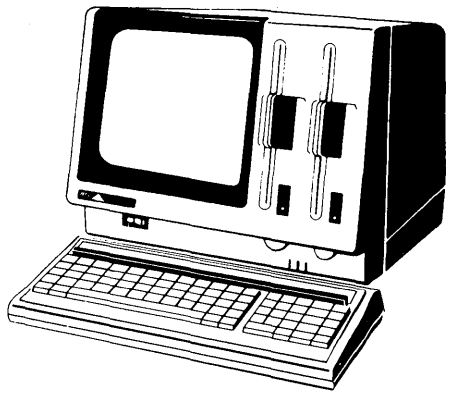
Advanced
Personal Computer
TM



CP/M-86 User/Programmer's Guide

NEC
NEC Information Systems, Inc.
819-000100-4001 Rev. 01
8-83

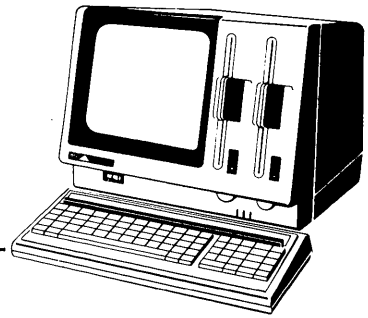
Contents

Contents (cont'd)

Contents (cont'd)

Illustrations

Tables

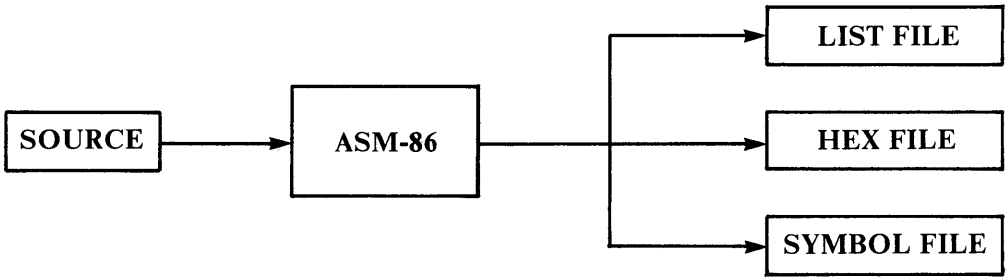


Chapter 1

Introduction

ASSEMBLER OPERATION

ASM-86 processes an 8086 assembly language source file in three passes and produces three output files, including an 8086 machine language file in hexadecimal format. This object file may be in either Intel or Digital Research hex format, both of which are described in Appendix C. ASM-86 typically produces three output files from one input source file as shown in Figure 1-1.



filename.A86 — contains source
filename.LST — contains listing
filename.H86 — contains assembled program in hexadecimal format
filename.SYM — contains all user-defined symbols

Figure 1-1 ASM-86 Source and Object Files

Figure 1-1 also lists ASM-86 filename extensions. ASM-86 accepts a source file with any three letter extension. However, if the extension is omitted from the command line entry, ASM-86 looks for the specified file name with the extension .A86 in the directory. If the file has an extension other than .A86 or has no extension at all, ASM-86 returns an error message.

The other extensions listed in Figure 1-1 identify ASM-86 output files. The .LST file contains the assembly language listing with any error messages. The .H86 file contains the machine language program in either Digital Research or Intel hexadecimal format. The .SYM file lists any user-defined symbols.

To invoke ASM-86, enter a command using the following form:

ASM86 *source filename* {$ *optional parameters*}

Specifications for the optional parameters, which generally affect the assembly output, are described in the next section. The source file is specified using the following form:

{*d:*}*filename.*{*filetype*}

where

d:	is a valid drive letter specifying the source file's location. Not needed if source is on current drive.
filename	is a valid CP/M filename of 1 to 8 characters.
filetype	is a valid file extension of 1 to 3 characters, usually A86.

The following are examples of valid ASM-86 commands:

```
A>ASM86 B:BIOS88
A>ASM86 BIOS88.A86  $FI  AA  HB  PB  SB
A>ASM86 D:TEST
```

Once invoked, ASM-86 responds with the message:

```
CP/M 8086 ASSEMBLER VER x.x
```

where x.x is the ASM-86 version number. ASM-86 then attempts to open the source file. If the file does not exist on the designated drive, or does not have the correct extension as described above, the assembler displays the message:

```
NO FILE
```

If an invalid parameter is given in the optional parameter list, ASM-86 aborts and displays the message:

```
PARAMETER ERROR
```

After opening the source file, the assembler creates the output files. Usually these files are located on the current drive, but they may be redirected using the optional

parameters, or by a drive specification in the source file name. In the latter case, ASM-86 directs the output files to the drive specified in the source file name.

During assembly, ASM-86 aborts if an error condition such as a full disk or symbol table overflow is detected. When ASM-86 detects an error in the source file, it places an error message line in the listing file in front of the line containing the error.

Each error message has a number and gives a brief explanation of the error. Appendix H lists ASM-86 error messages. When the assembly is complete, ASM-86 displays the message:

END OF ASSEMBLY. NUMBER OF ERRORS: n

The system also prints a message indicating the percentage of symbol table space that was used in the assembly.

OPTIONAL RUNTIME PARAMETERS

The dollar sign character ($) introduces an optional string of runtime parameters in the command line. A parameter is a single letter that must be followed by another single letter argument indicating the device name specification. The parameters are shown in Table 1-1.

Table 1-1 Runtime Parameter Summary

PARAMETER	TO SPECIFY	VALID ARGUMENTS
A	source file device	A ... H, X, Y, Z
H	hex output file device	A ... H, X, Y, Z
P	list file device	A ... H, X, Y, Z
S	symbol file device	A ... H, X, Y, Z
F	format of hex output file	I, D

All parameters are optional, and can be entered in the command line in any order. Enter the dollar sign only once at the beginning of the parameter string. Spaces may separate parameters, but are not required. No space is permitted, however, between a parameter and its device name.

A device name must follow parameters A, H, P and S. Use the following device names:

A, B, C, D or X, Y, Z

Device names A through D respectively specify floppy diskette drives A through D. Device names E and F specify the two partitions in hard disk unit 0. Devices G and H specify the two partitions in hard disk unit 1. X specifies the user console (CON:), Y specifies the line printer (LST:), and Z suppresses output (NUL:).

If output is directed to the console, it may be temporarily stopped at any time by pressing CTRL-S. Restart the output by pressing CTRL-S a second time or by pressing any other character.

The F parameter requires either an I or a D argument. When I is specified, ASM-86 produces an object file in Intel hex format. A D argument requests Digital Research hex format. Appendix C discusses these formats in detail. If the F parameter is not entered in the command line, ASM-86 produces Digital Research hex format.

The following table gives examples of valid ASM-86 command line entries.

Table 1-2 Runtime Parameter Examples

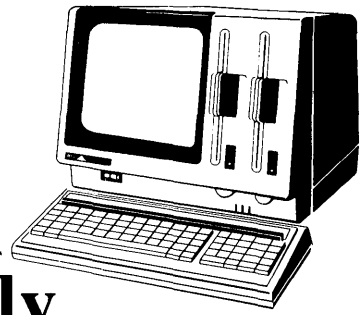
COMMAND LINE	RESULT
ASM86 IO	Assemble file IO.A86, produce IO.HEX, IO.LST and IO.SYM, all on the default drive.
ASM86 IO.ASM $ AD SZ	Assemble file IO.ASM on device D, produce IO.LST and IO.HEX, no symbol file.
ASM86 IO $ PY SX	Assemble file IO.A86, produce IO.HEX, route listing directly to printer, output symbols on console.
ASM86 IO $ FD	Produce Digital Research hex format.
ASM86 IO $ FI	Produce Intel hex format.

ABORTING ASM-86

You may abort ASM-86 execution at any time by pressing any key on the APC keyboard. When a key is pressed, ASM-86 responds with the question:

```
USER BREAK. OK(Y/N)?
```

Enter Y to abort the assembly and return to the operating system. Enter N to continue the assembly.



Chapter 2

Elements of ASM-86 Assembly Language

ASM-86 CHARACTER SET

ASM-86 recognizes a subset of the ASCII character set. The valid characters are the alphanumerics, special characters, and nonprinting characters shown below:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
+ - * / = ( ) [ ] ; ' . ! , _ : @ $
```

space, tab, carriage return, and line feed

Lowercase letters are treated as uppercase except within strings. Only alphanumerics, special characters, and spaces may appear within a string.

TOKENS AND SEPARATORS

A token is the smallest meaningful unit of an ASM-86 source program, much as a word is the smallest meaningful unit of an English composition. Adjacent tokens are commonly separated by a blank character or space. Any sequence of spaces may appear wherever a single space is allowed. ASM-86 recognizes horizontal tabs as separators and interprets them as spaces. Tabs are expanded to spaces in the list file. The tab stops are at each eighth column.

DELIMITERS

Delimiters mark the end of a token and add special meaning to the instruction, as opposed to separators, which merely mark the end of a token. When a delimiter is present, separators need not be used. However, separators entered after delimiters can make your program easier to read.

Table 2-1 describes ASM-86 separators and delimiters. Some delimiters are also operators and are explained in greater detail later in the chapter.

Table 2-1 Separators and Delimiters

CHARACTER	NAME	USE
20H	space	separator
09H	tab	legal in source files, expanded in list files
CR	carriage return	terminates source lines
LF	line feed	legal after CR; if within source lines, it is interpreted as a space
;	semicolon	starts comment field
:	colon	identifies a label, used in segment override specification
.	period	forms variables from numbers
$	dollar sign	notation for "present value of location pointer"
+	plus	arithmetic operator for addition
-	minus	arithmetic operator for subtraction
*	asterisk	arithmetic operator for multiplication
/	slash	arithmetic operator for division
@	at sign	legal in identifiers
_	underscore	legal in identifiers

Table 2-1 Separators and Delimiters (cont'd)

CHARACTER	NAME	USE
!	exclamation point	logically terminates a statement, thus allowing multiple statements on a single source line
'	apostrophe	delimits string constants

CONSTANTS

A constant is a value known at assembly time that does not change while the assembled program is executed. A constant may be either an integer or a character string.

Numeric Constants

A numeric constant is a 16-bit value in one of several bases. The base, called the radix of the constant, is identified by a radix indicator that follows the numeric constant. The radix indicators are shown in Table 2-2.

Table 2-2 Radix Indicators for Constants

INDICATOR	CONSTANT TYPE	BASE
B	binary	2
O	octal	8
Q	octal	8
D	decimal	10
H	hexadecimal	16

ASM-86 assumes that any numeric constant not terminated with a radix indicator is a decimal constant. Radix indicators may be represented in either uppercase or lowercase.

A constant is thus a sequence of digits followed by an optional radix indicator, where the digits are in the range allowed for the radix. Binary constants must be composed of 0s and 1s. Octal digits range from 0 to 7; decimal digits range from 0 to 9. Hexadecimal constants contain decimal digits as well as the hexadecimal digits A (10D), B (11D), C (12D), D (13D), E (14D), and F (15D). Note that the leading

character of a hexadecimal constant must be either a decimal digit so that ASM-86 cannot confuse a hex constant with an identifier, or leading 0 to prevent this problem. The following are valid numeric constants:

1234	1234D	1100B	1111000011110000B
1234H	0FFEH	33770	13772Q
33770	0FE3H	1234d	0ffffh

Character Strings

ASM-86 treats an ASCII character string delimited by apostrophes as a string constant. All instructions accept only one- or two-character constants as valid arguments. Instructions treat a one-character string as an 8-bit number. A two-character string is treated as a 16-bit number with the value of the second character in the low-order byte, and the value of the first character in the high-order byte.

The numeric value of a character is its ASCII code. ASM-86 does not translate case within character strings, so both uppercase and lowercase letters can be used. Note that only alphanumerics, special characters, and spaces are allowed within strings.

A DB assembler directive is the only ASM-86 statement that may contain strings longer than two characters. The string may not exceed 255 bytes. To include an apostrophe to be printed within the string, enter it twice. ASM-86 interprets the two keystrokes (") as a single apostrophe. Table 2-3 shows valid strings and how they appear after processing.

Table 2-3 String Constant Examples

ENTERED AS:	INTERPRETED AS
'a'	a
'Ab' 'Cd'	Ab'Cd
'I like CP/M'	I like CP/M
''''	'
'ONLY UPPER CASE'	ONLY UPPER CASE
'only lower case'	only lower case

ASM-86 flags an incomplete string, a string without a terminating quotation mark, as an error.

IDENTIFIERS

Identifiers are character sequences that have a special, symbolic meaning to the assembler. All identifiers in ASM-86 must obey the following rules:

- The first character must be alphabetic (A,...Z, a,...z).
- Any subsequent characters can be either alphabetical or numeric (0, 1,.....9). ASM-86 ignores the special characters @ and _, but they are still legal. For example, a_b becomes ab.
- Identifiers may be any length up to the limit of the physical line.

Two types of identifiers can be used. *Keywords* have predefined meanings to the assembler. *Symbols* are defined by the user. The following are all valid identifiers:

```
NOLIST
WORD
AH
Third_street
How_are_you_today
variable@number@1234567890
```

Keywords

A keyword is an identifier that has a predefined meaning to the assembler. Keywords are reserved; the user cannot define an identifier identical to a keyword. For a complete list of keywords, see Appendix D.

ASM-86 recognizes five types of keywords: instructions, directives, operators, registers, and predefined numbers. 8086 instruction mnemonic keywords and the actions they initiate are defined in Chapter 4. Directives are discussed in Chapter 3. Operators are described later in this chapter. Table 2-4 lists the ASM-86 keywords that identify 8086 registers.

Three keywords are predefined numbers: BYTE, WORD, and DWORD. The values of these numbers are 1, 2 and 4, respectively. In addition, a Type attribute is associated with each of these numbers. The Type attribute of the keyword is equal to the keyword's numeric value. The next section provides a complete discussion of Type attributes.

Table 2-4 Register Keywords

REGISTER SYMBOL	SIZE	NUMERIC VALUE	MEANING
AH	1 byte	100 B	Accumulator-High-Byte
BH	1 "	111 B	Base-Register-High-Byte
CH	1 "	101 B	Count-Register-High-Byte
DH	1 "	110 B	Data-Register-High-Byte
AL	1 "	000 B	Accumulator-Low-Byte
BL	1 "	011 B	Base-Register-Low-Byte
CL	1 "	001 B	Count-Register-Low-Byte
DL	1 "	010 B	Data-Register-Low-Byte
AX	2 bytes	000 B	Accumulator (full word)
BX	2 "	011 B	Base-Register "
CX	2 "	001 B	Count-Register "
DX	2 "	010 B	Data-Register "
BP	2 "	101 B	Base Pointer
SP	2 "	100 B	Stack Pointer
SI	2 "	110 B	Source Index
DI	2 "	111 B	Destination Index
CS	2 "	01 B	Code-Segment-Register
DS	2 "	11 B	Data-Segment-Register
SS	2 "	10 B	Stack-Segment-Register
ES	2 "	00 B	Extra-Segment-Register

Symbols and Their Attributes

A symbol is a user-defined identifier with attributes that specify what kind of information the symbol represents. Symbols fall into three categories:

- variables
- labels
- numbers

Variables identify data stored at a particular location in memory. All variables have the following three attributes:

- Segment — tells which segment was being assembled when the variable was defined.

- Offset — tells how many bytes there are from the beginning of the segment to the location of this variable.

- Type — tells how many bytes of data are manipulated when this variable is referenced.

A Segment may be a code-segment, a data-segment, a stack-segment or an extra-segment depending on its contents and the register that contains its starting address (see Chapter 3). A segment may start at any address divisible by 16. ASM-86 uses this boundary value as the Segment portion of the variable's definition.

The Offset of a variable may be any number between 0 and 0FFFFH (65535D). A variable must have one of the following Type attributes:

- BYTE
- WORD
- DWORD

BYTE specifies a one-byte variable, WORD a two-byte variable, and DWORD a four-byte variable. The DB, DW, and DD directives respectively define variables as these three types (see Chapter 3). For example, a variable is defined when it appears as the name for a storage directive:

```
VARIABLE DB 0
```

A variable may also be defined as the name in an EQU directive referencing another label, as shown below.

```
VARIABLE EQU  ANOTHER_VARIABLE
```

Labels identify locations in memory that contain instruction statements. They are referenced with jumps or calls. All labels have two attributes:

- Segment
- Offset

Label segment and offset attributes are essentially the same as variable segment and offset attributes. Generally, a label is defined when it precedes an instruction. A colon (:) separates the label from the instruction, as shown in the following example.

```
LABEL:  ADD  AX,BX
```

A label may also appear as the name in an EQU directive referencing another label, as shown in the following example.

