DIGITAL RESEARCH™

Concurrent CP/M-86™

Operating System

Programmer's Utilities Guide

COPYRIGHT

DISCLAIMER

TRADEMARKS

The *Concurrent CP/M-86 Programmer's Utilities Guide* was prepared using the Digital Research TEX Text Formatter and printed in the United States of America.

First Edition: March 1983

Foreword

The *Concurrent CP/M-86™ Programmer's Utilities Guide* documents the 8088 and 8086 assembly language instruction set, rules for use of the Digital Research ASM-86™ assembler, and rules for use of the Digital Research dynamic debugging tool, DDT-86™.

Section 1 contains an introduction to the Digital Research assembler, ASM-86, and the various options that can be used with it. Through one of these options, ASM-86 can generate 8086 machine code in either Intel® or Digital Research format. Appendix A describes these formats.

Section 2 discusses the elements of ASM-86 assembly language. It defines the ASM-86 character set, constants, variables, identifiers, operators, expressions, and statements.

Section 3 describes the ASM-86 housekeeping functions, such as conditional assembly, multiple source file inclusion, and control of the listing printout format.

Section 4 summarizes the 8086 instruction mnemonics accepted by ASM-86. These mnemonics are the same as those used by the Intel assembler, except for four instructions: the intrasegment short jump, intersegment jump, return, and call instructions. Appendix B summarizes these differences.

Section 5 discusses the Code-macro facilities of ASM-86, including Code-macro definition, specifiers, and modifiers, and nine special Code-macro directives. This information is also summarized in Appendix G.

Section 6 discusses DDT-86, the Dynamic Debugging Tool that allows the user to test and debug programs in the 8086 environment. The section includes a sample debugging section.

Concurrent CP/M-86 is supported and documented through four manuals:

- The *Concurrent CP/M-86 User's Guide* documents the user's interface to Concurrent CP/M-86, explaining the various features used to execute applications programs and Digital Research utility programs.
- The *Concurrent CP/M-86 Programmer's Reference Guide* documents the applications programmer's interface to Concurrent CP/M-86, explaining the internal file structure and system entry points, information essential to create applications programs that run in the Concurrent CP/M-86 environment.
- The *Concurrent CP/M-86 Programmer's Utilities Guide* documents the Digital Research utility programs programmers use to write, debug, and verify applications programs written for the Concurrent CP/M-86 environment.
- The *Concurrent CP/M-86 System Guide* documents the internal, hardware-dependent structures of Concurrent CP/M-86.

Table of Contents

Table of Contents (continued)

Table of Contents (continued)

Table of Contents (continued)

Appendixes

Table of Contents (continued)

Tables

Table of Contents (continued)

Tables

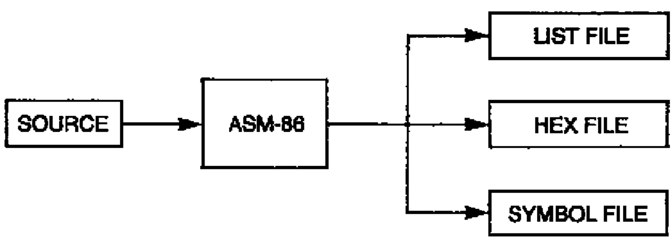
Figure

Listing

Section 1
Introduction to ASM-86

1.1 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 can be in either Intel or Digital Research hex formats, which are described in Appendix C. ASM-86 is shipped in two forms: an 8086 cross-assembler designed to run under CP/M® on the Intel 8080 or the Zilog Z80® based system, and an 8086 assembler designed to run under Concurrent CP/M-86 on an Intel 8086 or 8088 based system. ASM-86 typically produces three output files from one input 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 filetypes. ASM-86 accepts a source file with any three-letter extension, but if the filetype is omitted from the starting command, ASM-86 looks for the specified filename with the filetype .A86 in the directory. If the file has a filetype other than .A86 or has no filetype at all, ASM-86 returns an error message.

The other filetypes 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.

Start ASM-86 by entering a command of the following form:

ASM86 source filespec [$ parameters]

Section 1.2 explains the optional parameters. Specify the source file using the following form:

[d:] filename [.type]

where

[d:]	is an optional 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.
[.type]	is an optional valid filetype of 1 to 3 characters (usually .A86).

