoft (R) Overlay Linker Version 3.60
Copyright (C) Microsoft Corp 1983-1987. All rights reserved.
```

The map file lists the names of all segments in the program; this becomes important only when writing larger assembler programs or programs that link to

Figure 4-3 MASM Version 5.0 command line options.

Option	Action
/A	Writes segments in alphabetical order
/B*number*	Sets the buffer size
/C	Creates a cross-reference file
/D	Creates a pass 1 listing
/D*symbol*[= *value*]	Defines an assembler symbol
/E	Creates code for emulated floating-point instructions
/H	Lists command-line syntax and all assembler options
/I*path*	Sets the include-file search path
/L	Specifies an assembly-listing file
/ML	Makes names case sensitive
/MU	Converts names to uppercase letters
/MX	Makes public and external names case sensitive
/N	Suppresses tables in the listing file
/P	Checks for impure code
/T	Suppresses messages for successful assembly
/V	Displays extra statistics on the screen
/W{0|1|2}	Sets the error-display level
/Z	Displays erroneous source lines on the screen during assembly
/ZI	Creates a CODEVIEW-compatible object file

high-level languages. The LINK.EXE program supplied with the MASM disk must be used, rather than the linker supplied with DOS.

There is one important linker option, /CO. The command line to prepare an executable file for debugging with CODEVIEW is:

```
link/co sample;
```

The following files and programs were created during the assembly of SAMPLE.ASM:

Name	Comment	Command Used
SAMPLE.ASM	Source program	MASM
SAMPLE.OBJ	Object program	MASM
SAMPLE.LST	Listing file	MASM
SAMPLE.CRF	Cross-reference file	MASM
SAMPLE.EXE	Executable program	LINK
SAMPLE.MAP	Map file	LINK

Step 3. Run the Program. Once the SAMPLE program has been assembled and linked, it may be run by typing its name on the DOS command line. The program doesn't generate any output, so you will probably want to run the program using a debugger.

The SAMPLE.EXE program may be run repeatedly without having to assemble and link it. If we wish to modify the program, however, all changes must be made to the SAMPLE.ASM file. The program must then be assembled and linked again.

EXE Programs

Many assembly language programs are designed to be assembled as EXE programs. There are several reasons why EXE programs are preferred. First, CODEVIEW can display source statements only from EXE programs (not COM programs), making them better suited to debugging. Second, EXE-format assembler programs are more easily converted into subroutines for high-level languages.

The third reason has to do with memory management. To fully use a multitasking operating system, programs must be able to share computer memory and resources. An EXE program is easily able to do this.

Program Template. Figure 4-4 contains a program template to be used when creating new programs. Simplified segment directives are used, and one need only fill in the code and data sections when writing a new program.

Figure 4-4 EXE program template.

```
page ,132
title

dosseg
.model small
.stack 100h

.code
main proc
     mov   ax,@data        ; initialize the DS register
     mov   ds,ax


     mov   ax,4C00h        ; end program
     int   21h
main endp

.data     ;variables here

end main
```

Using a Batch File

Needless to say, it would be time-consuming to have to type all of the commands shown so far every time a program is assembled. Instead, a *batch* file may be used to perform both steps. A useful one is:

```
echo off
cls
masm/l/n %1;
if errorlevel 1 goto end
link %1;
:end
```

The ECHO OFF command tells DOS not to echo the commands to the screen while the batch file is executing. The messages generated directly by the assembler and linker still appear, however. The CLS command clears the screen. The MASM command invokes the Macro Assembler:

```
masm/l/n %1;
```

The batch file automatically halts after the MASM step if any assembly errors occurred by checking the error level returned by the Macro Assembler. An error level of 0 means that no errors occurred. An error level of 1 or more means that there were errors during assembly:

```
if errorlevel 1 goto end
```

The LINK command invokes the linker:

```
link %1;
```

Use a text editor to create this file, and call it CL.BAT (i.e., compile, link) or some other meaningful name. To run the batch file and assemble SAMPLE.ASM, type:

```
cl sample
```

Do not type a file extension—MASM assumes it is .ASM. The name SAMPLE is then substituted for each occurrence of %1 in the batch file, resulting in the following effective commands:

```
echo off
cls
masm/l/n sample;
if errorlevel 1 goto end
link sample;
:end
```

OTHER FILES CREATED DURING THE ASSEMBLY PROCESS

The Listing File

When MASM detects an error while assembling a program, a listing file can be useful. The listing file for the SAMPLE program (Figure 4-5) is called SAMPLE.LST. It contains a wealth of information about the program. Across the top line, the version of MASM used is shown. The program title (Exchange Two Variables) is printed at the top of each page.

The original source program statements are listed, with consecutive line numbers along the left margin. Just to the right are four-digit hexadecimal numbers representing the offset of each statement. An *offset* is the distance (in bytes) from the beginning of a segment to a statement's location.

To the right of each address is a hexadecimal representation of the machine language bytes generated by the instruction. In most cases, the hexadecimal bytes in the program listing are identical to those in the final EXE program. For example, the instruction to move **value1** to AL is on line 13 at address 0005h:

```
13 0005 A0 0000 R      mov al,value1
```

Figure 4-5 The SAMPLE.LST file.