```
LABEL  EQU  ANOTHER_LABEL
```

Numbers may also be defined as symbols. A number symbol is treated as if you had explicitly coded the number it represents. For example:

```
Number_five  EQU  5
MOV  AL, Number_five
```

is equivalent to:

```
MOV  AL,5
```

The following section describes operators and their effects on numbers and number symbols.

OPERATORS

ASM-86 operators fall into the following categories:

- arithmetic
- logical
- relational
- segment override
- variable manipulators and creators.

Table 2-5 defines ASM-86 operators. In this table, *a* and *b* represent two elements of the expression. The validity column defines the type of operands the operator can manipulate, with the bar character (|) indicating alternatives.

Table 2-5 ASM-86 Operators

SYNTAX	RESULT	VALIDITY
Logical Operators		
*a*XOR *b*	Bit-by-bit logical EXCLUSIVE OR of *a* and *b*	*a*, *b* = number
a OR *b*	Bit-by-bit logical OR of *a* and *b*	*a*, *b* = number
a AND *b*	Bit-by-bit logical AND of *a* and *b*	*a*, *b* = number
NOT *a*	Logical inverse of *a*: all 0's become 1's, 1's become 0's	*a* = 16-bit number
Relational Operators		
*a*EQ *b*	0FFFFH if *a* = *b*, otherwise 0.	*a*,*b* = unsigned number
*a*LT *b*	0FFFFH if *a* < *b*, otherwise 0.	*a*,*b* = unsigned number
a LE *b*	0FFFFH if *a* <=*b*, otherwise 0.	*a*,*b* = unsigned number
a GT *b*	0FFFFH if *a* > *b*, otherwise 0.	*a*,*b* = unsigned number
a GE *b*	0FFFFH if *a* >= *b*, otherwise 0.	*a*,*b* = unsigned number
a NE *b*	0FFFFH if *a*<>*b*, otherwise 0.	*a*,*b* = unsigned number

Table 2-5 ASM-86 Operators (cont'd)

SYNTAX	RESULT	VALIDITY
Arithmetic Operators		
a + *b*	Arithmetic sum of *a* and *b*	*a* = variable, label or number *b* = number
a - *b*	Arithmetic difference of *a* and *b*	*a* = variable, label or number *b* = number
a * *b*	Unsigned multiplication of *a* and *b*	*a*, *b* = number
a / *b*	Unsigned division of *a* and *b*	*a*, *b*, = number
a MOD *b*	Remainder of *a* / *b*	*a*, *b*, = number
a SHL *b*	Value that results from shifting *a* to left by an amount *b*	*a*, *b*, =, number
a SHR *b*	Value that results from shifting *a* to the right by an amount *b*	*a*, *b*, = number
+ *a*	*a*	*a* = number
- *a*	0 - *a*	*a* = number
Segment Override		
seg reg: addr exp	Overrides assembler's choice of segment register	< seg reg > = CS, DS, SS or ES

Table 2-5 ASM-86 Operators (cont'd)

SYNTAX	RESULT	VALIDITY
Variable Manipulators, Creators		
SEG *a*	Number whose value is the segment value of the variable or label *a*	*a* = label | variable
OFFSET *a*	Number whose value is the offset value of the variable or label *a*	*a* = label | variable
TYPE *a*	Number - If the variable *a* is of type BYTE, WORD or DWORD, the value of the number will be 1,2 or 4, respectively.	*a* = label | variable
LENGTH *a*	Number whose value is the LENGTH attribute of the variable *a* - The length attribute is the number of bytes associated with the variable.	*a* = label | variable
LAST *a*	If LENGTH *a* > 0, then LAST *a* = LENGTH *a* - 1. If LENGTH *a* = 0, then LAST *a* = 0.	*a* = label | variable
a PTR *b*	Virtual variable or label with type of *a* and attributes of *b*	*a* = BYTE | WORD, | DWORD *b* = <addr exp>
.*a*	Variable with an offset attribute of *a*. Segment attribute is current segment.	*a* = number

Table 2-5 ASM-86 Operators (cont'd)

SYNTAX	RESULT	VALIDITY
Arithmetic Operators		
$	Label with offset equal to current value of location counter; segment attribute is current segment.	no argument

Operator Examples

Logical operators perform the Boolean logic operations AND, OR, XOR, and NOT, accepting only numbers as operands. For example:

```
00FC            MASK        EQU     0FCH
0080            SIGNBIT     EQU     80H
0000 B180                   MOV     CL,MASK AND SIGNBIT
0002 B003                   MOV     AL,NOT MASK
```

Relational operators treat all operands as unsigned numbers. The relational operators are EQ (equal), LT (less than), LE (less than or equal), GT (greater than), GE (greater than or equal), and NE (not equal). Each operator compares two operands and returns all ones (0FFFFH) if the specified relation is true and all zeros if it is not. For example:

```
000A            LIMIT1      EQU     10
0019            LIMIT2      EQU     25
                .
                .
                .
0004 B8FFFF                 MOV     AX,LIMIT1 LT LIMIT2
0007 B80000                 MOV     AX,LIMIT1 GT LIMIT2
```

Addition and subtraction operators compute the arithmetic sum and difference of two operands. The first operand may be a variable, label, or number, but the second operand must be a number. When a number is added to a variable or label, the result is a variable or label whose offset is the numeric value of the second operand plus the

offset of the first operand. Subtraction from a variable or label returns a variable or label whose offset is that of first operand decremented by the number specified in the second operand. For example:

```
0002                    COUNT       EQU     2
0005                    DISP1       EQU     5
000A FF                 FLAG        DB      0FFH
                        .
                        .
                        .
000B 2EA00B00                       MOV     AL,FLAG+1
000F 2E8A0E0F00                     MOV     CL,FLAG+DISP1
0014 B303                           MOV     BL,DISP1-COUNT
```

The multiplication and division operators *, /, MOD, SHL, and SHR accept only numbers as operands. The operators, * and /, treat all operators as unsigned numbers. For example:

```
0016 BE5500                         MOV     SI,256/3
0019 B310                           MOV     BL,64/4
0050                    BUFFERSIZE          EQU 80
001B B8A000                         MOV     AX,BUFFERSIZE * 2
```

Unary operators accept both signed and unsigned operators as shown below:

```
001E B123                           MOV     CL,+35
0020 B007                           MOV     AL,2--5
0022 B2F4                           MOV     DL,-12
```

When manipulating variables, the assembler decides which segment register to use. You may override the assembler's choice by specifying a different register with the segment override operator. The syntax for the override operator is as follows:

segment register:address expression

where the segment register is CS, DS, SS, or ES. For example:

```
0024 3688B472D                      MOV     AX,SS:WORDBUFFER[BX]
0028 268B0E5B00                     MOV     CX,ES:ARRAY
```

A variable manipulator creates a number equal to one attribute of its variable operand. SEG extracts the variable's segment value, OFFSET its offset value, TYPE its type value (1, 2, or 4), and LENGTH the number of bytes associated with the variable. LAST compares the variable's LENGTH with 0 and if greater, then decrements LENGTH by one. If LENGTH equals 0, LAST leaves it unchanged. Variable manipulators accept only variables as operators. For example:

```
002D 000000000000   WORDBUFFER   DW      0,0,0
0033 0102030405     BUFFER       DB      1,2,3,4,5
                    .
                    .
                    .
0038 B80500                      MOV     AX,LENGTH BUFFER
0038 B80400                      MOV     AX,LAST BUFFER
003E B80100                      MOV     AX,TYPE BUFFER
0041 B80200                      MOV     AX,TYPE WORDBUFFER
```

The PTR operator creates a virtual variable or label, that is, one valid only during the execution of the instruction. PTR makes no changes to either of its operands. The temporary symbol has the same Type attribute as the left operator, and all other attributes of the right operator as shown below.

```
0044 C60705                      MOV     BYTE PTR [BX], 5
0047 8A07                        MOV     AL,BYTE PTR [BX]
0049 FF04                        INC     WORD PTR [SI]
```

The Period operator (.) creates a variable in the current data segment. The new variable has a segment attribute equal to the current data segment and an offset attribute equal to its operand. Its operand must be a number. For example:

```
0048 A10000                      MOV     AX, .0
004E 268B1E0040                  MOV     BX, ES: .4000H
```

The Dollar sign operator ($) creates a label with an offset attribute equal to the current value of the location counter. The label's segment value is the same as the current code segment. This operator takes no operand. For example:

```
0053 E9FDFF                      JMP     $
0056 EBFE                        JMPS    $
0058 E9FD2F                      JMP     $+3000H
```

Operator Precedence

Expressions combine variables, labels or numbers with operators. ASM-86 allows several kinds of expressions (discussed in the next section). If more than one operator appears in an expression, the operations they perform occur in a specific order of precedence.

In general, ASM-86 evaluates expressions left to right, but operators with higher precedence are evaluated before operators with lower precedence. When two operators have equal precedence, the leftmost is evaluated first. Table 2-6 presents ASM-86 operators in order of increasing precedence.

Parentheses can override normal rules of precedence. The part of an expression enclosed in parentheses is evaluated first. If parentheses are nested, the innermost expressions are evaluated first. However, only five levels of nested parentheses are legal. For example:

```
15/3 + 18/9 = 5 + 2 = 7
15/(3 + 18/9) = 15/(3 + 2) = 15/5 =3
```

Table 2-6 Precedence of Operations in ASM-86

ORDER	OPERATOR TYPE	OPERATORS
1	Logical	XOR, OR
2	Logical	AND
3	Logical	NOT
4	Relational	EQ, LT, LE, GT, GE, NE
5	Addition/subtraction	+, -
6	Multiplication/division	*, /, MOD, SHL, SHR
7	Unary	+, -
8	Segment override	<segment override>:

Table 2-6 Precedence of Operations in ASM-86 (cont'd)

ORDER	OPERATOR TYPE	OPERATORS
9	Variable manipulators, creators	SEG, OFFSET, PTR, TYPE, LENGTH, LAST
10	Parentheses/brackets	( ), [ ]
11	Period and Dollar	., $

EXPRESSIONS

ASM-86 allows address, numeric, and bracketed expressions. An address expression evaluates to a memory address and has three components:

- a segment value
- an offset value
- a type

Both variables and labels are address expressions. An address expression is not a number, but its components are. Numbers may be combined with operators such as PTR to make an address expression.

A numeric expression evaluates to a number. It does not contain any variables or labels, only numbers and operands.

Bracketed expressions specify base- and index-addressing modes. The base registers are BX and BP, and the index registers are DI and SI. A bracketed expression may consist of a base register, an index register, or both a base register and an index register.

Use the + operator between a base register and an index register to specify both base- and index-register addressing. For example:

```
MOV   variable[bx],0
MOV   AX,[BX+DI]
MOV   AX,[SI]
```

STATEMENTS

Just as tokens in the ASM-86 assembly language correspond to words in English, so are statements analogous to sentences. A statement tells ASM-86 what action to perform. Two types of statements are used: instructions and directives. Instructions are translated by the assembler into 8086 machine language instructions. Directives are not translated into machine code but instead direct the assembler to perform certain clerical functions.

Terminate each assembly language statement with a carriage return (CR) and line feed (LF), or with an exclamation point (!), which ASM-86 treats as an end-of-line. Multiple assembly language statements can be written on the same physical line if separated by exclamation points.

The AS instruction set is defined in Chapter 4. The following is the syntax for an instruction statement.

{label:} {prefix} mnemonic {operand(s)} {;comment}

where the fields are defined as follows:

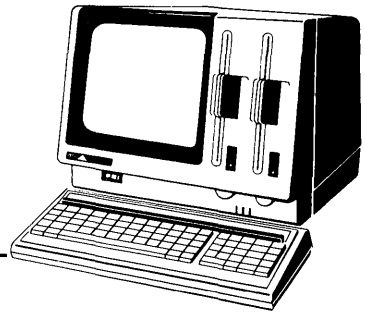
label:	A symbol followed by ":" defines a label at the current value of the location counter in the current segment. This field is optional.
prefix	Certain machine instructions such as LOCK and REP may prefix other instructions. This field is optional.
mnemonic	A symbol defined as a machine instruction, either by the assembler or by an EQU directive. This field is optional unless preceded by a prefix instruction. If it is omitted, no operands may be present, although the other fields may appear. ASM-86 mnemonics are defined in Chapter 4.
operand(s)	An instruction mnemonic may require other symbols to represent operands to the instruction. Instructions may have zero, one or two operands.
comment	Any semicolon (;) appearing outside a character string begins a comment. A comment is ended by a carriage return. An exclamation point in a comment is ignored and is not treated as a delimiter. Comments improve the readability of programs. This field is optional.

ASM-86 directives are described in Chapter 3. The following is the syntax for a directive statement:

{name} directive operand(s) {;comment}

where the fields are defined as follows:

name	Unlike the label field of an instruction, the name field of a directive is never terminated with a colon. Directive names are legal for only DB, DW, DD, RS and EQU. For DB, DW, DD and RS the name is optional; for EQU it is required.
directive	One of the directive keywords defined in Chapter 3.
operand(s)	Analogous to the operands to the instruction mnemonics. Some directives, such as DB, DW, and DD, allow any operand while others have special requirements.
comment	Exactly as defined for instruction statements.



Chapter 3

Assembler Directives

INTRODUCTION

Directive statements cause ASM-86 to perform housekeeping functions such as assigning portions of code to logical segments, requesting conditional assembly, defining data items, and specifying listing file format. General syntax for directive statements appears in Chapter 2.

In the sections that follow, the specific syntax for each directive statement is given under the heading and before the explanation. These syntax lines use special symbols to represent possible arguments and other alternatives. Brackets, { }, enclose optional arguments. Angle brackets, < >, enclose descriptions of user-supplied arguments. Do not include these symbols when coding a directive.

SEGMENT START DIRECTIVES

At runtime, every 8086 memory reference must have a 16-bit segment base value and a 16-bit offset value. These are combined to produce the 20-bit effective address needed by the CPU to physically address the location. The 16-bit segment base value or boundary is contained in one of the segment registers CS, DS, SS, or ES. The offset value gives the offset of the memory reference from the segment boundary. A 16-byte physical segment is the smallest relocatable unit of memory.

ASM-86 predefines four logical segments: the Code Segment, Data Segment, Stack Segment, and Extra Segment, which are respectively addressed by the CS, DS, SS, and ES registers. Future versions of ASM-86 will support additional segments such as multiple data or code segments. All ASM-86 statements must be assigned to one of the four currently supported segments so that they can be referenced by the CPU. A segment directive statement, CSEG, DSEG, SSEG, or ESEG, specifies that the statements following it belong to a specific segment. The statements are then addressed by the corresponding segment register. ASM-86 assigns statements to the specified segment until it encounters another segment directive.

Instruction statements must be assigned to the Code Segment. Directive statements may be assigned to any segment. ASM-86 uses these assignments to change from one segment register to another. For example, when an instruction accesses a memory variable, ASM-86 must know which segment contains the variable so it can generate a segment override prefix byte if necessary.

The CSEG Directive

```
CSEG    <numeric expression>
CSEG
CSEG    $
```

This directive tells the assembler that the following statements belong in the Code Segment. All instruction statements must be assigned to the Code Segment. All directive statements are legal within the Code Segment.

Use the first form when the location of the segment is known at assembly time; the code generated is not relocatable. Use the second form when the segment location is not known at assembly time; the code generated is relocatable. Use the third form to continue the Code Segment after it has been interrupted by a DSEG, SSEG, or ESEG directive. The continuing Code Segment starts with the same attributes, such as location and instruction pointer, as the previous Code Segment.

The DSEG Directive

```
DSEG    <numeric expression>
DSEG
DSEG    $
```

This directive specifies that the following statements belong to the Data Segment. The Data Segment primarily contains the data allocation directives DB, DW, DD and RS, but all other directive statements are also legal. Instruction statements are illegal in the Data Segment.

Use the first form when the location of the segment is known at assembly time; the code generated is not relocatable. Use the second form when the segment location is not known at assembly time; the code generated is relocatable. Use the third form to continue the Data Segment after it has been interrupted by a CSEG, SSEG, or ESEG directive. The continuing Data Segment starts with the same attributes as the previous Data Segment.

The SSEG Directive

```
SSEG    <numeric expression>
SSEG
SSEG    $
```

The SSEG directive indicates the beginning of source lines for the Stack Segment. Use the Stack Segment for all stack operations. All directive statements are legal in the Stack Segment, but instruction statements are illegal.

Use the first form when the location of the segment is known at assembly time; the code generated is not relocatable. Use the second form when the segment location is not known at assembly time; the code generated is relocatable. Use the third form to continue the Stack Segment after it has been interrupted by a CSEG, DSEG, or ESEG directive. The continuing Stack Segment starts with the same attributes as the previous Stack Segment.

The ESEG Directive

```
ESEG    <numeric expression>
ESEG
ESEG    $
```

This directive initiates the Extra Segment. All directive statements are legal in the Extra Segment, but instruction statements are illegal.

Use the first form when the location of the segment is known at assembly time; the code generated is not relocatable. Use the second form when the segment location is not known at assembly time; the code generated is relocatable. Use the third form to continue the Extra Segment after it has been interrupted by a DSEG, SSEG, or CSEG directive. The continuing Extra Segment starts with the same attributes as the previous Extra Segment.

THE ORG DIRECTIVE

```
ORG     <numeric expression>
```

The ORG directive sets the offset of the location counter in the current segment to the value specified in the numeric expression. Define all elements of the expression before the ORG directive to avoid ambiguity in forward references.

In most segments, an ORG directive is unnecessary. If no ORG is included before the first instruction or data byte in a segment, assembly begins at location zero relative to the beginning of the segment. A segment can have any number of ORG directives.

THE IF AND ENDIF DIRECTIVES