Some examples of valid ASM-86 commands are

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

Note that if you try to assemble an empty source file, ASM-86 generates empty list, hex, and symbol files.

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 filetype as described above, the assembler displays the message:

```
NO FILE
```

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

```
PARAMETER ERROR
```

After opening the source, the assembler creates the output files. Usually these are placed on the current disk drive, but they can be redirected by optional parameters or by a drive specification in the source filename. In the latter case, ASM-86 directs the output files to the drive specified in the source filename.

During assembly, ASM-86 halts if an error condition, such as disk full 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

1.2 Optional Run-time Parameters

The dollar-sign character, $, flags an optional string of run-time parameters. A parameter is a single letter followed by a single-letter device name specification. Table 1-1 lists the parameters.

Table 1-1. Run-time Parameter Summary

Parameter	*To Specify*	*Valid Arguments*
A	source file device	A, B, C, ... P
H	hex output file device	A ... P, X, Y, Z
P	list file device	A ... P, X, Y, Z
S	symbol file device	A ... P, 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 can 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. The devices are labeled

A, B, C, ... P or X, Y, Z

Device names A through P, respectively, specify disk drives A through P. 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 can be temporarily stopped at any time by entering a CTRL-S. Restart the output by entering a second CTRL-S or 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 details these formats. If the F parameter is not entered in the command line, ASM-86 produces Digital Research hex format.

Table 1-2. Run-time Parameter Examples

Command Line	*Result*
`ASM86 IO`	Assemble file IO.A86, and produce IO.H86, IO.LST, and IO.SYM, all on the default drive.
`ASM86 IO.ASM $ AD SZ`	Assemble file IO.ASM on device D, and produce IO.LST and IO.H86. No symbol file.
`ASM86 IO $ PY SX`	Assemble file IO.A86, produce IO.H86, route listing directly to printer, and output symbols on console.
`ASM86 IO $ FD`	Produce Digital Research hex format.
`ASM86 IO $ FI`	Produce Intel hex format.

1.3 Ending ASM-86

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

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

A Y response stops the assembly and returns to the operating system. An N response continues the assembly.

End of Section 1

Section 2
Elements of ASM-86 Assembly Language

2.1 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

Lower-case letters are treated as upper-case, except within strings. Only alphanumerics, special characters, and spaces can appear in a string.

2.2 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 can 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.

2.3 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, using separators after delimiters makes your program easier to read.

The following table, Table 2-1, describes ASM-86 separators and delimiters. Some delimiters are also operators and are explained in greater detail in Section 2.6.

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	terminate source lines
LF	line-feed	legal after CR if within source lines, 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
!	exclamation point	logically terminates a statement, allowing multiple statements on a single source line
'	apostrophe	delimits string constants

2.4 Constants

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

2.4.1 Numeric Constants

A numeric constant is a 16-bit value in one of several bases. The base, called the radix of the constant, is denoted by a trailing radix indicator. 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 can be upper- or lower-case.

A constant is thus a sequence of digits followed by an optional radix indicator, where the digits are in the range 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 and 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 a decimal digit, so that ASM-86 cannot confuse a hex constant with an identifier. The following are valid numeric constants:

```
1234   1234D 1100B 1111000011110000B
1234H  0FFEH 3377O 13772Q
3377o  0FE3H 1234d 0ffffh
```

2.4.2 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 a 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 in character strings, so it accepts both upper- and lower-case letters. Note that only alphanumerics, special characters, and spaces are allowed in strings.

A DB assembler directive is the only ASM-86 statement that can contain strings longer than two characters. The string cannot exceed 255 bytes. Include any apostrophe you want printed in the string by entering 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

String in Source Text	*After Processing by ASM-86*
`'a'`	`a`
`'Ab''Cd'`	`Ab'Cd`
`''''`	`'`
`'ONLY UPPER CASE'`	`ONLY UPPERCASE`
`'only lower case'`	`only lower case`

2.5 Identifiers

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

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

Identifiers are of two types. The first type are keywords that the assembler recognizes as having predefined meanings. The second type are symbols defined by the user. The following are all valid identifiers:

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

2.5.1 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 Section 4. Directives are discussed in Section 3. Section 2.6 defines operators. 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 keyword's type attribute is equal to the keyword's numeric value.