```
Microsoft (R) Macro Assembler Version 5.10
Exchange Two Variables  (SAMPLE.ASM)                    Page   1-1

 1                      page ,132
 2                      title Exchange Two Variables        (SAMPLE.ASM)
 3
 4                      dosseg
 5                      .model small
 6 0100                 .stack 100h
 7
 8 0000                 .code
 9 0000                 main proc
10 0000  B8 ---- R          mov   ax,@data        ; initialize DS register
11 0003  8E D8              mov   ds,ax
12 0005                 swap:
13 0005  A0 0000 R          mov   al,value1       ; load the AL register
14 0008  86 06 0001 R       xchg  al,value2       ; exchange AL with value2
15 000C  A2 0000 R          mov   value1,al       ; store new value of AL
16 000F  B8 4C00            mov   ax,4C00h        ; return to DOS
17 0012  CD 21              int   21h
18 0014                 main endp
19
20 0000                 .data
21 0000  0A             value1   db    0Ah
22 0001  14             value2   db    14h
23
24 0002                 end main
```

The instruction's op code is A0, and the address of **value1** is 0000. The letter *R* shown here indicates that **value1** is a *relocatable* operand, because its location is relative to the start of the data segment.

Other statements are given offset addresses but do not generate any machine code. The statement on line 9, for example, simply marks the beginning of the MAIN procedure.

The Cross-Reference File

A cross-reference listing can be helpful in tracking down the names of procedures, labels, and variables. A cross-reference file may be created during the assembly step, with an extension of CRF (i.e., SAMPLE.CRF). Then the CREF.EXE utility program is run, using SAMPLE.CRF as input, producing the sorted cross-reference listing (SAMPLE.REF). This file may be displayed or printed. The command to create a file called SAMPLE.REF is:

```
cref sample,;
```

The SAMPLE.REF file is shown in Figure 4-6.

Figure 4-6 The SAMPLE.REF file.

```
Microsoft Cross-Reference Version 5.10

  Symbol Cross-Reference   (# definition, + modification) Cref-1

CODE  ..................    8

DATA  ..................   20
DGROUP  ................   10

MAIN  ..................    9#   18    24

STACK  .................    6#    6
SWAP  ..................   12#

VALUE1  ................   13    15+   21#
VALUE2  ................   14+   22#

_DATA  .................   20#
_TEXT  .................    8#

10 Symbols
```

The Map File

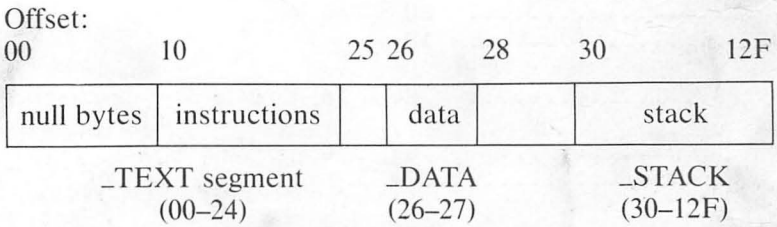
Created during the LINK step, a map file lists information about each of the program segments. In the following SAMPLE.MAP file, the starting and ending addresses and the length of each segment are shown:

```
Start   Stop    Length  Name        Class
00000H  00024H  00025H  _TEXT       CODE
00026H  00027H  00002H  _DATA       DATA
00030H  0012FH  00100H  STACK       STACK

Origin  Group
0002:0  DGROUP

Program entry point at 0000:0010
```

The columns labeled *Name* and *Class* identify each segment name and segment class. The linker automatically combines different segments of the same class into a single segment. This program was created using the .CODE, .DATA, and .STACK directives, so MASM has automatically initialized the name and class of each segment.

The segment attributes shown here are consistent with other Microsoft languages. This map file also shows the order of segments in memory, determined by the DOSSEG directive. When DOSSEG is used, 16 null bytes are automatically placed at the beginning of the code (_TEXT) segment. The *program entry point* in this example tells us that execution will begin at offset 0010h when the program is loaded.

After having looked at both the SAMPLE.LST file and the SAMPLE.MAP file, we can form a picture of the way the program is arranged in memory. The offsets from the beginning of the program are shown at the top; the contents of each part of the program are inside the boxes; the segment names are shown at the bottom:



Alternate Versions of the SAMPLE.ASM Program

Figure 4-7 shows the SAMPLE.ASM program (originally in Figure 4-2) with complete segment directives. This may be assembled by MASM Versions 1.0 through 4.0.

Figure 4-7 The SAMPLE.ASM program, written for MASM versions 1.0 through 4.0.