```
IF      <numeric expression>
        <source line 1>
        <source line 2>
                .
                .
                .
        <source line n>
        ENDIF
```

The IF and ENDIF directives allow a group of source lines to be conditionally included or excluded from the assembly. Use conditional directives to assemble several different versions of a single source program.

When the assembler finds an IF directive, it evaluates the numeric expression following the IF keyword. If the expression evaluates to a nonzero value, then *<source line 1>* through *<source line n>* are assembled. If the expression evaluates to zero, then the lines are not assembled. All elements in the numeric expression must be defined before they appear in the IF directive. IF directives can be nested to five levels.

THE IFLIST AND NOIFLIST DIRECTIVES

The IFLIST directive tells ASM-86 to list the lines in false IF blocks. This is the default condition. The NOIFLIST directive suppresses listing the lines when an IF directive evaluates to zero.

THE INCLUDE DIRECTIVE

```
INCLUDE <filename>
```

This directive includes another ASM-86 file in the source text. For example, the directive:

```
INCLUDE EQUALS.A86
```

instructs the assembler to insert into the source text the file named EQUALS.A86. Use INCLUDE when the source program resides in several different files. INCLUDE directives may not be nested; a source file called by an INCLUDE directive may not contain another INCLUDE statement. If *<filename>* does not

contain a file type, the file type is assumed to be .A86. If no drive name is specified with *<filename>*, ASM-86 assumes the drive containing the source file.

THE END DIRECTIVE

```
END
```

An END directive marks the end of a source file. Any subsequent lines are ignored by the assembler. END is optional. If not present, ASM-86 processes the source until it finds an end-of-file character (1AH).

THE EQU DIRECTIVE

```
symbol    EQU    <numeric expression>
symbol    EQU    <address expression>
symbol    EQU    <register>
symbol    EQU    <instruction mnemonic>
```

The EQU (equate) directive assigns values and attributes to user-defined symbols. The required symbol name may not be terminated with a colon. The symbol cannot be redefined by a subsequent EQU or another directive. Any elements used in numeric or address expressions must be defined before the EQU directive appears.

The first form assigns a numeric value to the symbol; the second assigns a memory address. The third form assigns a new name to an 8086 register. The fourth form defines a new instruction subset. The following are examples of these four forms:

```
0005        FIVE       EQU    2*2 + 1
0033        NEXT       EQU    BUFFER
0001        COUNTER    EQU    CX
            MOVVV      EQU    MOV
                        .
                        .
                        .
005D 8BC3               MOVVV  AX,BX
```

An error is reported if an invalid numeric quantity appears in an EQU directive. Forward references in EQU directives are flagged as errors.

THE DB DIRECTIVE

```
{symbol} DB <numeric expression> { , <numeric expression> .. }
{symbol} DB <string constant> {<string constant> ... }
```

The DB directive defines initialized storage areas in byte format. Numeric expressions are evaluated to 8-bit values and sequentially placed in the hex output file. String constants are placed in the output file according to the rules defined in Chapter 2. A DB directive is the only ASM-86 statement that accepts a string constant longer than two bytes. There is no translation from lowercase to uppercase within strings. Multiple expressions or constants, separated by commas, may be added to the definition, but may not exceed the physical line length.

Use an optional symbol to reference the defined data area throughout the program. The symbol has four attributes: the Segment and Offset attributes determine the symbol's memory reference, the Type attribute specifies single bytes, and Length tells the number of bytes (allocation units) reserved.

The following statements show DB directives with symbols:

```
005F  43502F4D2073  TEXT  DB  'CP/M system',0
      797374656D00
006B  E1            AA    DB  'a' + 80H
006C  0102030405    X     DB  1,2,3,4,5
                                .
                                .
                                .
0071  B90C00                MOV CX, LENGTH TEXT
```

THE DW DIRECTIVE

{symbol} DW *<numeric expression>* { , *<numeric expression>* .. }
{symbol} DW *<string constant>* { , *<string constant>* ... }

The DW directive initializes two-byte words of storage. String constants longer than two characters are illegal. Otherwise, DW uses the same procedure to initialize storage as DB. The following are examples of DW statements:

```
0074  0000          CNTR    DW  0
0076  63C166C169C1  JMPTAB  DW  SUBR1,SUBR2,SUBR3
007C  010002000300          DW  1,2,3,4,5,6
      040005000600
```

THE DD DIRECTIVE

{symbol} DD *<numeric expression>* { , *<numeric expression>* .. }

The DD directive initializes four bytes of storage. The Offset attribute of the address expression is stored in the two lower bytes, the Segment attribute in the two upper bytes. Otherwise, DD follows the same procedure as DB. The following are examples of DD statements:

```
1234                        CSEG            1234H
                                              .
                                              .
                                              .
0000  6CC134126FC1 LONG_JMPTAB  DD      ROUT1,ROUT2
      3412
0008  72C1341275C1              DD      ROUT3,ROUT4
      3412
```

THE RS DIRECTIVE

{symbol} RS *<numeric expression>*

The RS directive allocates storage in memory but does not initialize it. The numeric expression gives the number of bytes to be reserved. An RS statement does not give a byte attribute to the optional symbol. The following example shows the RS statement:

```
0010                        BUF             RS      80
0060                                        RS      4000H
4060                                        RS      1
```

THE RB DIRECTIVE

{symbol} RB *<numeric expression>*

The RB directive allocates byte storage in memory without any initialization. This directive is identical to the RS directive except that it does give the byte attribute.

THE RW DIRECTIVE

{symbol} RW *<numeric expression>*

The RW directive allocates two-byte word storage in memory but does not initialize it. The numeric expression gives the number of words to be reserved. The following example shows the RW statement:

```
4061            BUFF    RW      128
4161                    RW      4000H
C161                    RW      1
```

THE TITLE DIRECTIVE

TITLE *<string constant>*

ASM-86 prints the string constant defined by a TITLE directive statement at the top of each printout page in the listing file. The title character string should not exceed 30 characters. The following example shows the TITLE statement:

```
TITLE   'CP/M monitor'
```

THE PAGESIZE DIRECTIVE

PAGESIZE *<numeric expression>*

The PAGESIZE directive defines the number of lines to be included on each printout page. The default page size is 66.

THE PAGEWIDTH DIRECTIVE

PAGEWIDTH *<numeric expression>*

The PAGEWIDTH directive defines the number of columns printed across the page when the listing file is output. The default page width is 120 columns unless the listing is routed directly to the terminal; then the default page width is 79 columns.

THE EJECT DIRECTIVE

EJECT

The EJECT directive performs a page eject during printout. The EJECT directive itself is printed on the first line of the next page.

THE SIMFORM DIRECTIVE

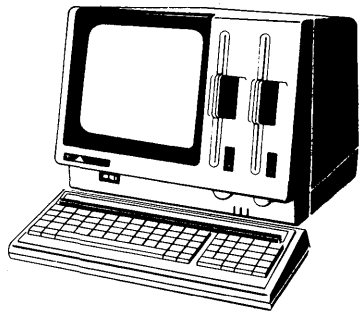
SIMFORM

The SIMFORM directive replaces a form feed (FF) character in the print file with the correct number of line feeds (LF). Use this directive when printing on a printer that is unable to interpret the form feed character.

THE NOLIST AND LIST DIRECTIVE

NOLIST
LIST

The NOLIST directive blocks the printout of lines following the directive. Restart the listing with a LIST directive.



Chapter 4

The ASM-86 Instruction Set

INTRODUCTION

The ASM-86 instruction set includes all 8086 machine instructions. The general syntax for instruction statements is given in Chapter 2. The following sections define the specific syntax and required operand types for each instruction, without reference to labels or comments. The instruction definitions are presented in tables for easy reference. For a more detailed description of each instruction, see Intel's *MCS-86 Assembly Language Reference Manual.* For descriptions of the instruction bit patterns and operations, see Intel's *MCS-86 User's Manual.*

The instruction definition tables present ASM-86 instruction statements as combinations of mnemonics and operands. A mnemonic is a symbolic representation for an instruction, and its operands are the required parameters. Instructions can take zero, one, or two operands. When two operands are specified, the left operand is the instruction's destination operand and the two operands are separated by a comma.

The instruction definition tables organize ASM-86 instructions into functional groups. Within each table, the instructions are listed alphabetically. Table 4-1 shows the symbols used in the instruction definition tables to define operand types.

Table 4-1 Operand Type Symbols

SYMBOL	OPERAND TYPE
numb	any NUMERIC expression
numb8	any NUMERIC expression which evaluates to an 8-bit number
acc	accumulator register, AX or AL
reg	any general purpose register, not segment register

Table 4-1 Operand Type Symbols (cont'd)

SYMBOL	OPERAND TYPE
reg16	a 16-bit general purpose register, not segment register
segreg	any segment register: CS, DS, SS, or ES
mem	any ADDRESS expression, with or without base- and/or index-addressing modes, such as: variable variable+3 variable[bx] variable[SI] variable[BX+SI] [BX] [BP+DI]
simpmem	any ADDRESS expression WITHOUT base- and index- addressing modes, such as: variable variable+4
mem|reg	any expression symbolized by "reg" or "mem"
mem|reg16	any expression symbolized by "mem reg", but must be 16 bits
label	any ADDRESS expression which evaluates to a label
lab8	any "label" which is within ±128 bytes distance from the instruction

The 8086 CPU has nine single-bit flag registers which reflect the state of the CPU. The user cannot access these registers directly, but can test them to determine the effects of an executed instruction on an operand or register. The effects of instructions on flag registers are also described in the instruction definition tables, using the symbols shown in Table 4-2 to represent the nine flag registers.

Table 4-2 Flag Register Symbols

AF	Auxiliary-Carry-Flag
CF	Carry-Flag
DF	Direction-Flag
IF	Interrupt-Enable-Flag
OF	Overflow-Flag
PF	Parity-Flag
SF	Sign-Flag
TF	Trap-Flag
ZF	Zero-Flag

DATA TRANSFER INSTRUCTIONS

There are four classes of data transfer operations: general purpose, accumulator specific, address-object and flag. Only SAHF and POPF affect flag settings. Note in Table 4-3 that if acc=AL, a byte is transferred, but if acc=AX, a word is transferred.

Table 4-3 Data Transfer Instructions

	SYNTAX	RESULT
IN	*acc,numb8|numb16*	Transfer data from input port given by *numb8* or *numb16* (0-255) to accumulator.
IN	*acc,*DX	Transfer data from input port given by DX register (0-0FFFFH) to accumulator.
LAHF		Transfer flags to the AH register.

Table 4-3 Data Transfer Instructions (cont'd)

SYNTAX		RESULT
LDS	*reg16, mem*	Transfer the segment part of the memory address (DWORD variable) to the DS segment register, transfer the offset part to a general purpose 16-bit register.
LEA	*reg16,mem*	Transfer the offset of the memory address to a (16-bit) register.
LES	*reg16,mem*	Transfer the segment part of the memory address to the ES segment register, transfer the offset part to a 16-bit general purpose register.
MOV	*reg,mem|reg*	Move memory or register to register.
MOV	*mem|reg,reg*	Move register to memory or register.
MOV	*mem|reg,numb*	Move immediate data to memory or register.
MOV	*segreg,mem|reg16*	Move memory or register to segment register.
MOV	*mem|reg16,segreg*	Move segment register to memory or register.
OUT	*numb8|numb16,acc*	Transfer data from accumulator to output port (0-255) given by *numb8* or *numb16.*
OUT	DX,*acc*	Transfer data from accumulator to output port (0-FFFFH) given by DX register.
POP	*mem|reg16*	Move top stack element to memory or register.
POP	*segreg*	Move top stack element to segment register; note that CS segment register not allowed.
POPF		Transfer top stack element to flags.

Table 4-3 Data Transfer Instructions (cont'd)

SYNTAX		RESULT
PUSH	*mem|reg16*	Move memory or register to top stack element.
PUSH	*segreg*	Move segment register to top stack element.
PUSHF		Transfer flags to top stack element.
SAHF		Transfer the AH register to flags.
XCHG	*reg,mem|reg*	Exchange register and memory or register.
XCHG	*mem|reg,reg*	Exchange memory or register and register.
XLAT	*mem|reg*	Perform table lookup translation. The table is given by "mem reg", which is always BX. Replaces AL with AL offset from BX.

ARITHMETIC, LOGICAL, AND SHIFT INSTRUCTIONS

The 8086 CPU performs the four basic mathematical operations in several different ways. It supports both 8-bit and 16-bit operations as well as signed and unsigned arithmetic.

Six of the nine flag bits are set or cleared by most arithmetic operations to reflect the result of the operation. Table 4-4 summarizes the effects of arithmetic instructions on flag bits. Table 4-5 defines arithmetic instructions. Table 4-6 defines logical and shift instructions.

Table 4-4 Effects of Arithmetic Instructions on Flags

FLAG	EFFECT
CF	is set if the operation resulted in a carry out of (from addition) or a borrow into (from subtraction) the high-order bit of the result; otherwise CF is cleared.

Table 4-4 Effects of Arithmetic Instructions on Flags (cont'd)

FLAG	EFFECT
AF	is set if the operation resulted in a carry out of (from addition) or a borrow into (from subtraction) the low-order four bits of the result; otherwise AF is cleared.
ZF	is set if the result of the operation is zero; otherwise ZF is cleared.
SF	is set if the result is negative.
PF	is set if the modulo 2 sum of the low-order eight bits of the result of the operation is 0 (even parity); otherwise PF is cleared (odd parity).
OF	is set if the operation resulted in an overflow; the size of the result exceeded the capacity of its destination.

Table 4-5 Arithmetic Instructions

SYNTAX		RESULT
AAA		Adjust unpacked BCD (ASCII) for addition - adjusts AL.
AAD		Adjust unpacked BCD(ASCII) for division - adjusts AL.
AAM		Adjust unpacked BCD (ASCII) for multiplication - adjusts AX.
AAS		Adjust unpacked BCD (ASCII) for subtraction -adjusts AL.
ADC	*reg,mem|reg*	Add (with carry) memory or register to register.
ADC	*mem|reg,reg*	Add (with carry) register to memory or register.

Table 4-5 Arithmetic Instructions (cont'd)

SYNTAX		RESULT
ADC	*mem|reg,numb*	Add (with carry) immediate data to memory or register.
ADD	*reg,mem|reg*	Add memory or register to register.
ADD	*mem|reg,reg*	Add register to memory or register.
ADD	*mem|reg,numb*	Add immediate data to memory or register.
CBW		Convert byte in AL to word in AH by sign extension.
CWD		Convert word in AX to double word in DX/AX by sign extension.
CMP	*reg,mem|reg*	Compare register with memory or register.
CMP	*mem|reg,reg*	Compare memory or register with register.
CMP	*mem|reg,numb*	Compare data constant with memory or register.
DAA		Perform decimal adjustment for addition - adjusts AL.
DAS		Perform decimal adjustment for subtraction - adjusts AL.
DEC	*mem|reg*	Subtract 1 from memory or register.
INC	*mem|reg*	Add 1 to memory or register.
DIV	*mem|reg*	Divide (unsigned) accumulator (AX or AL) by memory or register. If byte results, AL = quotient, AH = remainder. If word results, AX = quotient, DX = remainder.
IDIV	*mem|reg*	Divide (signed) accumulator (AX or AL) by memory or register - quotient and remainder stored as in DIV.

Table 4-5 Arithmetic Instructions (cont'd)

SYNTAX		RESULT
IMUL	*mem|reg*	Multiply (signed) memory or register by accumulator (AX or AL) - if byte, results in AH, AL. If word, results in DX, AX,
MUL	*mem|reg*	Multiply (unsigned) memory or register by accumulator (AX or AL) - results stored as in IMUL.
NEG	*mem|reg*	Two's complement memory or register.
SBB	*reg,mem|reg*	Subtract (with borrow) memory or register from register.
SBB	*mem|reg,reg*	Subtract (with borrow) register from memory or register.
SBB	*mem|reg,numb*	Subtract (with borrow) immediate data from memory or register.
SUB	*reg,mem|reg*	Subtract memory or register from register.
SUB	*mem|reg,reg*	Subtract register from memory or register.
SUB	*mem|reg,numb*	Subtract data constant from memory or register.

Table 4-6 Logic and Shift Instructions

SYNTAX		RESULT
AND	*reg,mem|reg*	Perform bitwise logical "and" of a register and memory register.
AND	*mem|reg,reg*	Perform bitwise logical "and" of memory register and register.
AND	*mem|reg,numb*	Perform bitwise logical "and" of memory register and data constant.

Table 4-6 Logic and Shift Instructions (cont'd)

SYNTAX		RESULT
NOT	*mem|reg*	Form ones complement of memory or register.
OR	*reg,mem|reg*	Perform bitwise logical "or" of a register and memory register.
OR	*mem|reg,reg*	Perform bitwise logical "or" of memory register and register.
OR	*mem|reg,numb*	Perform bitwise logical "or" of memory register and data constant.
RCL	*mem|reg*,1	Rotate memory or register 1 bit left through carry flag.
RCL	*mem|reg*,CL	Rotate memory or register left through carry flag, number of bits given by CL register.
RCR	*mem|reg*,1	Rotate memory or register 1 bit right through carry flag.
RCR	*mem|reg*,CL	Rotate memory or register right through carry flag, number of bits given by CL register.
ROL	*mem|reg*,1	Rotate memory or register 1 bit left.
ROL	*mem|reg*,CL	Rotate memory or register left, number of bits given by CL register.
ROR	*mem|reg*,1	Rotate memory or register 1 bit right.
ROR	*mem|reg*,CL	Rotate memory or register right, number of bits given by CL register.
SAL	*mem|reg*,1	Shift memory or register 1 bit left, shift in low-order zero bits.

Table 4-6 Logic and Shift Instructions (cont'd)

SYNTAX		RESULT
SAL	*mem|reg*,CL	Shift memory or register left, number of bits given by CL register, shift in low-order zero bits.
SAR	*mem|reg*,1	Shift memory or register 1 bit right, shift in high-order bits equal to the original high-order bit.
SAR	*mem|reg*,CL	Shift memory or register right, number of bits given by CL register, shift in high-order bits equal to the original high-order bit.
SHL	*mem|reg*1	Shift memory or register 1 bit left, shift in low-order zero bits. Note that SHL is a different mnemonic for SAL.
SHL	*mem|reg*,CL	Shift memory or register left, number of bits given by CL register, shift in low-order zero bits. Note that SHL is a different mnemonic for SAL.
SHR	*mem|reg*,1	Shift memory or register 1 bit right, shift in high-order zero bits.
SHR	*mem|reg*,CL	Shift memory or register right, number of bits given by CL register, shift in high-order zero bits.
TEST	*reg,mem|reg*	Perform bitwise logical "and" of a register and memory or register. Set condition flags but do not change destination.
TEST	*mem|reg,reg*	Perform bitwise logical "and" of memory register and register. Set condition flags but do not change destination.
TEST	*mem|reg,numb*	Perform bitwise logical "and" test of memory register and data constant. Set condition flags but do not change destination.

Table 4-6 Logic and Shift Instructions (cont'd)

SYNTAX		RESULT
XOR	*reg,mem|reg*	Perform bitwise logical "exclusive or" of a register and memory or register.
XOR	*mem|reg,reg*	Perform bitwise logical "exclusive or" of memory register and register.
XOR	*mem|reg,numb*	Perform bitwise logical "exclusive or" of memory register and data constant.

STRING INSTRUCTIONS

String instructions take one or two operands. The operands specify only the operand type, determining whether the operation is on bytes or words. If there are two operands, the source operand is addressed by the SI register and the destination operand is addressed by the DI register. The DI and SI registers are always used for addressing. Note that for string operations, destination operands addressed by DI must always reside in the Extra Segment (ES).

Table 4-7 String Instructions

SYNTAX		RESULT
CMPS	*mem|reg,mem|reg*	Compare byte or word of string.
CMPSB	*mem|reg,mem|reg*	Compare string in byte form.
CMPSW	*mem|reg,mem|reg*	Compare string in word form.
LODS	*mem|reg*	Transfer a byte or word from the source operand to the accumulator.
LODSB	*mem|reg*	Transfer in byte form.
LODSW	*mem|reg*	Transfer in word form.
MOVS	*mem|reg,mem|reg*	Move 1 byte (or word) from source to destination.
MOVSB	*mem|reg,mem|reg*	Move 1 byte.
MOVSW	*mem|reg,mem|reg*	Move 1 word.
SCAS	*mem|reg*	Subtract destination operand from accumulator (AX or AL), affect flags, but do not return result.
SCASB	*mem|reg*	Subtract destination operand from accumulator in byte form.
SCASW	*mem|reg*	Subtract destination operand from accumulator in word form.
STOS	*mem|reg*	Transfer a byte or word from accumulator to the destination operand.
STOSB	*mem|reg*	Transfer from accumulator to the destination operand in byte form.
STOSW	*mem|reg*	Transfer from the accumulator to the destination operand in word form.

Table 4-8 defines prefixes for string instructions. A prefix repeats the string instruction the number of times contained in the CX register, which is decremented by 1 for each iteration. Prefix mnemonics precede the string instruction mnemonic in the statement line as shown in Chapter 2.

Table 4-8 Prefix Instructions

SYNTAX	RESULT
REP	Repeat until CX register is zero.
REPZ	Repeat until CX register is zero and zero flag (ZF) is not zero.
REPE	Same as "REPZ"
REPNZ	Repeat until CX register is zero and zero flag (ZF) is zero.
REPNE	Same as "REPNZ"

CONTROL TRANSFER INSTRUCTIONS

All of the following classes of control transfer instructions cause program execution to continue at some new location in memory, possibly in a new code segment. The transfer may be absolute or depend on a certain condition.

- calls, jumps, and returns
- conditional jumps
- iterational control
- interrupts

Table 4-9 defines control transfer instructions. In the definitions of conditional jumps, *above* and *below* refer to the relationship between unsigned values, and *greater than* and *less than* refer to the relationship between signed values.

Table 4-9 Control Transfer Instructions

SYNTAX	RESULT
CALL *label*	Push the offset address of the next instruction on the stack, jump to the target label.

Table 4-9 Control Transfer Instructions (cont'd)

SYNTAX		RESULT
CALL	*mem|reg16*	Push the offset address of the next instruction on the stack, jump to location indicated by contents of specified memory or register.
CALLF	*label*	Push CS segment register on the stack, push the offset address of the next instruction on the stack (after CS), jump to the target label.
CALLF	*mem*	Push CS register on the stack, push the offset address of the next instruction on the stack, jump to location indicated by contents of specified double word in memory.
INT	*numb8*	Push the flag registers (as in PUSHF), clear TF and IF flags, transfer control with an indirect call through any one of the 256 interrupt-vector elements. This instruction uses three levels of stack.
INTO		If OF (the overflow flag) is set, push the flag registers (as in PUSHF), clear TF and IF flags, transfer control with an indirect call through interrupt-vector element 4 (location 10H). If the OF flag is cleared, no operation takes place.
IRET		Transfer control to the return address saved by a previous interrupt operation, restore saved flag registers, CS, and IP. This instruction pops three levels of stack.
JA	*lab8*	Jump if "not below or equal" or "above" ((CF or ZF) = 0).
JAE	*lab8*	Jump if "not below" or "above or equal" (CF = 0).
JB	*lab8*	Jump if "below" or "not above or equal" (CF = 1).

Table 4-9 Control Transfer Instructions (cont'd)

SYNTAX		RESULT
JBE	*lab8*	Jump if "below or equal" or "not above" ((CF or ZF) = 1).
JC	*lab8*	Same as "JB"
JCXZ	*lab8*	Jump to target label if CX register is zero.
JE	*lab8*	Jump if "equal" or "zero" (ZF = 1).
JG	*lab8*	Jump if "not less or equal" or "greater" (((SF xor OF) or ZF) =0).
JGE	*lab8*	Jump if "not less" or "greater or equal" ((SF xor OF) = 0).
JL	*lab8*	Jump if "less" or "not greater or equal" ((SF xor OF) = 1).
JLE	*lab8*	Jump if "less or equal" or "not greater" (((SF xor OF) or ZF) = 1).
JMP	*label*	Jump to the target label.
JMP	*mem|reg16*	Jump to location indicated by contents of specified memory or register.
JMPF	*label*	Jump to the target label possibly in another code segment.
JMPS	*lab8*	Jump to the target label within ±128 bytes from instruction.
JNA	*lab8*	Same as "JBE"
JNAE	*lab8*	Same as "JB"

Table 4-9 Control Transfer Instructions (cont'd)

SYNTAX		RESULT
JNB	*lab8*	Same as "JAE"
JNBE	*lab8*	Same as "JA"
JNC	*lab8*	Same as "JNB"
JNE	*lab8*	Jump if "not equal" or "not zero" (ZF = 0).
JNG	*lab8*	Same as "JLE"
JNGE	*lab8*	Same as "JL"
JNL	*lab8*	Same as "JGE"
JNLE	*lab8*	Same as "JG"
JNO	*lab8*	Jump if "not overflow" (OF = 0).
JNP	*lab8*	Jump if "not parity" or "parity odd".
JNS	*lab8*	Jump if "not sign".
JNZ	*lab8*	Same as "JNE"
JO	*lab8*	Jump if "overflow" (OF = 1).
JP	*lab8*	Jump if "parity" or "parity even" (PF = 1).
JPE	*lab8*	Same as "JP"
JPO	*lab8*	Same as "JNP"
JS	*lab8*	Jump if "sign" (SF = 1).
JZ	*lab8*	Same as "JE"

Table 4-9 Control Transfer Instructions (cont'd)

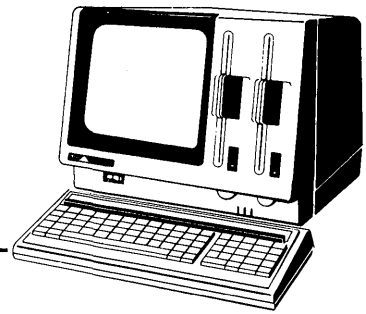
SYNTAX		RESULT
LOOP	*lab8*	Decrement CX register by one, jump to target label if CX is not zero.
LOOPE	*lab8*	Decrement CX register by one, jump to target label if CX is not zero and the ZF flag is set. "Loop while zero" or "loop while equal".
LOOPNE	*lab8*	Decrement CX register by one, jump to target label if CX is not zero and ZF flag is cleared. "Loop while not zero" or "loop while not equal".
LOOPNZ	*lab8*	Same as "LOOPNE"
LOOPZ	*lab8*	Same as "LOOPE"
RET		Return to the return address pushed by a previous CALL instruction, increment stack pointer by 2.
RET	*numb*	Return to the address pushed by a previous CALL, increment stack pointer by 2+*numb*.
RETF		Return to the address pushed by a previous CALLF instruction, increment stack pointer by 4.
RETF	*numb*	Return to the address pushed by a previous CALLF instruction, increment stack pointer by 4+*numb*.

PROCESSOR CONTROL INSTRUCTIONS

Processor control instructions manipulate the flag registers. Moreover, some of these instructions can synchronize the 8086 CPU with external hardware.

Table 4-10 Processor Control Instructions

SYNTAX	RESULT
CLC	Clear CF flag.
CLD	Clear DF flag, causing string instructions to auto-increment the operand points.
CLI	Clear IF flag, disabling maskable external interrupts.
CMC	Complement CF flag.
ESC *numb8,mem|reg*	Do no operation other than compute the effective address and place it on the address bus (ESC is used by the 8087 numeric co-processor), *numb8* must be in the range 0-63.
LOCK	PREFIX instruction, cause the 8086 processor to assert the *buslock* signal for the duration of the operation caused by the following instruction. The LOCK prefix instruction may precede any other instruction. Buslock prevents co-processors from gaining the bus; this is useful for shared-resource semaphores.
HLT	Cause 8086 processor to enter halt state until an interrupt is recognized.
STC	Set CF flag.
STD	Set DF flag, causing string instructions to auto-decrement the operand pointers.
STI	Set IF flag, enabling maskable external interrupts.
WAIT	Cause the 8086 processor to enter a wait state if the signal on its "TEST" pin is not asserted.



Chapter 5

Codemacro Facilities

INTRODUCTION TO CODEMACROS

ASM-86 does not support traditional assembly language macros, but it does allow you to define your own instructions by using the CodeMacro Directive. Like traditional macros, codemacros are assembled wherever they appear in assembly language code, but there the similarity ends. Traditional macros contain assembly language instructions; a codemacro contains only codemacro directives. Macros are usually defined in the user's symbol table; ASM-86 codemacros are defined in the assembler's symbol table. A macro simplifies using the same block of instructions over and over again throughout a program; a codemacro sends a bit stream to the output file and in effect adds a new instruction to the assembler.

Because ASM-86 treats a codemacro as an instruction, you can invoke codemacros by using them as instructions in your program. The example below shows how MAC, an instruction defined by a codemacro, can be invoked.