Table 2-4. Register Keywords

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

2.5.2 Symbols and Their Attributes

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

- variables
- labels
- numbers

Variables

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 between the beginning of the segment and the location of this variable.
- Type tells how many bytes of data are manipulated when this variable is referenced.

A segment can 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 Section 3.2. A segment can 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 can be any number between 00H and 0FFFFH (65535 decimal). 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 Section 3.2.2. For example, a variable is defined when it appears as the name for a storage directive:

```
VARIABLE DB 0
```

A variable can also be defined as the name for an EQU directive referencing another label, as shown below:

```
VARIABLE EQU ANOTHER_VARIABLE
```

Labels

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

Label segment and offset attributes are essentially the same as variable segment and offset attributes. In general, a label is defined when it precedes an instruction. A colon, :, separates the label from the instruction. For example,

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

A label can also appear as the name for an EQU directive referencing another label. For example,

```
LABEL EQU ANOTHER_LABEL
```

Numbers

Numbers can also be defined as symbols. A number symbol is treated as though you had explicitly coded the number it represents. For example,

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

equals

```
MOV    AL,5
```

Section 2.6 describes operators and their effects on numbers and number symbols.

2.6 Operators

ASM-86 operators fall into the following categories: arithmetic, logical, and relational operators, segment override, variable manipulators, and creators. The following table 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, using the OR bar character | to separate 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
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 0s become 1s, 1s become 0s	a = 16-bit number
	Relational Operators	
a EQ b	returns 0FFFFH if a = b, otherwise 0	a, b = unsigned number
a LT b	returns 0FFFFH if a < b, otherwise 0	a, b = unsigned number
a LE b	returns 0FFFFH if a <= b, otherwise 0	a, b = unsigned number
a GT b	returns 0FFFFH if a > b, otherwise 0	a, b = unsigned number
a GE b	returns 0FFFFH if a >= b otherwise 0	a, b = unsigned number
a NE b	returns 0FFFFH if a <> b, otherwise 0	a, b = unsigned number

Table 2-5. (continued)

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	does unsigned multiplication of a and b	a, b = number
a / b	does unsigned division of a and b	a, b = number
a MOD b	returns remainder of a / b	a, b = number
a SHL b	returns the value which results from shifting a to left by an amount b	a, b = number
a SHR b	returns the value which results from shifting a to the right by an amount b	a, b = number
+ a	gives a	a = number
– a	gives 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. (continued)

Syntax	*Result*	*Validity*
Variable Manipulators, Creators		
SEG a	creates a number whose value is the segment value of the variable or label a. The variable or label must be declared in an absolute segment (i.e. CSEG 1234H); otherwise the SEG operator is undefined.	a = label | variable
OFFSET a	creates a number whose value is the offset value of the variable or label a.	a = label | variable
TYPE a	creates a number. If the variable a is of type BYTE, WORD or DWORD, the value of the number is 1, 2, or 4, respectively.	a = label | variable
LENGTH a	creates 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	creates virtual variable or label with type of a and attributes of b.	a = BYTE | WORD, | DWORD b = <addr exp>
.a	creates variable with an offset attribute of a; segment attribute is current segment.	a = number
$	creates label with offset equal to current value of location counter; segment attribute is current segment.	no argument

2.6.1 Operator Examples

Logical operators accept only numbers as operands. They perform the Boolean logic operations AND, OR, XOR, and NOT. 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 can 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, the offset of which 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, the offset of which 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. * and / treat all operands as unsigned numbers. For example,

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

Unary operators accept both signed and unsigned operators, as shown in the following example:

```
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 can override the assembler's choice by specifying a different register with the segment override operator. The syntax for the override operator is

<segment register> : <address expression>

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

```
0024 368B472D      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,

```
  1234                                     DSEG      1234H
002D 000000000000   WORDBUFFER             DW        0,0,0
0033 0102030405     BUFFER                 DB        1,2,3,4,5
                                           .
                                           .
                                           .
0038 B80500                       MOV      AX,LENGTH BUFFER
003B B80400                       MOV      AX,LAST BUFFER
003E B80100                       MOV      AX,TYPE BUFFER
0041 B80200                       MOV      AX,TYPE WORDBUFFER
0044 B83412                       MOV      AX,SEG BUFFER
```

The PTR operator creates a virtual variable or label that is valid only during the execution of the instruction. It 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 in the following example:

```
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. The operand of the new variable must be a number. For example,

```
004B 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 segment. This operator takes no operand. For example,

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

2.6.2 Operator Precedence

Expressions combine variables, labels, or numbers with operators. ASM-86 allows several kinds of expressions. See Section 2.7. This section defines the order in which operations are executed if more than one operator appears in an expression.