```
page  ,132

title  Exchange Two Variables  (MASM Version 4.0)

code segment
     assume cs:code,ds:data,ss:stack

main proc

     mov   ax,data            ; initialize DS register
     mov   ds,ax

swap:
     mov   al, value1         ; load the AL register
     xchg  value2,al          ; exchange AL and value2
     mov   value1,al          ; store AL back into value1
     mov   ax,4C00h           ; return to DOS
     int   21h

main endp
code ends

data segment

value1   db    0Ah
value2   db    14h

data ends

stack segment STACK
    db 100h dup(0)
stack ends

end main
```

Figure 4-8 shows the same program written in COM format, which may be assembled by any version of MASM. The linker will display a warning message (*no stack segment*), which may be ignored. After SAMPLE.EXE has been created, you must run the EXE2BIN.EXE program (supplied with DOS) to convert the executable file to a COM file. Therefore, the following commands assemble, link, and convert SAMPLE.ASM to a COM file:

```
masm sample;
link sample;
exe2bin sample sample.com
```

Figure 4-8 The SAMPLE.ASM program, written in COM format.

```
page   ,132
title   Exchange Two Variables    (COM format)

code segment
     assume cs:code,ds:code,ss:code
     org 100h

main proc

 swap:
       mov   al, value1       ; load the AL register
       xchg  value2,al        ; exchange AL and value2
       mov   value1,al        ; store AL back into value1
       mov   ax,4C00h         ; return to DOS
       int   21h

main endp

value1   db    0Ah
value2   db    14h

code ends

end main
```

Programming Tip: The COMMENT Directive

As a matter of habit, assembly language programs should be carefully commented. In longer programs, one often provides comments at the beginning of each program module and procedure. Rather than writing a semicolon at the beginning of each line of comments, you may wish to begin a group of comment lines with the COMMENT directive. The syntax is:

```
COMMENT  delimiter
   text
delimiter     [text]
```

The *delimiter* may be any character not appearing within the text. Additional text on the same line as the last delimiter is also ignored by MASM. For example,

```
COMMENT @

     This program displays all 256 ASCII codes on the screen,
     using the LOOP instruction.

     Last update: 11/01/89
@
```

EQUATES

Equate directives allow constants and literals to be given symbolic names. A constant may be defined at the start of a program and, in some cases, redefined later on.

The Equal-Sign Directive

Known as a *redefinable equate*, the equal-sign directive creates an absolute symbol by assigning the value of a numeric expression to a name. The syntax is:

```
name = expression
```

In contrast to the DB and DW directives, the equal-sign directive allocates no storage. As the program is assembled, all occurrences of *name* are replaced by *expression*. The expression value may be from 0 to 65,535. Examples are as follows:

```
count     = 50          ; Assigns the value 50
prod      = 10 * 5      ; Evaluates an expression
maxint    = 7FFFh       ; Maximum signed value
maxUint   = 0FFFFh      ; Maximum unsigned value
string    = 'XY'        ; Up to two characters allowed
endvalue  = count + 1   ; May use a predefined symbol
```

A symbol defined with the equal-sign directive may be redefined as many times as desired. In the following program excerpt, **count** changes value several times. On the right side of the example, we see how MASM evaluates the constant:

Original	Assembled As
`count=5`	
`mov al,count`	`mov al,5`
`mov dl,al`	`mov dl,al`
`count=10`	
`mov cx,count`	`mov cx,10`
`mov dx,count`	`mov dx,10`
`count=2000`	
`mov ax,count`	`mov ax,2000`

The EQU Directive

The EQU directive assigns a symbolic name to a string or numeric constant. This increases the readability of a program and makes it possible to change constants in a single place at the beginning of a program.

There is one limitation imposed on EQU: A symbol defined with EQU may not be redefined later in the program. At the same time, EQU is very flexible:

- A name may be assigned to a string, allowing the name to be replaced by the string wherever it is found. For example, the **prompt1** string is defined here using the **continue** constant:

```
continue equ 'Do you wish to continue (Y/N)?'
.
.
.
prompt1 db continue
```

- An *alias,* which is a name representing another predefined symbol, may be created. For example:

Original			Assembled As		
move	equ	mov			
address	equ	offset			
.					
.					
move	bx,address	value1	mov	bx,offset	value1
move	al,20		mov	al,20	

Expressions involving integers evaluate to numeric values, but floating-point values evaluate to text strings. Also, beginning with MASM Version 5.0, string equates may be enclosed in angle brackets (< and >) to force their evaluation as string expressions. This eliminates ambiguity on the part of the assembler when assigning the correct type of value to a name:

Example			Type of Value
maxint	equ	32767	Numeric
maxuint	equ	0FFFFh	Numeric
count	equ	10 * 20	Numeric
float1	equ	<2.345>	String
msg1	equ	'Press [Enter] to Continue'	String
tl	equ	<top-line>	String
bpt	equ	<byte ptr>	String

OPERATORS AND EXPRESSIONS

An expression is a combination of operators and operands that is converted by the assembler into a single value. The size of the value must not be greater than the size of the destination operand, which is either 8 or 16 bits. MASM performs arithmetic and logical operations at assembly time, not at runtime. The result is calculated and placed in memory at the current program location. A list of the most frequently used operators follows.

Frequently Used Operators

Arithmetic Operators (+, −, *, /, MOD)
Addition, subtraction, multiplication, division, modulus arithmetic.

Boolean Operators (NOT, AND, OR, XOR)
Boolean operations.

Index Operator ([])
Adds the value between the square brackets to the operand outside. For example, the expression 4[2] is equal to 6.

OFFSET Operator
Returns the offset address of a variable or label.

SEG Operator
Returns the segment value of a variable, label, or any other symbol.

PTR Operator
Forces a variable or label to be given a specified type (i.e., BYTE, WORD, DWORD, QWORD, TBYTE, NEAR, or FAR).

Segment Override (CS:, DS:, ES:, SS:, segname:)
Forces the CPU to calculate the offset of a variable or label from the named segment.

Shift Operators (SHL, SHR)
Shifts an operand left or right *count* number of bits. The operand may be a single value or an expression.

SHORT Operator
Identifies a label as being within the range of -128 to +127 bytes from the current location. Used often with the JMP instruction.

Arithmetic Operators

Arithmetic operators may be used only with integers, with the exception of unary plus (+) and minus (−), which may also be used with real numbers. Examples of arithmetic expressions follow.

Example	Comment
`20 * 5`	Value is 100
`-4 + -2`	Value is -6
`count+2`	**Count** is a constant
`31 MOD 6`	Remainder is 1
`6/4`	Value is 1
`'2'-30h`	Value is 2

Each of the operators has a *precedence* level, meaning that it will be evaluated before another operator of lower precedence. The set of arithmetic operators in order of precedence is:

Operator	Description
()	Parentheses
+	Positive sign (unary)
−	Negative sign (unary)
*	Multiplication
/	Division
MOD	Modulus arithmetic
+	Addition
−	Subtraction

The order of operations is shown for each of the following examples:

Example	Order of Operations
3 + 2 * 5	Multiplication, addition
count / 5 MOD 3	Division, MOD
+4 * 3 − 1	Positive sign, multiplication, subtraction
(1000h − 30h) * 4	Subtraction, multiplication
−((count MOD 5) + 2) * 4	MOD, addition, negative sign, multiplication

Boolean Operators

The AND, OR, NOT, and XOR operators may be used in assembler expressions, which are evaluated at assembly time. Assuming that **status** and **count** have been declared using equates, the following expressions are valid:

```
status and 0Fh
(count or 10h) and 7Fh
not status
(count * 20) and 11001010b
```

OFFSET Operator

The OFFSET operator returns the distance of a label or variable from the beginning of its segment. The destination operand must be a 16-bit register, as in the following example:

```
mov   bx,offset count  ; BX points to count
```

We say that BX points to **count** because it contains the variable's address. When a register or variable holds an address, we call it a *pointer.* It may be easily manipulated to point to subsequent addresses.

SEG Operator

The SEG operator returns the segment value of a label or variable. We usually use it to place an address in a segment register. In the following example, the segment value must be transferred via AX because an immediate value may not be moved to a segment register:

```
mov  ax,seg array   ; set DS to segment of array
mov  ds,ax
```

PTR Operator

The PTR operator forces a variable or label to be used as if it were of a specified type. The syntax is:

type PTR

The various types of PTR operators follow. They are related to the characteristics of a memory variable or to the distance attribute of a label:

Operator	Description
Byte ptr	Byte operand
Word ptr	Word operand
Dword ptr	Doubleword (32-bit) operand
Qword ptr	Quadword (64-bit) operand
Tbyte ptr	Ten-byte (80-bit) operand
Near ptr	Within the current segment
Far ptr	Outside the current segment

PTR specifies the exact size of an operand, as in the following examples:

```
mov   ax,word ptr var1        ; Var1 is a word operand
mov   bl,byte ptr var2        ; Var2 is a byte operand
inc   dword ptr newaddress    ; Doubleword operand
call  far ptr clear_screen    ; Call a far subroutine
```

PTR can also override an operand's default type. Suppose the variable **wordval** contains 1026h, and we want to move its low byte into AL. Because word values are always stored in memory with the bytes in reverse order, the low byte is stored at the lowest address:

```
          mov   al,byte ptr wordval   ; al = 26h
81        .
82        .
83    wordval   dw  1026h             ; stored as 26 10
84
```

Operands with Displacements

A particularly good use of the addition and subtraction operators (+, −) is to access a list of values. The addition operator adds to the *offset* of a label. For example, an instruction to move a byte at location **array** into BL is:

```
mov  bl,array
```

To access the byte following **array**, we add 1 to the label:

```
mov  bl,array+1
```

In the following series of instructions, the first byte of **array** is moved to AL, the second byte to BL, the third byte to CL, and the fourth byte to DL. The value of each register after the moves is shown on the right:

```
        mov  al,array     ; AL = 0Ah
        mov  bl,array+1   ; BL = 0Bh
        mov  cl,array+2   ; CL = 0Ch
        mov  dl,array+3   ; DL = 0Dh
        .
        .
array db 0Ah,0Bh,0Ch,0Dh
```

Although less typical, we may wish to subtract from a label's offset. In the following example, the label **endlist** is 1 byte beyond the last byte in **list**. To move the last byte in **list** to AL, we write:

```
        mov  al,endlist-1    ; move 5 to AL
        .
        .
list    db 1,2,3,4,5
endlist db 0
```

In a list of 16-bit values, you add 2 to the offset of any element to point to the next element. This is done here, using an array called **wvals**:

```
        mov   ax,wvals      ; AX = 1000h
        mov   bx,wvals+2    ; BX = 2000h
        mov   cx,wvals+4    ; CX = 3000h
        mov   dx,wvals+6    ; DX = 4000h
        .
        .
wvals dw 1000h,2000h,3000h,4000h
```

The addition or subtraction operator may be combined with the OFFSET operator to calculate the location of an item in a list. Assuming that the offset of **bytelist** is 0h in the following example, we will end up with the following values in AX and BX:

```
        mov   ax,offset bytelist      ; AX = 0000h
        mov   bx,offset bytelist+4    ; BX = 0004h
        .
        .
bytelist db '0123456789'
```

Programming Tip: Other MASM Operators

A number of other operators available in MASM are briefly described here. Consult the MASM manual for more complete information.

HIGH returns the high 8 bits of a constant expression.

LOW returns the low 8 bits of a constant expression.

LENGTH returns the number of byte, word, dword, qword, or tenbyte elements in a variable. This is meaningful only if the variable was initialized with the DUP operator.

MASK returns a bit mask for the bit positions in a field within a variable. A *bit mask* preserves just the important bits, setting all others equal to zero. The variable must be defined with the RECORD directive.

EQ, NE, LT, LE, GT, GE (relational operators) return a value of 0FFFFh when a relation is true or 0 when it is false.

SEG returns the segment value of an expression, whether it be a variable, a segment/group name, a label, or any other symbol.

SIZE returns the total number of bytes allocated for a variable. This is calculated as the LENGTH multiplied by the TYPE.

Structure Field-Name (.) identifies a field within a predefined structure by adding the offset of the field to the offset of the variable. The format is:

variable.field

THIS creates an operand of a specified type at the current program location. The type may be any of those used with the PTR operator or the LABEL directive.

TYPE returns a numeric value representing either the size of a variable or the type of a label. For example, the TYPE of a word variable is 2.

.TYPE returns a byte that defines the mode and scope of an expression. The result is bit mapped and used to show whether a label or variable is program related, data related, undefined, or external in scope.

WIDTH returns the number of bits of a given field within a variable. The variable must be defined using the RECORD directive.

DEBUGGING WORKSHOP

Most syntax errors result either from the incorrect use of assembler directives or from the use of invalid instruction operands. Several programs containing common errors are presented here. The general types of errors made at the beginning level are:

- Missing or misplaced .CODE, .DATA, and .STACK directives.
- Missing or misplaced PROC and ENDP directives.
- Missing END directive or a label missing after the END directive.
- Mismatching operand sizes.

When MASM displays an error message, it lists the name of the source program, the line number (from the listing file) in parentheses, the error number, and a description of the error. For example:

```
PROG1.ASM (7): warning A4031: Operand types must match
```

The name of the source file is PROG1.ASM, and the error occurred on line 7. The error number (A4031) refers you to a more complete explanation in the MASM manual. The message ("Operand types must match") says that a statement tried to mix operands of two different sizes. Although this is a warning message, the problem should be resolved before linking and running the program. Otherwise, MASM may generate incorrect object code.

In the sample error messages given in this chapter, we have omitted the file name, so the line number that caused the error is at the beginning of the error message.

Open Procedures. The error message "Open procedures" occurs when an ENDP directive is not found to mark the end of a procedure. The following program demonstrates this:

```
1:    title Error Example
2:    dosseg
3:    .model small
4:    .stack 100h
5:    .code
6:    main proc
7:         mov ax,4C00h
8:         int 21h
9:                                   <- ENDP directive missing
10:   end main