```
.
.
.
XCHG    BX,WO       RD3
MAC     PAR1,PAR2
MUL     AX,WORD4
.
.
.
```

Note that MAC accepts two operands. When MAC was defined, these two operands were also classified as to type, size, and so on by defining MAC's formal parameters. The names of the formal parameters, however, are not fixed. They merely indicate where and how the actual operands are to be used. The actual names or values of operands are supplied when the codemacro is invoked.

The definition of a codemacro starts with a line specifying its name and its formal parameters, if any.

CodeMacro *<name>* {*<formal parameter list>*}

The optional *<formal parameter list>* is defined as follows:

<formal name>:*<specifier letter>*{*<modifier letter>*{*<range>*}

As stated above, the formal name is not fixed, but merely acts as a place holder. If a formal parameter list is present, the specifier letter is required and the modifier letter is optional. The following are possible specifier letters:

A, C, D, E, M, R, S, X

The following are possible modifier letters:

b, d, w, sb

The assembler ignores case except within strings, but for clarity, this section shows specifiers in uppercase and modifiers in lowercase. The following sections describe specifiers, modifiers, and the optional range in detail.

The body of the codemacro describes the bit pattern and formal parameters. Only the following directives are legal within codemacros:

SEGFIX
NOSEGFIX
MODRM
RELB
RELW
DB
DW
DD
DBIT

These directives are unique to codemacros, and those which appear to duplicate ASM-86 directives (DB, DW, and DD) have different meanings in codemacro context. These directives are discussed in detail in later sections. The definition of each codemacro ends with the line:

EndM

CodeMacro, EndM, and the codemacro directives are all reserved words. Codemacro definition syntax is defined in Backus-Naur-like form in Appendix H.

The following examples are typical codemacro definitions.

```
CodeMacro AAA
        DB 37H
EndM

CodeMacro DIV divisor:Eb
        SEGFIX  divisor
        DB      6FH
        MODRM   divisor
EndM

CodeMacro ESC opcode:Db(0,63),src:Eb
        SEGFIX  src
        DBIT 5  (1BH),3(opcode(3))
        MODRM   opcode,src
EndM
```

SPECIFIERS

Every formal parameter must have a specifier letter that indicates what type of operand is needed to match the formal parameter. Table 5-1 defines the eight possible specifier letters.

Table 5-1 Codemacro Operand Specifiers

LETTER	OPERAND TYPE
A	Accumulator register, AX or AL.
C	Code, a label expression only.
D	Data, a number to be used as an immediate value.
E	Effective address, either an M (memory address) or an R (register).

Table 5-1 Codemacro Operand Specifiers (cont'd)

LETTER	OPERAND TYPE
M	Memory address. This can be either a variable or a bracketed register expression.
R	A general register only.
S	Segment register only.
X	A direct memory reference.

MODIFIERS

The optional modifier letter is a further requirement on the operand. The meaning of the modifier letter depends on the type of the operand. For variables, the following modifiers are used depending on the operand: "*b*" for byte, "*w*" for word, "*d*"for double-word and "*sb*" for signed byte. For numbers, the modifier depends on the size of the number: "*b*" for -256 to 255 and "*w*" for other numbers. Table 5-2 summarizes codemacro modifiers.

Table 5-2 Codemacro Operand Modifiers

VARIABLES		NUMBERS	
Modifier	**Type**	**Modifier**	**Size**
b	byte	*b*	-256 to 255
w	word	*w*	Any other number
d	dword		
sb	signed byte		

RANGE SPECIFIERS

The optional range is specified within parentheses by either one expression or two expressions separated by a comma. The following are valid formats:

(numberb)
(register)
(numberb,numberb)
(numberb,register)
(register,numberb)
(register,register)

where numberb is an 8-bit number, not an address. The following example specifies that the input port must be identified by the DX register:

```
CodeMacro IN dst:Aw,port:Rw(DX)
```

The next example specifies that the CL register is to contain the "count" of rotation:

```
CodeMacro ROR dst:Ew,count:Rb(CL)
```

The next example specifies that the "opcode" is to be immediate data, and may range from 0 to 63 inclusive:

```
CodeMacro ESC opcode:Db(0,63),adds:Eb
```

CODEMACRO DIRECTIVES

Codemacro directives define the bit pattern as well as make further requirements on how the operand is to be treated. Directives are reserved words, and those that appear to duplicate assembly language instructions have different meanings within a codemacro definition. Only the nine directives defined here are legal within codemacro definitions.

SEGFIX

If SEGFIX is present, it instructs the assembler to determine whether a segment-override prefix byte is needed to access a given memory location. If so, the segment-override prefix is output as the first byte of the instruction. If not, no action is taken. SEGFIX takes the form:

SEGFIX *<formal name>*

where *<formal name>* is the name of a formal parameter which represents the memory address. Because it represents a memory address, the formal parameter must have one of the specifiers E, M, or X.

NOSEGFIX

Use NOSEGFIX for operands in instructions that must use the ES register for that operand. This applies only to the destination operand of these instructions: CMPS, MOVS, SCAS, STOS. The following is the form of NOSEGFIX:

NOSEGFIX *segreg, <form name>*

where *segreg* is one of the segment registers ES, CS, SS, or DS and *<form name>* is the name of the memory-address formal parameter, which must have a specifier E, M, or X. No code is generated from this directive, but an error check is performed. The following is an example of the use of NOSEGFIX:

```
CodeMacro MOVS si_ptr:Ew,di_ptr:Ew
        NOSEGFIX   ES,di_ptr
        SEGFIX     si_ptr
        DB         0A5H
EndM
```

MODRM

This directive instructs the assembler to generate the ModRM byte, which follows the opcode byte in many of the 8086's instructions. The ModRM byte contains either the indexing type or the register number to be used in the instruction. It also specifies which register is to be used, or gives more information to specify an instruction.

The ModRM byte carries the information in three fields. The mod field occupies the two most significant bits of the byte, and combines with the register memory field to form 32 possible values: 8 registers and 24 indexing modes.

The reg field occupies the three next bits following the mod field. It specifies either a register number or three more bits of opcode information. The meaning of the reg field is determined by the opcode byte.

The register memory field occupies the last three bits of the byte. It specifies a register as the location of an operand, or forms a part of the address-mode in combination with the mod field described above.

For further information of the 8086's instructions and their bit patterns, see Intel's *8086 Assembly Language Programing Manual* and the *Intel 8086 Family User's Manual.*

The following forms of MODRM are used:

```
MODRM   <form name>,<form name>
MODRM   NUMBER7,<form name>
```

where NUMBER7 is a value 0 to 7 inclusive and *<form name>* is the name of a formal parameter.

The following examples show the use of MODRM:

```
CodeMacro RCR dst:Ew,count:Rb(CL)
        SEGFIX  dst
        DB      0D3H
        MODRM   3,dst
EndM

CodeMacro OR dst:Rw,src:Ew
        SEGFIX  src
        DB      0BH
        MODRM   dst,src
EndM
```

RELB and RELW

These directives, used in IP-relative branch instructions, instruct the assembler to generate displacement between the end of the instruction and the label that is supplied as an operand. RELB generates one byte and RELW two bytes of displacement. The directives take the following forms:

```
RELB <form name>
RELW <form name>
```

where *<form name>* is the name of a formal parameter with a "C" (code) specifier. For example:

```
CodeMacro LOOP place:Cb
        DB      0E2H
        RELB    place
EndM
```

DB, DW and DD

These directives differ from those which occur outside of codemacros. Use the following forms of these directives:

DB <*form name*>| NUMBERB
DW <*form name*>| NUMBERW
DD <*form name*>

where NUMBERB is a single-byte number, NUMBERW is a two-byte number, and *form name* is a name of a formal parameter. For example:

```
CodeMacro XOR dst:Ew,src:Db
        SEGFIX  dst
        DB      81H
        MODRM   6,dst
        DW      src
EndM
```

DBIT

This directive manipulates bits in combinations of a byte or less. The directive takes the following form:

DBIT <*field description*>{,<*field description*>}

where a *field description* has two forms:

<*number*><*combination*>
<*number*>(<*form name*>(<*rshift*>))

where *number* ranges from 1 to 16 and specifies the number of bits to be set. The *combination* parameter specifies the desired bit combination. The total of all the *numbers* listed in the field descriptions must not exceed 16. The second form shown above contains *form name*, a formal parameter name that instructs the assembler to put a certain number in the specified position. This number normally refers to the register specified in the first line of the codemacro. The numbers used in this special case for each register follow.

```
AL:   0
CL:   1
DL:   2
BL:   3
AH:   4
CH:   5
DH:   6
BH:   7
AX:   0
CX:   1
DX:   2
BX:   3
SP:   4
BP:   5
SI:   6
DI:   7
ES:   0
CS:   1
SS:   2
DS:   3
```

The parameter *rshift*, which is contained in the innermost parentheses, specifies a number of right shifts. For example, "0" specifies no shift, "1" shifts right one bit, "2" shifts right two bits, and so on. The definition below uses this form.

```
CodeMacro DEC dst:Rw
        DBIT 5(9H),3(dst(0))
EndM
```

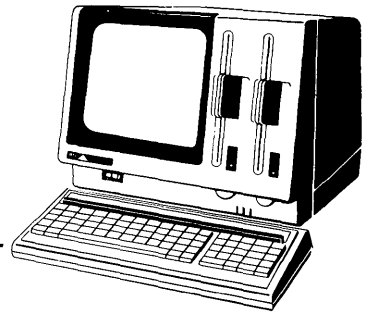
The first five bits of the byte have the value 9H. If the remaining bits are zero, the hex value of the byte will be 48H. If the following instruction is assembled:

```
DEC     DX
```

and DX has a value of 2H, then 48H + 2H = 4AH, which is the final value of the byte for execution. If this sequence had been present in the definition:

```
DBIT 5(9H),3(dst(1))
```

then the register number would have been shifted right once and the result would had been 48H + 1H = 49H, which is erroneous.



Chapter 6

DDT-86

DDT-86 OPERATION

The DDT-86 program allows the user to test and debug programs interactively in a CP/M-86 environment. To use DDT-86, you should be familiar with the 8086 processor, ASM-86, and the CP/M-86 operating system as described in the *CP/M-86 System Reference Guide* for the APC.

Invoking DDT86

Invoke DDT-86 by entering one of the following commands.

```
DDT86
DDT86 filespec
```

The first command loads and executes DDT-86. After displaying its sign-on message and prompt character (-), DDT-86 is ready to accept operator commands. The second command is similar to the first, except that after DDT-86 is loaded, the file specified by *filespec* is also loaded. If the filetype is omitted from the filespec, ".CMD" is assumed. Note that DDT-86 cannot load a file with a filetype of .H86.

The second form of the invoking command is equivalent to the following sequence.

```
A>DDT86
DDT86 x.x
-Efilename
```

At this point, the loaded program is ready for execution.

DDT-86 Command Conventions

When DDT-86 is ready to accept a command, it displays a hyphen (-) as the prompt. In response, you can enter a command line or press CTRL-C to end the debugging session. A command line can have up to 64 characters, and is terminated when you press RETURN. When entering the command, use standard CP/M-86 line-editing functions (CTRL-X, CTRL-H, CTRL-R, and so forth) to correct typing errors. DDT-86 does not process the command line until RETURN is pressed.

The first character of each command line determines the command action. Table 6-1 summarizes the DDT-86 commands, which are defined individually later in the chapter.

Table 6-1 DDT-86 Command Summary

COMMAND	ACTION
A	Enter assembly language statements.
D	Display memory in hexadecimal and ASCII.
E	Load program for execution.
F	Fill memory block with a constant.
G	Begin execution with optional breakpoints.
H	Perform hexadecimal arithmetic.
I	Set up file control block and command tail.
L	List memory using 8086 mnemonics.
M	Move memory block.
R	Read disk file into memory.
S	Set memory to new values.
T	Trace program execution.
U	Perform untraced program monitoring.
V	Show memory layout of disk file read.
W	Write contents of memory block to disk.
X	Examine and modify CPU state.

The command character can be followed by one or more arguments, consisting of hexadecimal values, file names, or other information, depending on the command. Arguments are separated from each other by commas or spaces. No spaces are allowed between the command character and the first argument.

Specifying a 20-Bit Address

Most DDT-86 commands require one or more addresses as operands. Because the 8086 can address up to one megabyte of memory, addresses must be 20-bit values. Enter a 20-bit address as follows:

ssss:oooo

where *ssss* represents an optional 16-bit segment number and *oooo* is a 16-bit offset. DDT-86 combines these values to produce a 20-bit effective address as follows:

```
  ssss0
+ oooo
  ----
  eeeee
```

The optional value *ssss* may be a 16-bit hexadecimal value or the name of a segment register. If you specify a segment register name, the value of *ssss* represents the contents of that register in the user's CPU state, as indicated by the X command. If omitted, a default value appropriate to the command being executed is used, as described later in this chapter.

Terminating DDT-86

Terminate DDT-86 by pressing CTRL-C in response to the hyphen prompt. This returns control to the CCP. Note that CP/M-86 does not have the SAVE facility found in CP/M for 8-bit machines. Therefore, when using DDT-86 to patch a file, write the file to disk with the W command before exiting DDT-86.

DDT-86 OPERATION WITH INTERRUPTS

DDT-86 operates with interrupts disabled while a single instruction is being traced. It preserves the interrupt state of the program being executed under DDT-86. When DDT-86 has control of the CPU (either when it is initially invoked or when it regains control from the program being tested), the condition of the interrupt flag is the same as it was when DDT-86 was invoked. While the program being tested has control of the CPU, the user's CPU state, which can be displayed with the X command, determines the state of the interrupt flag.

DDT-86 COMMANDS

This section defines DDT-86 commands and their arguments. DDT-86 commands give you control of program execution and allow you to display and modify system memory and the CPU state.

The A (Assemble) Command

The A command assembles 8086 mnemonics directly into memory. The following is the form of the A command:

A*s*

where *s* is the 20-bit address where assembly is to start. DDT-86 responds to the A command by displaying the address of the memory location where assembly is to begin. At this point, enter assembly language statements as described in the Assembly Language Syntax section later in this chapter. When you enter a statement, DDT-86 converts it to binary, places the value(s) in memory, and displays the address of the next available memory location. This process continues until you enter a blank line or a line containing only a period.

DDT-86 responds to invalid statements by displaying a question mark (?) and redisplaying the current assembly address.

The D (Display) Command

The D command displays the contents of memory as 8-bit or 16-bit hexadecimal values in ASCII. The following are the forms of the D command:

D
D*s*
D*s,f*
DW
DW*s*
DW*s,f*

where *s* is the 20-bit address where the display is to start, and *f* is the 16-bit offset within the segment specified by *s* where the display is to finish.

Memory is displayed on one or more display lines. Each display line shows the values of up to 16 memory locations. For the first three forms, the display line appears as follows:

ssss:oooo bb bb . . . bb cc . . .c

where *ssss* is the segment being displayed and *oooo* is the offset within segment *ssss*. The *bb*'s represent the contents of the memory locations in hexadecimal, and the *c*'s represent the contents of memory in ASCII. Any nongraphic ASCII characters are represented by periods.

In response to the first form shown above, DDT-86 displays memory from the current display address for 12 display lines. The response to the second form is similar to the first, except that the display address is first set to the 20-bit address *s*. The third form displays the memory block between locations *s* and *f*. The next three forms are analogous to the first three, except that the contents of memory are displayed as 16-bit values, rather than 8-bit values, as shown below.

ssss:oooo wwww wwww . . . wwww cccc . . .cc

During a long display, you can abort the D command by typing any character at the console.

The E (Load for Execution) Command

The E command loads a file into memory so that a subsequent G, T or U command can begin program execution. The E command takes the form:

E*filename*

where *filename* is the name of the file to be loaded. If no filetype is specified, ".CMD" is assumed. The contents of the user segment registers and IP register are altered according to the information in the header of the file loaded.

An E command releases all blocks of memory allocated by any previous E or R commands or by programs executed under DDT-86. Therefore, only one file at a time may be loaded for execution.

When the load is complete, DDT-86 displays the starting and ending addresses of each segment in the file loaded. Use the V command to redisplay this information at a later time.

If the file does not exist or cannot be successfully loaded in the available memory, DDT-86 issues an error message.

The F (Fill) Command

The F command fills an area of memory with a byte or word constant. The following are the forms of the F command:

F*s,f,b*
FW*s,f,w*

where *s* is a 20-bit starting address of the block to be filled, and *f* is a 16-bit offset of the final byte of the block within the segment specified in *s*.

In response to the first form, DDT-86 stores the 8-bit value *b* in locations *s* through *f*. In the second form, the 16-bit value *w* is stored in locations *s* through *f* in standard form, low 8 bits first followed by high 8 bits.

If *s* is greater than *f* or if the value *b* is greater than 255, DDT-86 responds with a question mark. DDT-86 issues an error message if the value stored in memory cannot be read back successfully, indicating faulty or nonexistent RAM at the location indicated.

The G (Go) Command

The G command transfers control to the program being tested, and optionally sets one or two breakpoints. The following are the forms of the G command:

G
G,*b1*
G,*b1,b2*
G*s*
G*s,b1*
G*s,b1,b2*

where *s* is a 20-bit address where program execution is to start, and *b1* and *b2* are 20-bit breakpoint addresses. If no segment value is supplied for any of these three addresses, the segment value defaults to the contents of the CS register.

In the first three forms, no starting address is specified; instead, DDT-86 derives the 20-bit address from the CS and IP registers. The first form transfers control to the program being tested without setting any breakpoints. The next two forms respec-

tively set one and two breakpoints before passing control to the user's program. The next three forms are analogous to the first three, except that the CS and IP registers are first set to *s*.

Once control has been transferred to the program under test, it executes in realtime until a breakpoint is encountered. At this point, DDT-86 regains control, clears all breakpoints, and indicates the address at which execution of the program under test was interrupted as follows:

**ssss:oooo*

where *ssss* corresponds to the CS and *oooo* corresponds to the IP where the break occurred. When a breakpoint returns control to DDT-86, the instruction at the breakpoint address has not yet been executed.

The H (Hexadecimal Math) Command

The H command computes the sum and difference of two 16-bit values. The following is the form of the H command:

H*a,b*

where *a* and *b* are the values whose sum and difference are to be computed. DDT-86 displays the sum (*ssss*) and the difference (*dddd*) truncated to 16 bits on the next line as shown below:

ssss dddd

The I (Input Command Tail) Command

The I command prepares a file control block and command tail buffer in DDT-86's base page, and copies this information into the base page of the last file loaded with the E command. The following is the form of the I command:

I*commandtail*

where the *commandtail* parameter is a character string that usually contains one or more filenames. The first filename is parsed into the default file control block at 005CH. The optional second filename (if specified) is parsed into the second part of the default file control block beginning at 006CH. The characters in the command-tail parameter are also copied into the default command buffer at 0080H. The length of *commandtail* is stored at 0080H, followed by the character string terminated with a binary zero.

If a file has been loaded with the E command, DDT-86 copies the file control block and command buffer from the base page of DDT-86 to the base page of the program loaded. The location of DDT-86's base page can be obtained from the SS register in the user's CPU state when DDT-86 is invoked. The location of the base page of a program loaded with the E command is the value displayed for DS upon completion of the program load.

The L (List) Command

The L command lists the contents of memory in assembly language. The following forms of the L command are used:

```
L
Ls
Ls,f
```

where *s* is the 20-bit address at which the list is to start, and *f* is a 16-bit offset within the segment specified in *s* where the list is to finish.

The first form lists twelve lines of disassembled machine code from the current list address. The second form sets the list address to *s*, then lists twelve lines of code. The last form lists disassembled code from *s* through *f*. In all three cases, the list address is set to the next unlisted location in preparation for a subsequent L command. When DDT-86 regains control from a program being tested (see G, T and U commands), the list address is set to the current value of the CS and IP registers.

To abort long displays, press any key during the list process. Press CTRL-S to halt the display temporarily.

The syntax of the assembly language statements produced by the L command is described later in the chapter.

The M (Move) Command

The M command moves a block of data values from one area of memory to another. The following is the form of the M command:

```
Ms,f,d
```

where *s* is the 20-bit starting address of the block to be moved, *f* is the offset of the final byte to be moved within the segment described by *s*, and *d* is the 20-bit address

of the first byte of the area to receive the data. If the segment is not specified in *d*, the value equals that used for *s*. Note that if the value of *d* is between *s* and *f*, part of the block being moved will be overwritten before it is moved, because data is transferred starting from location *s*.

The R (Read) Command

The R command reads a file into a contiguous block of memory. The following is the form of the R command:

R*filename*

where *filename* is the name and type of the file to be read.

DDT-86 reads the file into memory and displays the starting and ending addresses of the block of memory occupied by the file. Note that a V command can redisplay this information at a later time. The default display pointer (for subsequent D commands) is set to the start of the block occupied by the file.

The R command does not free any memory previously allocated by another R or E command. Therefore, a number of files can be read into memory without overlapping. The number of files that can be loaded is limited to seven, which is the number of memory allocations allowed by the BDOS, minus one for DDT-86 itself.

If the file does not exist or there is not enough memory to load the file, DDT-86 issues an error message.

The S (Set) Command

The S command can change the contents of bytes or words of memory. The following forms of the S command are used:

S*s*
SW*s*

where *s* is the 20-bit address at which the change is to occur.

DDT-86 displays the memory address and its current contents on the following line. In response to the first form, the display appears as follows:

ssss:oooo bb

In response to the second form, the display appears as follows:

ssss:oooo wwww

In the above examples, *bb* and *wwww* are the contents of memory in byte and word formats, respectively.

In response to one of the above displays, you can choose to alter the memory location or to leave it unchanged. If you enter a valid hexadecimal value, the contents of the byte (or word) in memory is replaced with the value. If no value is entered, the contents of memory are unaffected and the contents of the next address are displayed. In either case, DDT-86 continues to display successive memory addresses and values until either a period or an invalid value is entered.

DDT-86 issues an error message if the value stored in memory cannot be read back successfully, indicating faulty or nonexistent RAM at the location indicated.

The T (Trace) Command

The T command traces program execution for 1 to 0FFFFH program steps. The following forms of the T command are used:

T
T*n*
TS
TS*n*

where *n* is the number of instructions to execute before returning control to the console.

Before an instruction is executed, DDT-86 displays the current CPU state and the disassembled instruction. In the first two forms, the segment registers are not displayed, allowing the entire CPU state to be displayed on one line. The next two forms are analogous to the first two, except that all the registers are displayed, forcing the disassembled instruction to be displayed on the next line, as in the X command.

In all of the forms, control transfers to the program under test at the address indicated by the CS and IP registers. If *n* is not specified, one instruction is executed. Otherwise DDT-86 executes *n* instructions, displaying the CPU state before each step. To abort a long trace before *n* steps have been executed, press any key.

After a T command, the list address used in the L command is set to the address of the next instruction to be executed.

Note that DDT-86 does not trace through a BDOS interrupt instruction, since DDT-86 itself makes BDOS calls and the BDOS is not reentrant. Instead, the entire sequence of instructions from the BDOS interrupt through the return from BDOS is treated as one traced instruction.

The U (Untrace) Command

The U command is identical to the T command except that the CPU state is displayed only before the first instruction is executed, rather than before every step. The following forms of the U command are used:

U
U*n*
US
US*n*

where *n* is the number of instructions to execute before returning control to the console. To abort the U command before *n* steps have been executed, press any key.

The V (Value) Command

The V command displays information about the last file loaded with the E or R commands. The following is the form of the V command:

V

If the last file was loaded with the E command, the V command displays the starting and ending addresses of each of the segments contained in the file. If the last file was read with the R command, the V command displays the starting and ending addresses of the block of memory where the file was read. If neither the R nor E commands have been used, DDT-86 responds to the V command with a question mark (?).

The W (Write) Command

The W command writes the contents of a contiguous block of memory to disk. The following forms of the W command are used:

W*filename*
W*filename,s,f*

where *filename* is the filename and filetype of the disk file to receive the data, and *s* and *f* are the 20-bit first and last addresses of the block to be written. If the segment is not specified in *f*, DDT-86 uses the same value used for *s*.

If the first form is used, DDT-86 assumes the *s* and *f* values from the last file read with an R command. If no file was read with an R command, DDT-86 responds with a question mark (?). This form is useful for writing out files after patches have been installed, assuming the overall length of the file is unchanged.

In the second form, where *s* and *f* are specified as 20-bit addresses, the low four bits of *s* are assumed to be 0. Thus the block being written must always start on a paragraph boundary.

If the file with the name specified in the W command already exists, DDT-86 deletes it before writing a new file.

The X (Examine CPU State) Command

The X command allows the operator to examine and alter the CPU state of the program under test. The following forms of the X command are used:

```
X
Xr
Xf
```

where *r* is the name of one of the 8086 CPU registers and *f* is the abbreviation of one of the CPU flags. The first form displays the CPU state in the format:

```
         AX   BX   CX  . . .  SS   ES   IP
--------- XXXX XXXX XXXX . . . XXXX XXXX XXXX
<instruction>
```

The nine hyphens at the beginning of the line indicate the state of the nine CPU flags. Each position can be either a hyphen, indicating that the corresponding flag is not set (0), or a one-character abbreviation of the flag name, indicating that the flag is set (1). The abbreviations of the flag names are shown in Table 6-2. The value for *instruction* is the disassembled instruction at the next location to be executed, indicated by the CS and IP registers.

Table 6-2 Flag Name Abbreviations

CHARACTER	NAME
O	Overflow
D	Direction
I	Interrupt Enable
T	Trap
S	Sign
Z	Zero
A	Auxiliary Carry
P	Parity
C	Carry

The second form allows you to alter the registers in the CPU state of the program being tested. The *r* following the X is the name of one of the 16-bit CPU registers. DDT-86 responds by displaying the name of the register followed by its current value. If RETURN is pressed, the value of the register is not changed. If a valid value is typed, the contents of the register are changed to that value. In either case, the next register is then displayed. This process continues until a period or an invalid value is entered, or the last register is displayed.

The third form allows you to alter one of the flags in the CPU state of the program being tested. DDT-86 responds by displaying the name of the flag followed by its current state. If RETURN is pressed, the state of the flag is not changed. If a valid value is typed, the state of the flag is changed to that value. Only one flag may be examined or altered with each X*f* command. Set or reset flags by entering a value of 1 or 0.

DEFAULT SEGMENT VALUES

DDT-86 has an internal mechanism that keeps track of the current segment value, making segment specification an optional part of a DDT-86 command. DDT-86 divides the command set into two types of commands, according to which segment a command defaults to if no segment value is specified in the command line.

The first type of command pertains to the code segment: A (Assemble), L (List Mnemonics) and W (Write). These commands use the internal type-1 segment value if no segment value is specified in the command.

When invoked, DDT-86 sets the type-1 segment to 0, and changes it when one of the following actions is taken.

- When a file is loaded by an E command, DDT-86 sets the type-1 segment value to the value of the CS register.
- When a file is read-by an R command, DDT-86 sets the type-1 segment value to the base segment where the file was read.
- When an X command changes the value of the CS register, DDT-86 changes the type-1 segment value to the new value of the CS register.
- When DDT-86 regains control from a user program after a G, T or U command, it sets the type-1 segment value to the value of the CS register.
- When a segment value is specified explicitly in an A or L command, DDT-86 sets the type-1 segment value to the segment value specified.

The second type of command pertains to the data segment: D (Display), F (Fill), M (Move) and S (Set). These commands use the internal type-2 segment value if no segment value is specified in the command.

When invoked, DDT-86 sets the type-2 segment value to 0, and changes it when one of the following actions is taken.

- When a file is loaded by an E command, DDT-86 sets the type-2 segment value to the value of the DS register.
- When a file is read by an R command, DDT-86 sets the type-2 segment value to the base segment where the file was read.
- When an X command changes the value of the DS register, DDT-86 changes the type-2 segment value to the new value of the DS register.
- When DDT-86 regains control from a user program after a G, T or U command, it sets the type-2 segment value to the value of the DS register.
- When a segment value is specified explicitly in a D, F, M or S command, DDT-86 sets the type-2 segment value to the segment value specified.

When evaluating programs that use identical values in the CS and DS registers, all DDT-86 commands default to the same segment value unless explicitly overridden.

Note that the G (Go) command does not fall into either group, since it defaults to the CS register.

Table 6-3 summarizes DDT-86's default segment values.

Table 6-3 DDT-86 Default Segment Values

COMMAND	TYPE-1	TYPE-2
A	x	
D		x
E	c	c
F		x
G	c	c
H		
I		
L	x	
M		x
R	c	c
S		x
T	c	c
U	c	c
V		
W	x	
X	c	c

x - use this segment default if none specified;
change default if specified explicitly
c -change this segment default

ASSEMBLY LANGUAGE SYNTAX FOR A AND L COMMANDS

In general, the syntax of the assembly language statements used in the A and L commands is standard 8086 assembly language. Several minor exceptions are listed below.

- DDT-86 assumes that all numeric values entered are hexadecimal.
- Up to three prefixes (LOCK, repeat, segment override) may appear in one statement, but they all must precede the opcode of the statement. Alternately, a prefix may be entered on a line by itself.

- The distinction between byte and word string instructions is made as follows:

byte	word
LODSB	LODSW
STOSB	STOSW
SCASB	SCASW
MOVSB	MOVSW
CMPSB	CMPSW

- The mnemonics for near and far control transfer instructions are as follows:

short	normal	far
JMPS	JMP	JMPF
	CALL	CALLF
	RET	RETF

- If the operand of a CALLF or JMPF instruction is a 20-bit absolute address, it is entered in the following form:

  ssss:oooo

  where *ssss* is the segment and *oooo* is the offset of the address.

- Operands that could refer to either a byte or word are ambiguous, and must be preceded either by the prefix BYTE or WORD. These prefixes may be abbreviated to BY and WO. For example:

```
INC         BYTE [BP]
NOT         WORD [1234]
```

  Failure to supply a prefix when needed results in an error message.

- Operands which address memory directly are enclosed in square brackets to distinguish them from immediate values. For example:

```
ADD     AX,5        ;add 5 to register AX
ADD     AX,[5]      ;add the contents of location 5 to AX
```

- The following are forms of register indirect memory operands:

  [*pointer register*]
  [*index register*]
  [*pointer register* + *index register*]

  where the *pointer registers* are BX and BP, and the *index registers* are SI and DI. Any of these forms can be preceded by a numeric offset, as in the following examples.

```
ADD     BX,[BP+SI]
ADD     BX,3[BP+SI]
ADD     BX,1D47[BP+SI]
```

DDT-86 SAMPLE SESSION

In the following sample session, a simple sort program is interactively debugged. Comments that explain the steps involved are in italics.

Source file of program to test.

```
A>type sort.a86
;
;       simple sort program
;
sort:
        mov     si,0            ;initialize index
        mov     bx,offset nlist ;bx = base of list
        mov     sw,0            ;clear switch flag
comp:
        mov     al,[bx+si]      ;get byte from list
        cmp     al,1[bx+si]     ;compare with next byte
        jna     inci            ;don't switch if in order
        xchg    al,1[bx+si]     ;do first part of switch
        mov     [bx+si],al      ;do second part
        mov     sw,1            ;set switch flag
inci:
        inc     si              ;increment index
        cmp     si,count        ;end of list?
        jnz     comp            ;no, keep going
        test    sw,1            ;done - any switches?
        jnz     sort            ;yes, sort some more
```

```
done:
        jmp     done            ;get here when list ordered
;
        dseg
        org     100h            ;leave space for base page
;
nlist   db      3,8,4,6,31,6,4,1
count   equ     offset $ - offset nlist
sw      db      0
        end
```

Assemble program.

```
A>asm86 sort
CP/M 8086 ASSEMBLER VER 1.1
END OF PASS 1
END OF PASS 2
END OF ASSEMBLY. NUMBER OF ERRORS:  0
```

Type listing file generated by ASM-86

```
A>type sort.lst
CP/M ASM86 1.1  SOURCE: SORT.A86

                        ;
                        ;       simple sort program
                        ;
                        sort:
0000 BE0000                     mov     si,0            ;initialize index
0003 BB0001                     mov     bx,offset nlist ;bx = base of list
0006 C606080100                 mov     sw,0            ;clear switch flag
                        comp:
000B 8A00                       mov     al,[bx+si]      ;get byte from list
000D 3A4001                     cmp     al,1[bx+si]     ;compare with next byte
0010 760A                       jna     inci            ;don't switch if in order
0012 864001                     xchg    al,1[bx+si]     ;do first part of switch
0015 8800                       mov     [bx+si],al      ;do second part
0017 C606080101                 mov     sw,1            ;set switch flag
                        inci:
001C 46                         inc     si              ;increment index
001D 83FE08                     cmp     si,count        ;end of list?
```

```
0020 75E9                  jnz     comp            ;no, keep going
0022 F606080101            test    sw,1            ;done - any switches?
0027 75D7                  jnz     sort            ;yes, sort some more
                    done:
0029 E9FDFF                jmp     done            ;get here when list ordered
                    ;
                           dseg
                           org     100h            ;leave space for base page
                    ;
0100 030804061F06  nlist   db      3,8,4,6,31,6,4,1
     0401
0008               count   equ     offset $ - offset nlist
0108 00            sw      db      0
                           end
```

END OF ASSEMBLY. NUMBER OF ERRORS: 0

Type symbol table file generated by ASM-86.

```
A>type sort.sym
0000 VARIABLES
0100 NLIST     0108 SW
0000 NUMBERS
0008 COUNT
0000 LABELS
000B COMP      0029 DONE      001C INCI      0000 SORT
```

Type hex file generated by ASM-86.

```
A>type sort.h86
:0400000300000000F9
:1B000081BE0000BB0001C6060801008A003A4001760A8640018800C60608016C
:11001B81014683FE0875E9F60608010175D7E9FDFFEE
:0901008203080406 1F0604010035
:00000001FF
```

Generate CMD file from H86 file.

```
A>gencmd sort
BYTES READ        0039
RECORDS WRITTEN 04
```

Invoke DDT-86 and load SORT.CMD.

```
A>ddt86 sort
DDT86 1.0
      START     END
CS  047D:0000 047D:002F
DS  0480 :0000 0480 :010F
```

Display initial register values.

```
-x
            AX  BX  CX  DX  SP  BP  SI  DI  CS  DS  SS  ES  IP
---------   0000 0000 0000 0000 119E 0000 0000 0000 047D 0480 0491 0480 0000
MOV  SI, 0000
```

Disassemble the beginning of the code segment.

```
-l
047D:0000  MOV    SI,0000
047D:0003  MOV    BX,0100
047D:0006  MOV    BYTE [0108],00
047D:000B  MOV    AL,[BX+SI]
047D:000D  CMP    AL,01[BX+SI]
047D:0010  JBE    001C
047D:0012  XCHG   AL,01[BX+SI]
047D:0015  MOV    [BX+SI],AL
047D:0017  MOV    BYTE [0108],01
047D:001C  INC    SI
047D:001D  CMP    SI,0008
047D:0020  JNZ    000B
```

Display the start of the data segment.

```
-d100,10f
0480:0100 03 08 04 06 1F 06 04 01 00 00 00 00 00 00 00 00 ................
```

Disassemble the rest of the code.

```
-l
047D:0022  TEST   BYTE [0108],01
047D:0027  JNZ    0000
047D:0029  JMP    0029
047D:002C  ADD    [BX+SI],AL
047D:002E  ADD    [BX+SI],AL
047D:0030  DAS
```

```
047D:0031  ADD     [BX+SI],AL
047D:0033  ??=     6C
047D:0034  POP     ES
047D:0035  ADD     [BX],CL
047D:0037  ADD     [BX+SI],AX
047D:0039  ??=     6F
```

Execute program from IP (=0) setting breakpoint at 29H.

```
-g,29
*047D:0029
```

Breakpoint encountered. Display sorted list.

```
-d100,10f
0480:0100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ............
```

Doesn't look good. Reload file.

```
-esort
      START     END
CS  047D:0000  047D:002F
DS  0480 :0000  0480 :010F
```

Trace 3 instructions.

```
-t3
             AX  BX  CX  DX  SP  BP  SI  DI  IP
-----Z-P-    0000 0100 0000 0000 119E 0000 0008 0000 0000  MOV   SI,0000
-----Z-P-    0000 0100 0000 0000 119E 0000 0000 0000 0003  MOV   BX,0100
-----Z-P-    0000 0100 0000 0000 119E 0000 0000 0000 0006  MOV   BYTE [0108],00
*047D:000B
```

Trace some more.

```
-t3
             AX  BX  CX  DX  SP  BP  SI  DI  IP
-------Z-P   0000 0100 0000 0000 119E 0000 0000 0000 000B  MOV   AL, [BX + SI]
-------Z-P   0003 0100 0000 0000 119E 0000 0000 0000 000D  CMP   AL, 01 [BX + SI]
----S-A-C    0003 0100 0000 0000 119E 0000 0000 0000 0010  JBE   001C
*047D:001C
```

Display unsorted list.

```
-d100,10f
0480:0100 03 08 04 06 1F 06 04 01 00 00 00 00 00 00 00 00  ............
```

Display next instructions to be executed.

```
-l
047D:001C INC     SI
047D:001D CMP     SI,0008
047D:0020 JNZ     000B
047D:0022 TEST    BYTE [0108],01
047D:0027 JNZ     0000
047D:0029 JMP     0029
047D:002C ADD     [BX+SI],AL
047D:002E ADD     [BX+SI],AL
047D:0030 DAS
047D:0031 ADD     [BX+SI],AL
047D:0033 ??=     6C
047D:0034 POP     ES
```

Trace some more.

```
-t3
          AX   BX   CX   DX   SP   BP   SI   DI   IP
----S-A-C 0003 0100 0000 0000 119E 0000 0000 0000 001C INC   SI
----------C 0003 0100 0000 0000 119E 0000 0001 0000 001D CMP   SI,0008
---S-APC 0003 0100 0000 0000 119E 0000 0001 0000 0020 JNZ   000B
*047D:000B
```

Display instructions from current IP.

```
-l
047D:000B MOV     AL,[BX+SI]
047D:000D CMP     AL,01[BX+SI]
047D:0010 JBE     001C
047D:0012 XCHG    AL,01[BX+SI]
047D:0015 MOV     [BX+SI],AL
047D:0017 MOV     BYTE [0108],01
047D:001C INC     SI
047D:001D CMP     SI,0008
D47D:0020 JNZ     000B
047D:0022 TEST    BYTE [0108],01
047D:0027 JNZ     0000
047D:0029 JMP     0029
```

```
-t3
            AX  BX  CX  DX  SP   BP   SI   DI   IP
----S-APC   0003 0100 0000 0000 119E 0000 0001 0000 000B   MOV   AL,[BX+SI]
----S-APC   0008 0100 0000 0000 119E 0000 0001 0000 000D   CMP   AL,01[BX+SI]
---------   0008 0100 0000 0000 119E 0000 0001 0000 0010   JBE   001C
*047D:0012

-l
047D:0012  XCHG   AL,01[BX+SI]
047D:0015  MOV    [BX+SI],AL
047D:0017  MOV    BYTE [0108],01
047D:001C  INC    SI
047D:001D  CMP    SI,0008
047D:0020  JNZ    000B
047D:0022  TEST   BYTE [0108],01
047D:0027  JNZ    0000
047D:0029  JMP    0029
047D:002C  ADD    [BX+SI],AL
047D:002E  ADD    [BX+SI],AL
047D:0030  DAS
```

Go until switch has been performed.

```
-g,20
*047D:0020
```

Display list.

```
-d100.10f
0480:0100 03 04 08 06 1F 06 04 01 01 00 00 00 00 00 00 00 ............
```

Looks like 4 and 8 were switched okay. (And toggle is true.)

```
-t
            AX  BX  CX  DX  SP   BP   SI   DI   IP
----S-APC   0004 0100 0000 0000 119E 0000 0002 0000 0020   JNZ   000B
*047D:000B
```

Display next instructions.