Error message:  Open procedures: MAIN
```

Operand Sizes and Addressing Errors

Recent versions of MASM perform stronger type checking on memory operands than do earlier versions. By *type checking*, we mean ensuring that the sizes of both operands match. As a result, many older assembly language source programs must be modified to assemble correctly. Such errors are usually corrected by using the BYTE PTR or WORD PTR operators to identify the attributes of certain operands. This may seem to be an annoyance, but it often helps us avoid subtle logic errors.

Mismatching Operand Sizes. In the following program, the 8-bit size of **value1** does not match the 16-bit size of AX, and **value2** does not match the size of AH. This is probably the most common of all syntax errors, made by beginner and expert alike:

```
1:    title Mismatching Operand Sizes
2:    dosseg
3:    .model small
4:    .stack 100h
5:    .code
6:    main  proc
7:          mov   ax,@data
8:          mov   ds,ax
9:          mov   ax,value1
10:         mov   ah,value2
11:         mov   ax,4C00h
12:         int   21h
13:   main  endp
14:
15:   .data
16:   value1  db    0Ah
17:   value2  dw    1000h
18:   end     main

  (9): warning A4031: Operand types must match
 (10): warning A4031: Operand types must match
```

Figure 4-9 Program with numerous syntax errors.

```
1:   title  Miscellaneous Errors
2:   dosseg
3:   .model small
4:   .stack 100h
5:   .code
6:   main  proc
7:         mov   ax,@data      ; initialize DS register
8:         mov   ds,ax
9:         mov   ax,bx * cx
10:        mov   bx,value1 * 2
11:        mov   byte ptr value3, al
12:        mov   cx,ax
13:        mov   cs,ds
14:        mov   ax,4C00h
15:        int   21h
16:  main  endp
17:  .data
18:  value1   db    0Ah
19:  value2   db    14h
20:  value3   dw    1000h
21:  end    main

          (9): error A2042: Constant expected
         (10): error A2042: Constant expected
         (13): error A2059: Illegal use of CS register
```

These errors may be corrected by adjusting the instruction operands to the correct sizes of the memory variables. We can rewrite lines 9 and 10 as follows:

```
mov  al,value1
mov  ax,value2
```

Miscellaneous Errors. In the program shown in Figure 4-9, a number of different mistakes were made. On line 9 the expression (bx * cx) is interpreted by MASM as an immediate operand. The values of BX and CX are known only at runtime, so MASM prints an error message. On line 10, the expression (value1 * 2) appears to be the product of two constants. When MASM encounters the declaration of **value1** (as a variable) later in the program, it declares the instruction illegal. The error message for line 13 reminds us that CS may not be used as a destination operand (nor may IP).

Programming Tip: Creating Your Own Segments

You can use the SEGMENT and ENDS directives to create additional segments. You might want to create a 64K buffer the size of an entire segment. Another use might be

to isolate data inside each program module. Virtually any number of segments may appear in a single program.

To create a segment, begin with the SEGMENT directive. This is followed by instructions or data, which in turn are followed by the ENDS directive:

```
my_seg segment
   var1   db  10
   var2   dw  2000h
   .
   .
my_seg ends
```

Caution: You must set DS or ES to the desired segment location before trying to access a variable in the segment. The following program excerpt illustrates the use of two programmer-defined segments, along with the default data segment. The SEG operator may be used to obtain a variable's segment value, or the name of the segment itself may be used to obtain its location:

```
.code
mov   ax,seg value1    ; use the SEG operator
mov   ds,ax
mov   al,value1        ; result: AL = 1
mov   ax,seg2          ; use the segment's name
mov   ds,ax
mov   al,value2        ; result: AL = 2
mov   ax,@data         ; use predefined @DATA value
mov   ds,ax
mov   al,value3        ; result: AL = 3
.
.
seg1 segment
  value1  db   1
seg1 ends

seg2 segment
  value2  db   2
seg2 ends

.data
  value3  db   3
```

POINTS TO REMEMBER

A debugger should always be used when learning about data storage and assembly language instructions and when tracing executable programs.

The assembly process involves converting a source program to an executable program. Several other files are created at assembly time: a *listing* file containing a list of the program's source code, a *cross-reference* file containing the names of labels and other names, and a *map file* created by the linker identifying segment names in the program.

A *batch* file can make the assembly and link steps easier. A batch file is a short program containing commands that would otherwise have to be typed at the DOS command prompt.

REVIEW QUESTIONS

1. Why could a listing file not be assembled and linked?
2. Which three files could be created by MASM if the program PROJ1.ASM were assembled?
3. Name two files created during the LINK step.
4. What is the maximum length of an assembler statement?
5. If the program TEST.ASM has been designed as an EXE program, can the program TEST.OBJ be executed successfully?
6. What does the MODEL directive identify?
7. Name six different types of memory models.
8. After the cross-reference file (extension CRF) has been created, can it be printed?
9. What is the exact meaning of the following directive?

```
PAGE 55,132
```

10. During the first pass by the assembler, is any machine code generated for each instruction?
11. What happens when the length of an operand changes between the assembler's first and second passes?
12. In a program consisting of separately compiled modules, what is the assembler's name for a reference to a label outside the current module?
13. Which directive controls the segment order in a program?
14. If a variable name is coded in uppercase letters, must all references to the variable also be in uppercase?

15. If a program had two procedures, how could we be sure that one was executed before the other?

16. If the following statements were used in a batch file, would a program assemble and link properly?

```
masm %1,,,;
if errorlevel 0 goto end
link %1;
:end
```

17. Which distance attribute for PROC is the default?

18. Which directive marks the end of a procedure?

19. What is the significance of the label used with the END directive?

20. Would the following command (typed from DOS) generate an object file and a listing file?

```
masm test;
```

21. Which file(s) will be generated by the following LINK command?

```
link test;
```

22. When the listing file displays the *offset* of a statement, what does the offset indicate?

23. In the following excerpt from a program listing, what do the values B8 4C00 represent?

```
12 010E B8 4C00    mov  ax,4C00h
```

24. In the following program, do you think that the instruction at **label2** will be executed? Use a debugger to confirm the result.

```
title
dosseg
.model small
.stack 100h
.code
main proc
label1:
    mov   ax,20
    mov   ax,4C00h
    int   21h
label2:
    mov   ax,10
main endp
end main
```

25. Correct any erroneous lines in the following program, based on the error messages printed by MASM.

```
 1:   title Find The Errors
 2:   segdos
 3:   .model
 4:   .stack 100h
 5:
 6:   main proc
 7:        mov  bl,val1
 8:        mov  cx,val2
 9:        mov  ax,4C00h
10:        int  21h
11:   main endp
12:
13:   .data
14:   val1   db  10h
15:   val2   dw  1000h
16:   end  main
```

MASM Error Messages

```
(2):  error A2105: Expected: instruction or directive
(6):  error A2062: Missing or unreachable CS
(7):  error A2086: Data emitted with no segment
(8):  error A2086: Data emitted with no segment
(9):  error A2086: Data emitted with no segment
(10): error A2086: Data emitted with no segment
```

26. The following program contains numerous errors. Write a correction next to each erroneous statement, and add any new lines you deem necessary.

```
 1:   title Find The Errors
 2:   dosseg
 3:   .model small
 4:   .stack 100h
 5:   .data
 6:        mov  ax,value1
 7:        mov  bx,value2
 8:        inc  bx,1
 9:        int  21h
10:        mov  4C00h,ax
11:   main  endp
12:
13:   value1   0Ah
14:   value2   1000h
15:   end main
```

MASM Error Messages

```
(6):  error A2009:   Symbol not defined: VALUE1
(7):  error A2009:   Symbol not defined: VALUE2
(8):  warning A4001: Extra characters on line
(10): error A2056:   Immediate mode illegal
```

```
(11): error A2000:   Block nesting error
(13): error A2105:   Expected: instruction or directive
(14): error A2105:   Expected: instruction or directive
(15): error A2009:   Symbol not defined: MAIN
```

27. Assume that **array1** is located at offset address 0120. As each instruction is executed, fill in the value of the operand shown on the right side:

```
.code
  mov     ax,@data
  mov     ds,ax
  mov     ax,ptr1          ; a. AX =
  mov     bx,array1        ; b. BX =
  xchg    ax,bx            ; c. AX =
  sub     al,2             ; d. AX =
  mov     ptr2,bx          ; e. ptr2 =
.data
  array1   dw  10h,20h
  ptr1     dw  array1
  ptr2     dw  0
```

28. Assume that **val1** is located at offset 0120h and that **ptr1** is located at offset 0122h. As each instruction is executed, fill in the value of the operand shown on the right-hand side.

```
.code
  mov     ax,@data
  mov     ds,ax
  mov     ax,0
  mov     al,byte ptr val1    ; a. AX =
  mov     bx,ptr1             ; b. BX =
  xchg    ax,bx               ; c. BX =
  sub     al,2                ; d. AX =
  mov     ax,ptr2             ; e. AX =
.data
  val1     dw  3Ah
  ptr1     dw  val1
  ptr2     dw  ptr1
```

PROGRAMMING EXERCISES

Exercise 1 **Define and Display 8-Bit Numbers**

Write, assemble, and test a program to do the following:

Use the DB directive to define the following list of numbers with the label **array**:

31h, 32h, 33h, 34h

Write instructions to load each number into DL and display it on the console. (The following instructions display the byte in DL on the console:)