```
-l
047D:000B  MOV    AL,[BX+SI]
047D:000D  CMP    AL,01[BX+SI]
047D:0010  JBE    001C
047D:0012  XCHG   AL,01[BX+SI]
047D:0015  MOV    [BX+SI],AL
047D:0017  MOV    BYTE [0108],01
047D:001C  INC    SI
047D:001D  CMP    SI,0008
047D:0020  JNZ    000B
047D:0022  TEST   BYTE [0108],01
047D:0027  JNZ    0000
047D:0029  JMP    0029
```

Since switch worked, reload and check boundary conditions.

```
-esort
      START     END
CS  047D:0000 047D:002F
DS  0480:0000  0480 :010F
```

*Make it quicker by setting list length to 3. (Could also have used s47d=*1e.)

```
-a1d
047D:001D cmp si,3
047D:0020
```

Display unsorted list.

```
-d100
0480:0100 03 08 04 06 1F 06 04 01 00 00 00 00 00 00 00 00 ............
0480:0110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ............
0480:0120 00 00 00 00 00 00 00 00 00 00 00 00 00 00 20 20 20 ............
```

Set breakpoint when first 3 elements of list should be sorted.

```
-g,29
*047D:0029
```

See if list is sorted.

```
-d100,10f
0480:0100 03 04 06 08 1F 06 04 01 00 00 00 00 00 00 00 00 ............
```

Interesting. The fourth element seems to have been sorted in.

```
-esort
      START     END
CS  047D:0000  047D:002F
DS  0480 :0000  0480 :010F
```

Let's try again with some tracing.

```
-a1d
047D:001D cmp si,3
047D:0020 .

-t9
             AX  BX  CX  DX  SP   BP   SI   DI   IP
-----Z-P-    0006 0100 0000 0000 119E 0000 0003 0000 0000  MOV   SI,0000
-----Z-P-    0006 0100 0000 0000 119E 0000 0000 0000 0003  MOV   BX,0100
-----Z-P-    0006 0100 0000 0000 119E 0000 0000 0000 0006  MOV   BYTE [0108],00
-----Z-P-    0006 0100 0000 0000 119E 0000 0000 0000 000B  MOV   AL,[BX+SI]
-----Z-P-    0003 0100 0000 0000 119E 0000 0000 0000 000D  CMP   AL,01[BX+SI]
----S-A-C    0003 0100 0000 0000 119E 0000 0000 0000 0010  JBE   001C
----S-A-C    0003 0100 0000 0000 119E 0000 0000 0000 001C  INC   SI
----------C  0003 0100 0000 0000 119E 0000 0001 0000 001D  CMP   SI,0003
----S-A-C    0003 0100 0000 0000 119E 0000 0001 0000 0020  JNZ   000B
*047D:000B

-l
047D:000B  MOV    AL,[BX+SI]
047D:000D  CMP    AL,O1[BX+SI]
047D:0010  JBE    001C
047D:0012  XCHG   AL,01[BX+SI]
047D:0015  MOV    [BX+SI],AL
047D:0017  MOV    BYTE [0108],01
047D:001C  INC    SI
047D:001D  CMP    S1,0003
047D:0020  JNZ    000B
047D:0022  TEST   BYTE [0108],01
047D:0027  JNZ    0000
047D:0029  JMP    0029
```

```
-t3
            AX  BX  CX  DX  SP  BP  SI  DI  IP
----S-A-C  0003 0100 0000 0000 119E 0000 0001 0000 000B  MOV    AL,[BX+SI]
----S-A-C  0008 0100 0000 0000 119E 0000 0001 0000 000D  CMP    AL,01[BX+SI]
---------  0008 0100 0000 0000 119E 0000 0001 0000 0010  JBE    001C
*047D:0012

-l
047D:0012 XCHG   AL,01[BX+SI]
047D:0015 MOV    [BX+SI],AL
047D:0017 MOV    BYTE [0108],01
047D:001C INC    SI
047D:001D CMP    SI,0003
047D:0020 JNZ    000B
047D:0022 TEST   BYTE [0108],01

-t3
            AX  BX  CX  DX  SP  BP  SI  DI  IP
---------  0008 0100 0000 0000 119E 0000 0001 0000 0012  XCHG   AL,01[BX+SI]
---------  0004 0100 0000 0000 119E 0000 0001 0000 0015  MOV    [BX+SI],AL
---------  0004 0100 0000 0000 119E 0000 0001 0000 0017  MOV    BYTE [ 0108],01
*047D:001C

-d100,10 f
0480:0100 03 04 08 06 1F 06 04 01 01 00 00 00 00 00 00 00 ................
```

So far, so good.

```
-t3
            AX  BX  CX  DX  SP  BP  SI  DI  IP
---------  0004 0100 0000 0000 119E 0000 0001 0000 001C  INC    SI
---------  0004 0100 0000 0000 119E 0000 0002 0000 001D  CMP    SI,0003
--S-APC    0004 0100 0000 0000 119E 0000 0002 0000 0020  JNZ    000B
*047D:000B

-l
047D:000B MOV    AL,[BX+SI]
047D:000D CMP    AL,01[BX+SI]
```

```
047D:0010 JBE     001C
047D:0012 XCHG    AL,01[BX+SI]
047D:0015 MOV     [BX+SI],AL
047D:0017 MOV     BYTE [0108],01
047D:001C INC     SI
047D:001D CMP     SI,0003
047D:0020 JNZ     000B
047D:0022 TEST    BYTE [0108],01
047D:0027 JNZ     0000
047D:0029 JMP     0029
```

```
-t3
            AX  BX  CX  DX  SP  BP  SI  DI  IP
----S-APC 0004 0100 0000 0000 119E 0000 0002 0000 000B MOV   AL,[BX+SI]
----S-APC 0008 0100 0000 0000 119E 0000 0002 0000 000D CMP   AL,01[BX+SI]
--------- 0008 0100 0000 0000 119E 0000 0002 0000 0010 JBE   001C
*047D:0012
```

Sure enough, it's comparing the third and fourth elements of the list. Reload program.

```
-esort
   START       END
CS 047D:0000 047D:002F
DS 0480 :0000 0480 :010F

-l
047D:0000 MOV     SI,0000
047D:0003 MOV     BX,0100
047D:0006 MOV     BYTE [0108],00
047D:000B MOV     AL,[BX+SI]
047D:000D CMP     AL,01[BX+SI]
047D:0010 JBE     001C
047D:0012 XCHG    AL,01[BX+SI]
047D:0015 MOV     [BX+SI],AL
047D:0017 MOV     BYTE [0108],01
047D:001C INC     SI
047D:001D CMP     SI,0008
047D:0020 JNZ     000B
```

Patch length.

```
-a1d
047D:001D     cmp si,7
047D:0020
```

Try it out.

```
-g,29
*047D:0029
```

See if list is sorted.

```
-d100,10f
0480:0100 01 03 04 04 06 06 08 1F 00 00 00 00 00 00 00 00 ................
```

Looks better. Install patch in disk file. To do this, must read CMD file including header, so we can use R command.

```
-rsort.cmd
  START             END
2000:0000       2000:01FF
```

First 80h bytes contain header, so code starts at 80h.

```
-l80
2000:0080  MOV     SI,0000
2000:0083  MOV     BX,0100
2000:0086  MOV     BYTE [0108],00
2000:008B  MOV     AL,[BX+SI]
2000:008D  CMP     AL,01[BX+SI]
2000:0090  JBE     009C
2000:0092  XCHG    AL,01[BX+SI]
2000:0095  MOV     [BX+SI],AL
2000:0097  MOV     BYTE [0108],01
2000:009C  INC     SI
2000:009D  CMP     SI,0008
2000:00A0  JNZ     008B
```

Install patch.

```
-a9d
2000:009D       cmp si,7
2000:00A0
```

Write file back to disk. (Length of file assumed to be unchanged since no length specified).

```
-wsort.cmd
```

Reload file.

```
-esort
    START          END
CS 047D:0000 047D:002F
DS 0480 :0000 0480 :010F
```

Verify that patch was installed.

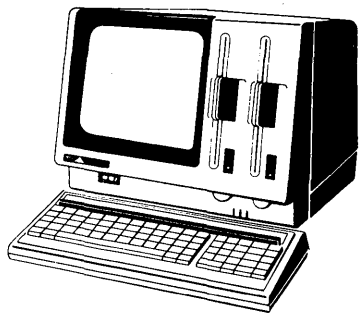
```
-l
047D:0000  MOV     SI,0000
047D:0003  MOV     BX,0100
047D:0006  MOV     BYTE [0108],00
047D:000B  MOV     AL,[BX+SI]
047D:000D  CMP     AL,01[BX+SI]
047D:0010  JBE     001C
047D:0012  XCHG    AL,01[BX+SI]
047D:0015  MOV     [BX+SI],AL
047D:0017  MOV     BYTE [0108],01
047D:001C  INC     SI
047D:001D  CMP     SI,0007
047D:0020  JNZ     000B
```

Run it.

```
-g,29
*047D:0029
```

Still looks good. Ship it!

```
-d100,10f
0480:0100 01 03 04 04 06 06 08 1F 00 00 00 00 00 00 00 00 ................

-^C

A>
```



Appendix A

ASM-86 Invocation

Command:

ASM86

Syntax:

ASM86 *filename* {$ *parameters*}

where *filename*	is the 8086 assembly source file. Drive and extension are optional. The default file extension is .A86.
parameters	are a one-letter type followed by a one-letter device from the table below.

Parameters:

form: $ T*d* where T = type and *d* = device

Table A-1 Parameter Types and Devices

DEVICES			PARAMETERS		
	A	H	P	S	F
A - H	x	x	x	x	
X	x	x	x	x	
Y	x	x	x	x	
x = valid, d = default					

Table A-1 Parameter Types and Devices (cont'd)

DEVICES			PARAMETERS		
	A	H	P	S	F
Z	x	x	x	x	
I					x
D					d
x = valid, d = default					

Valid Parameters

Except for the F type, the default device is the current default drive.

Table A-2 Parameter Types

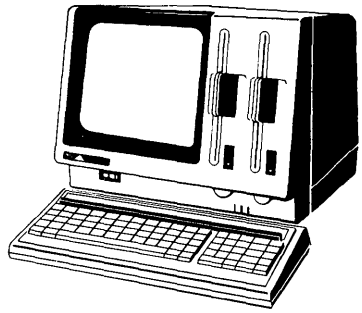
A	controls location of ASSEMBLER source file
H	controls location of HEX file
P	controls location of PRINT file
S	controls location of SYMBOL file
F	controls type of hex output FORMAT

Table A-3 Device Types

A - H	Drives A - H
X	console device
Y	printer device
Z	byte bucket
I	Intel hex format
D	Digital Research hex format

Table A-4 Invocation Examples

ASM86 IO	Assemble file IO.A86, produce IO.HEX IO.LST and IO.SYM.
ASM86 IO.ASM $ AD SZ	Assemble file IO.ASM on device D, produce IO.LST and IO.HEX, no symbol file.
ASM86 IO $ PY SX	Assemble file IO.A86, produce IO.HEX, route listing directly to printer, output symbols on console.
ASM86 IO $ FD	Produce Digital Research hex format.
ASM86 IO $ FI	Produce Intel hex format.



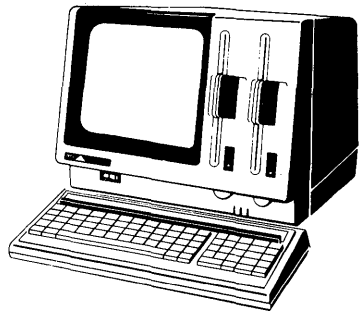
Appendix B

Mnemonic Differences from the Intel Assembler

The CP/M 8086 assembler uses the same instruction mnemonics as the INTEL 8086 assembler except for explicitly specifying far and short jumps, calls and returns. The following table shows the four differences.

Table B-1 Mnemonic Differences

MNEMONIC FUNCTION	CP/M	INTEL
Intra segment short jump:	JMPS	JMP
Inter segment jump:	JMPF	JMP
Inter segment return:	RETF	RET
Inter segment call:	CALLF	CALL



Appendix C

ASM-86 Files

ASM-86 HEXADECIMAL OUTPUT FORMAT

ASM-86 produces machine code in either Intel or Digital Research hexadecimal format. The Intel format is identical to the format defined by Intel for the 8086. The Digital Research format is nearly identical to the Intel format, but adds segment information to hexadecimal records. Output of either format can be input to GENCMD, but the Digital Research format automatically provides segment identification. A segment is the smallest unit of a program that can be relocated.

Table C-1 defines the sequence and contents of bytes in a hexadecimal record. Each hexadecimal record has one of the four formats shown in Table C-2. An example of a hexadecimal record is shown below.

Byte number = > 0 1 2 3 4 5 6 7 8 9 .....n

Contents = > : l l a a a a t t d d d .....c c CR LF

Table C-1 Hexadecimal Record Contents

BYTE	CONTENTS	SYMBOL
0	record mark	:
1-2	record length	l l
3-6	load address	a a a a
7-8	record type	t t
9-(n-1)	data bytes	d d .....d
n-(n+1)	check sum	c c
n+2	carriage return	CR
n+3	line feed	LF

Table C-2 Hexadecimal Record Formats

RECORD TYPE	CONTENT	FORMAT
00	Data record	:11 aaaa DT <data...> cc
01	End-of-file	:00 0000 01 FF
02	Extended address mark	:02 0000 ST ssss cc
03	Start address	:04 0000 03 ssss iiii cc

ll	=> record length — number of data bytes
cc	=> check sum — sum of all record bytes
aaaa	=> 16 bit address
ssss	=> 16 bit segment value
iiii	=> offset value of start address
DT	=> data record type
ST	=> segment address record type

It is in the definition of record types 00 and 02 that Digital Research's hexadecimal format differs from Intel's. Intel defines one value each for the data record type and the segment address type. Digital Research identifies each record with the segment that contains it, as shown in Table C-3.

Table C-3 Segment Record Types

SYMBOL	INTEL'S VALUE	DIGITAL'S VALUE	MEANING
DT	00		for data belonging to all 8086 segments
		81H	for data belonging to the CODE segment
		82H	for data belonging to the DATA segment
		83H	for data belonging to the STACK segment
		84H	for data belonging to the EXTRA segment

Table C-3 Segment Record Types (cont'd)

SYMBOL	INTEL'S VALUE	DIGITAL'S VALUE	MEANING
ST	02		for all segment address records
		85H	for a CODE absolute segment address
		86H	for a DATA segment address
		87H	for a STACK segment address
		88H	for a EXTRA segment address

ASM-86 SYMBOL FILE FORMAT

The .SYM file produced by ASM-86 has the following characteristics.

- There is a form feed at the start of the file.
- Symbols are alphabetized within groups.
- Tabs are expanded if symbols are sent to the printer ($SY).

INCLUDE FILES

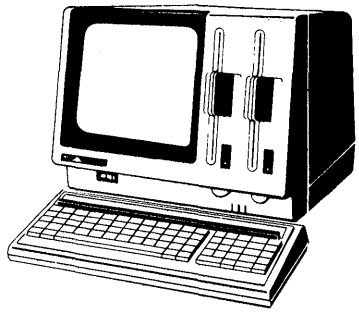
INCLUDE files have the following characteristics.

- The filetype defaults to .A86 if no extension is specified.
- The filetype may have fewer than three characters.
- The system defaults to the same drive as the source file when $A is used in the command.
- ASM-86 aborts if the file is not found.

ASM-86 List File Format

The .LST file produced by ASM-86 has the following characteristics.

- There is a form feed at the start of the file.
- There is no form feed at end of file.
- There is no <cr><lf> at the top of each page.
- An absolute address field is given for relative instructions.
- No spaces are placed between hex bytes. This allows more space for comments.
- Errors are printed when NOLIST is active.



Appendix D

Reserved Words

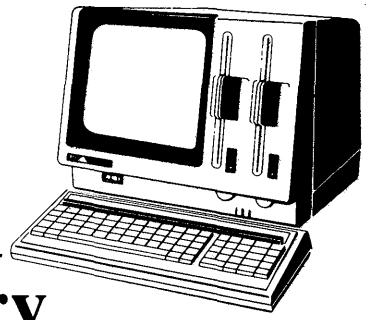
Table D-1 Reserved Words

Predefined Numbers				
BYTE	WORD	DWORD		
Operators				
EQ	GE	GT	LE	LT
NE	OR	AND	MOD	NOT
PTR	SEG	SHL	SHR	XOR
LAST	TYPE	LENGTH	OFFSET	
Assembler Directives				
DB	DD	DW	IF	RS
RB	RW	END	ENDM	EQU
ORG	CSEG	DSEG	ESEG	SSEG
EJECT	ENDIF	TITLE	LIST	NOLIST
INCLUDE	SIMFORM	PAGESIZE	CODEMACRO	PAGEWIDTH
Codemacro Directives				
DB	DD	DW	DBIT	RELB
RELW	MODRM	SEGFIX	NOSEGFIX	

Table D-1 Reserved Words (cont'd)

8086 Registers				
AH	AL	AX	BH	BL
BP	BX	CH	CL	CS
CX	DH	DI	DL	DS
DX	ES	SI	SP	SS

Instruction Mnemonics - See Appendix E.



Appendix E

ASM-86 Instruction Summary

Table E-1 ASM-86 Instruction Summary

MNEMONIC	DESCRIPTION
AAA	ASCII Adjust for Addition
AAD	ASCII Adjust for Division
AAM	ASCII Adjust for Multiplication
AAS	ASCII Adjust for Subtraction
ADC	Add with Carry
ADD	Add
AND	And
CALL	Call (intra segment)
CALLF	Call (inter segment)
CBW	Convert Byte to Word
CLC	Clear Carry
CLD	Clear Direction
CLI	Clear Interrupt
CMC	Complement Carry
CMP	Compare
CMPS	Compare Byte or Word (of string)
CMPSB	Compare String in Byte Form
CMPSW	Compare String in Word Form
CWD	Convert Word to Double Word
DAA	Decimal Adjust for Addition
DAS	Decimal Adjust for Subtraction
DEC	Decrement
DIV	Divide
ESC	Escape
HLT	Halt
IDIV	Integer Divide
IMUL	Integer Multiply
IN	Input Byte or Word
INC	Increment
INT	Interrupt

Table E-1 ASM-86 Instruction Summary (cont'd)

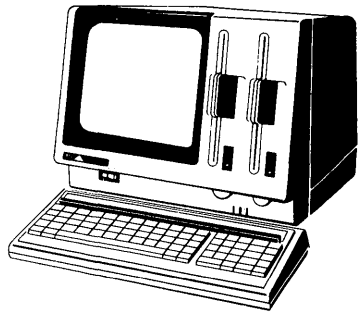
MNEMONIC	DESCRIPTION
INTO	Interrupt on Overflow
IRET	Interrupt Return
JA	Jump on Above
JAE	Jump on Above or Equal
JB	Jump on Below
JBE	Jump on Below or Equal
JC	Jump on Carry
JCXZ	Jump on CX Zero
JE	Jump on Equal
JG	Jump on Greater
JGE	Jump on Greater or Equal
JL	Jump on Less
JLE	Jump on Less or Equal
JMP	Jump (intra segment)
JMPF	Jump (inter segment)
JMPS	Jump (8 bit displacement)
JNA	Jump on Not Above
JNAE	Jump on Not Above or Equal
JNB	Jump on Not Below
JNBE	Jump on Not Below or Equal
JNC	Jump on Not Carry
JNE	Jump on Not Equal
JNG	Jump on Not Greater
JNGE	Jump on Not Greater or Equal
JNL	Jump on Not Less
JNLE	Jump on Not Less or Equal
JNO	Jump on Not Overflow
JNP	Jump on Not Parity
JNS	Jump on Not Sign
JNZ	Jump on Not Zero
JO	Jump on Overflow
JP	Jump on Parity
JPE	Jump on Parity Even
JPO	Jump on Parity Odd
JS	Jump on Sign
JZ	Jump on Zero
LAHF	Load AH with Flags

Table E-1 ASM-86 Instruction Summary (cont'd)

MNEMONIC	DESCRIPTION
LDS	Load Pointer into DS
LEA	Load Effective Address
LES	Load Pointer into ES
LOCK	Lock Bus
LODS	Load Byte or Word (of string)
LODSB	Load String in Byte Form
LODSW	Load String in Word Form
LOOP	Loop
LOOPE	Loop While Equal
LOOPNE	Loop While Not Equal
LOOPNZ	Loop While Not Zero
LOOPZ	Loop While Zero
MOV	Move
MOVS	Move Byte or Word (of string)
MOVSB	Move String in Byte Form
MOVSW	Move String in Word Form
MUL	Multiply
NEG	Negate
NOT	Not
OR	Or
OUT	Output Bye or Word
POP	Pop
POPF	Pop Flags
PUSH	Push
PUSHF	Push Flags
RCL	Rotate through Carry Left
RCR	Rotate through Carry Right
REP	Repeat
RET	Return (intra segment)
RETF	Return (inter segment)
ROL	Rotate Left
ROR	Rotate Right
SAHF	Store AH into Flags
SAL	Shift Arithmetic Left
SAR	Shift Arithmetic Right

Table E-1 ASM-86 Instruction Summary (cont'd)

MNEMONIC	DESCRIPTION
SBB	Subtract with Borrow
SCAS	Scan Byte or Word (of string)
SCASB	Scan String in Byte Form
SCASW	Scan String in Word Form
SHL	Shift Left
SHR	Shift Right
STC	Set Carry
STD	Set Direction
STI	Set Interrupt
STOS	Store Byte or Word (of string)
STOSB	Store String in Byte Form
STOSW	Store String in Word Form
SUB	Subtract
TEST	Test
WAIT	Wait
XCHG	Exchange
XLAT	Translate
XOR	Exclusive Or



Appendix F

Sample Program

Listing F-1 Sample Program APPF.A86