```
mov  ah,2
int  21h
```

Explain why the output on the screen is a series of single digits:

```
1234
```

Exercise 2 **Add a List of 16-Bit Numbers**

Write, assemble, and test a program to do the following: Use the DW directive to define the following list of 16-bit numbers:

```
1000h, 2000h, 1234h
```

Add each number to the AX register and move the sum to a variable called **sum**. Move the offset of the second number to BX, using the OFFSET operator. Move the third number into CX, using BX as a register indirect operand.

Exercise 3 **Using the = and EQU Directives**

Write, assemble, and test a program to do the following:

Define a variety of constants, using both the = and EQU directives. Include character strings, integers, and arithmetic expressions. Be sure to use the +, -, *, /, and MOD operators.

Move the constants to both 8-bit and 16-bit registers, trying to anticipate in advance which moves will generate syntax errors.

Exercise 4 **Using PTR and OFFSET**

Assemble the following program using MASM and trace the executable program using a debugger. Print a trace of the program. Before you run the program, write down what you think each register will contain after it is changed by an instruction. Assume that **first** is located at offset 0h:

```
        mov  al,byte ptr first + 1      ; AL =
        mov  bx,word ptr second + 2     ; BX =
        mov  dx,offset first + 2        ; DX =
        mov  ax,4C00h
        int  21h
        .
        .
first   dw 1234h
second  dw 16386
third   db 10,20,30,40 ; (decimal values)
```

Exercise 5 **Arithmetic Sums**

Write, assemble, and trace a program that finds the sum of three byte variables and places it in memory at location **sums**. Find the sum of three word variables and place it in memory at location **sums+2**. Print out a dump of the variables.

Use the following variables:

```
byte_array  db  10h, 20h, 30h
word_array  dw  1000h, 2000h, 3000h
sums        dw  0, 0
```

Exercise 6 **Batch File for CODEVIEW**

Using the sample batch file presented in this chapter for compiling and linking, create your own for use with CODEVIEW. Test the file, and make sure that it works. You may want to have the batch file run CODEVIEW automatically.

ANSWERS TO REVIEW QUESTIONS

1. A listing file contains line numbers and a hexadecimal representation of the machine language program generated by the assembler. This would generate many syntax errors if it were used as input to the assembler.

2. PROJ1.LST, PROJ1.OBJ, and PROJ1.CRF.

3. An EXE file and a MAP file.

4. 128 characters.

5. No, it must be linked before it can be run.

6. The MODEL directive identifies the memory model to be used in assembling the program.

7. The models are tiny, small, medium, compact, large, and huge.

8. No, the CREF.EXE program must read the CRF file and generate a REF file, which may then be printed.

9. When the listing file is generated, the page length will be 55 lines, and the line length will be 132 characters.

10. Yes, the assembler generates as much machine code as possible, with the exception of offsets for variables that are forward referenced.

11. A phase error is generated, because all subsequent instructions and labels are shifted to new addresses.

12. A label outside the current module is an *unresolved external* whose address will not be known until the program is linked.

13. The DOSSEG directive.

14. No, MASM does not distinguish between uppercase and lowercase letters.

15. The procedure identified by the END directive would be executed first.

16. No, because the ERRORLEVEL 0 would return TRUE even when a program assembled successfully, causing the statement

```
GOTO END
```

to be executed. As a result, no program would ever be linked.

17. NEAR.

18. ENDP.

19. It refers to the location of the first executable instruction in the program, often known as the *entry point*.

20. Only an object file, TEST.OBJ.

21. Only an executable file, TEST.EXE.

22. The offset indicates the distance (in bytes) between the statement and the beginning of the segment in which the statement resides.

23. B8 is the op code for moving immediate data to the AX register, and 4C00 is the immediate data being moved.

24. The instruction at **label2** will not be executed because it is after the program terminate instructions (mov ax,4C00h; int 21h).

25. The errors are corrected in the following listing:

```
title Find The Errors
dosseg
.model small
.stack 100h
.code
```

```
main  proc
      mov  ax,@data
      mov  ds,ax
      mov  bl,val1
      mov  cx,val2
      mov  ax,4C00h
      int  21h
main  endp

.data
val1    db 10h
val2    dw 1000h
end main
```

26. The errors are corrected in the following listing:

```
title Find The Errors
dosseg
.model small
.stack 100h
.code
main  proc
      mov   ax,@data
      mov   ds,ax
      mov   al,value1
      mov   bx,value2
      inc   bx
      mov   ax,4C00h
      int   21h
main  endp
.data

value1   db 0Ah
value2   dw 1000h
end main
```

27. a. 0120
    b. 0010
    c. 0010
    d. 000E
    e. 0120

28. a. 003A
    b. 0120
    c. 003A
    d. 011E
    e. 0122