```
CP/M ASM86 1.1   SOURCE: APPF.A86   Terminal Input/Output   PAGE 1

                title 'Terminal Input/Output'
                pagesize 50
                pagewidth 79
                simform
                ;****** Terminal I/O subroutines********
                ;
                ;           The following subroutines
                ;           are included:
                ;
                ;           CONSTAT  — console status
                ;           CONIN    — console input
                ;           CONOUT   — console output
                ;
                ;           Each routine requires CONSOLE NUMBER
                ;           in the BL — register
                ;
                ;
                ;           ***************
                ;           * Jump table:  *
                ;           ***************
                ;
                CSEG            start of code segment
                ;
                ;
```

```
                        jmp_tab:
0000 E90600                     jmp constat
0003 E91900                     jmp conin
0006 E92B00                     jmp conout
                        ;
                        ;
                        ;       **********************
                        ;       *   I/O port numbers   *
                        ;       **********************
```

CP/M ASM86 1.1 SOURCE: APPF.A86 Terminal Input/Output PAGE 2

```
                        ;
                        ;               Terminal 1:
0010                    instat1         equ     10h     ; input status port
0011                    indata1         equ     11h     ; input port
0011                    outdata1        equ     11h     ; output port
0001                    readyinmask1    equ     01h     ; input ready mask
0002                    readyoutmask1   equ     02h     ; output ready mask
                        ;
                        ;               Terminal 2:
                        ;
0012                    instat2         equ     12h     ; input status port
0013                    indata2         equ     13h     ; input port
0013                    outdata2        equ     13h     ; output port
0004                    readyinmask2    equ     04h     ; input ready mask
0008                    readyoutmask2   equ     08h     ; output ready mask
                        ;
                        ;
                        ;               *************
                        ;               * CONSTAT *
                        ;               *************
                        ;
                        ;
                        ;               Entry: BL -reg = terminal no
                        ;               Exit: AL - reg = 0 if not ready
                        ;                                0ffh if ready
                        constat:
0009 53E83F00                   push bx ! call okterminal
                        constat1:
000D 52                         push dx
```

```
000E B600          mov  dh,0                    ; read status port
0010 8A17          mov  dl,instatustab [BX]
0012 EC            in   al,dx
0013 224706        and  al,readyinmasktab [bx]
0016 7402          jz   constatout
0018 B0FF          mov  a1,0ffh
```

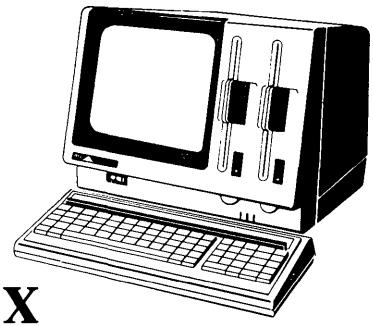
CP/M ASM86 1.1 SOURCE: APPF.A86 Terminal Input/Output PAGE 3

```
                   constatout:
001A 5A5B0AC0C3                 pop dx ! pop bx ! or al,al ! ret
                   ;
                   ;
                   ;            **********
                   ;            * CONIN *
                   ;            **********
                   ;
                   ;
                   ;            Entry: BL - reg = terminal no
                   ;            Exit: AL - reg = read character
                   ;
001F 53E82900      conin:       push bx ! call okterminal !
0023 E8E7FF        conin1:      call  constat1              ; test status
0026 74FB                       jz    conin1
0028 52                         push dx                     ; read character
0029 B600                       mov dh,0
002B 8A5702                     mov dl,indatatab [BX]
002E EC                         in    al,dx
002F 247F                       and   al,7fh                ; strip parity bit
0031 5A5BC3                     pop   dx ! pop bx ! ret
                   ;
                   ;
                   ;            ************
                   ;            * CONOUT *
                   ;            ************
                   ;
                   ;
                   ;            Entry: BL - reg = terminal no
                   ;                   AL - reg = character to print
                   ;
```

```
0034 53E81400      conout:     push bx ! call okterminal
0038 52                        push dx
0039 50                        push ax
003A B600                      mov dh,0                ; test status
003C 8A17                      mov dl,instatustab [BX]
                   conout1:
003E EC                        in al,dx

CP/M ASM86 1.1   SOURCE: APPF.A86   Terminal Input/Output   PAGE 4

003F 224708                    and  al,readyoutmasktab[BX]
0042 74FA                      jz   conout1
0044 58                        pop  ax                 ; write byte
0045 8A5704                    mov  dl,outdatatab [BX]
0048 EE                        out  dx,al
0049 5A5BC3                    pop  dx ! pop bx ! ret
                   ;
                   ;
                   ;           ++++++++++++++
                   ;           + OKTERMINAL +
                   ;           ++++++++++++++
                   ;
                   ;
                   ;           Entry: BL - reg = terminal no
                   ;
                   okterminal:
004C 0ADB                  or   bl,bl
004E 740A                  jz   error
0050 80FB03                cmp  bl,length instatustab + 1
0053 7305                  jae  error
0055 FECB                  dec  bl
0057 B700                  mov  bh,0
0059 C3                    ret
                   ;
005A 5B5BC3        error: pop bx ! pop bx ! ret        ; do nothing
                   ;
                   ;************** end of code segment ***************
                   ;
                   ;
                   ;           ****************
                   ;           * Data segment *
                   ;           ****************
```

```
                        dseg
                ;
                ;       *************************
                ;       *  Data for each terminal  *
                ;       *************************
                ;
                ;
0000 1012       instatustab     db      instat1,instat2
0002 1113       indatatab       db      indata1,indata2
0004 1113       outdatatab      db      outdata1,outdata2
0006 0104       readyinmasktab  db      readyinmask1,readyinmask2
0008 0208       readyoutmasktab db      readyoutmask1,readyoutmask2
                ;
                ;***************end of file**********************
                end
```

END OF ASSEMBLY. NUMBER OF ERRORS: 0



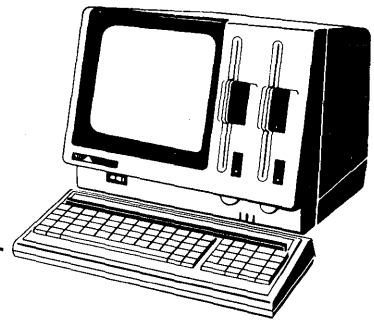
Appendix G

Codemacro Definition Syntax

```
<codemacro> : : = CODEMACRO <name> [<formal$list>]
                  [<list$of$macro$directives>]
                  ENDM

<name> : : = IDENTIFIER

<formal$list> : : = <parameter$descr>[{,<parameter$descr>}]

<parameter$descr> : : = <form$name>:<specifier$letter>
                        <modifier$letter>[(<range>)]

<specifier$letter> : : = A | C | D | E | M | R | S | X

<modifier$letter> : : = b | w | d | sb

<range> : : = <single$range>|<double$range>

<single$range> : : = REGISTER | NUMBERB

<double$range> : : =    NUMBERB, NUMBERB | NUMBERB, REGISTER |
                        REGISTER, NUMBERB | REGISTER, REGISTER

<list$of$macro$directives> : : = <macro$directive>
                                 {<macro$directive>}

<macro$directive> : : = <db> | <dw> | <dd> | <segfix> |
                        <ncsegfix> | <modrm> | <relb> |
                        <relw> | <dbit>

<db> : : = DB NUMBERB | DB <form$name>
```

```
<dw> : : = DW NUMBERW | DW <form$name>

<dd> : : = DD <form$name>

<segfix> : : = SEGFIX <form$name>

<nosegfix> : : = NOSEGFIX <form$name>

<modrm> : : = MODRM NUMBER7, <form$name> |
              MODRM <form$name>, <form$name>

<relb> : : = RELB <form$name>

<relw> : : = RELW <form$name>

<dbit> : : = DBIT <field$descr>{,<field$descr>}

<field$descr> : : = NUMBER15 ( NUMBERB ) |
                    NUMBER15 ( <form$name> ( NUMBERB ) )

<form$name> : : = IDENTIFIER
```

NUMBERB is 8 - bits
NUMBERW is 16 - bits
NUMBER7 are the values 0, 1, ..., 7
NUMBER15 are the values 0, 1, ..., 15



Appendix H

ASM-86 Error Messages

ASM-86 produces two types of error messages: fatal errors and diagnostics. Fatal errors occur when ASM-86 is unable to continue assembling. Diagnostic messages report problems with the syntax and semantics of the program being assembled. The following messages indicate fatal errors encountered by ASM-86 during assembly:

NO FILE
DISK FULL
DIRECTORY FULL
DISK READ ERROR
CANNOT CLOSE
SYMBOL TABLE OVERFLOW
PARAMETER ERROR

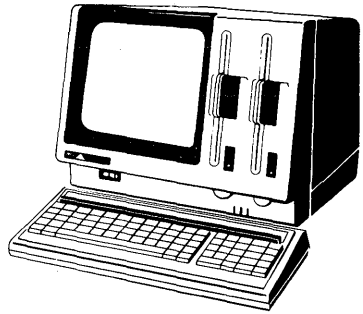
ASM-86 reports semantic and syntax errors by placing a numbered ASCII message in front of the erroneous source line. If there is more than one error in the line, only the first one is reported. Table H-1 summarizes ASM-86 diagnostic error messages.

Table H-1 ASM-86 Diagnostic Error Messages

NUMBER	MEANING
0	ILLEGAL FIRST ITEM
1	MISSING PSEUDO INSTRUCTION
2	ILLEGAL PSEUDO INSTRUCTION
3	DOUBLE DEFINED VARIABLE
4	DOUBLE DEFINED LABEL

Table H-1 ASM-86 Diagnostic Error Messages (cont'd)

NUMBER	MEANING
5	UNDEFINED INSTRUCTION
6	GARBAGE AT END OF LINE — IGNORED
7	OPERAND (S) MISMATCH INSTRUCTION
8	ILLEGAL INSTRUCTION OPERANDS
9	MISSING INSTRUCTION
10	UNDEFINED ELEMENT OF EXPRESSION
11	ILLEGAL PSEUDO OPERAND
12	NESTED "IF" ILLEGAL — "IF" IGNORED
13	ILLEGAL "IF" OPERAND — "IF" IGNORED
14	NO MATCHING "IF" FOR "ENDIF"
15	SYMBOL ILLEGALLY FORWARD REFERENCED — NEGLECTED
16	DOUBLE DEFINED SYMBOL — TREATED AS UNDEFINED
17	INSTRUCTION NOT IN CODE SEGMENT
18	FILE NAME SYNTAX ERROR
19	NESTED INCLUDE NOT ALLOWED
20	ILLEGAL EXPRESSION ELEMENT
21	MISSING TYPE INFORMATION IN OPERAND (S)
22	LABEL OUT OF RANGE
23	MISSING SEGMENT INFORMATION IN OPERAND
24	ERROR IN CODEMACRO BUILDING



Appendix I

DDT-86 Error Messages

Table I-1 DDT-86 Error Messages

ERROR MESSAGE	MEANING
AMBIGUOUS OPERAND	An attempt was made to assemble a command with an ambiguous operand. Precede the operand with the prefix "BYTE" or "WORD".
CANNOT CLOSE	The disk file written by a W command cannot be closed.
DISK READ ERROR	The disk file specified in an R command could not be read properly.
DISK WRITE ERROR	A disk write operation could not be successfully performed during a W command, probably due to a full disk.
INSUFFICIENT MEMORY	There is not enough memory to load the file specified in an R or E command.
MEMORY REQUEST DENIED	A request for memory during an R command could not be fulfilled. Up to eight blocks of memory may be allocated at a given time.

Table I-1 DDT-86 Error Messages (cont'd)

ERROR MESSAGE	MEANING
NO FILE	The file specified in an R or E command could not be found on the disk.
NO SPACE	There is no space in the directory for the file being written by a W command.
VERIFY ERROR AT s:o	The value placed in memory by a Fill, Set, Move, or Assemble command could not be read back correctly, indicating bad RAM or attempting to write to ROM or nonexistent memory at the indicated location.

Index

D

K

L

M

N

O

P

Q

R

S

Advanced Personal Computer
TM

NEC
NEC Information Systems, Inc.

USER'S COMMENTS FORM

Document: CP/M-86 User/Programmer's Guide

Document No.: 819-000100-4001 Rev. 01

Please suggest improvements to this manual.

Please list any errors in this manual. Specify by page.

From:

Name _____

Title _____

Company _____

Address _____

Dealer Name _____

Date: _____

Please cut along this line.

Seal or tape all edges for mailing-do not use staples.

FOLD HERE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 386 LEXINGTON, MA

POSTAGE WILL BE PAID BY ADDRESSEE

NEC Information Systems, Inc.
Dept: Publications –APC
5 Militia Drive
Lexington, MA 02173

FOLD HERE

Seal or tape all edges for mailing-do not use staples.