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 rules of precedence. The part of an expression enclosed in parentheses is evaluated first. If parentheses are nested, the innermost expressions are evaluated first. 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>:
9	Variable manipulators, creators	SEG, OFFSET, PTR, TYPE, LENGTH, LAST
10	Parentheses/brackets	( ), [ ]
11	Period and Dollar	., $

2.7 Expressions

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

- segment value
- offset value
- type

Both variables and labels are address expressions. An address expression is not a number, but its components are numbers. Numbers can 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 can 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   AX,[BX+DI]
MOV   AX,[SI]
```

2.8 Statements

Just as tokens in this assembly language correspond to words in English, statements are analogous to sentences. A statement tells ASM-86 what action to perform. Statements can be instructions or 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, !. ASM-86 treats these as an end-of-line. Multiple assembly language statements can be written on the same physical line if separated by exclamation points.

The ASM-86 instruction set is defined in Section 4. The syntax for an instruction statement is

[label:] [prefix] mnemonic [operand(s)] [;comment]

where the fields are defined as

- 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 can 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 can be present, although the other fields can appear. ASM-86 mnemonics are defined in Section 4.
- operands — An instruction mnemonic can require other symbols to represent operands to the instruction. Instructions can have zero, one, or two operands.
- comment — Any semicolon appearing outside a character string begins a comment. A comment ends with a carriage return. Comments improve the readability of programs. This field is optional.

ASM-86 directives are described in Section 3. The syntax for a directive statement is

[name] directive operand(s) [;comment]

where the fields are defined as

- name — Unlike the label field of an instruction, the name field of a directive is never terminated with a colon. Directive names are legal only for DB, DW, DD, RB, RS, RW, 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 Section 3.
- operands — Analogous to the operands for instruction mnemonics. Some directives, such as DB, DW, and DD, allow any operand; others have special requirements.
- comment — Exactly as defined for instruction statements.

End of Section 2

Section 3
Assembler Directives

3.1 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 Section 2.8.

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. Square brackets, [], enclose optional arguments.

3.2 Segment Start Directives

At run-time, 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 Segments, which are addressed respectively 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 can 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.

3.2.1 The CSEG Directive

Syntax:

```
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 in 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.

3.2.2 The DSEG Directive

Syntax:

```
DSEG    numeric expression
DSEG
DSEG    $
```

This directive specifies that the following statements belong to the Data Segment. The Data Segment 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.

3.2.3 The SSEG Directive

Syntax:

```
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.

3.2.4 The ESEG Directive

Syntax:

```
ESEG    numeric expression
ESEG
ESEG    $
```

This directive initiates the Extra Segment. Instruction statements are not legal in this segment, but all directive statements are legal.

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.

3.3 The ORG Directive

Syntax:

```
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 because forward references can be ambiguous.

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.

3.4 The IF and ENDIF Directives

Syntax:

```
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 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, the lines are not assembled, but are listed unless a NOIFLIST directive is active. All elements in the numeric expression must be defined before they appear in the IF directive. IF directives can be nested to a maximum depth of five levels.

3.5 The INCLUDE Directive

Syntax:

INCLUDE filespec

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

```
INCLUDE   EQUALS.A86
```

Use INCLUDE when the source program resides in several different files. INCLUDE directives cannot be nested; a source file called by an INCLUDE directive cannot contain another INCLUDE statement. If filespec does not contain a filetype, the filetype is assumed to be .A86. If the file specification does not include a drive specification, ASM-86 assumes that the file resides on the drive containing the source file.

3.6 The END Directive

Syntax:

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).

3.7 The EQU Directive

Syntax:

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 cannot terminate 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 (sub)set. 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
```

3.8 The DB Directive

Syntax:

[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 Section 2.4.2. A DB directive is the only ASM-86 statement that accepts a string constant longer than two bytes. There is no translation from lower- to upper-case within strings. Multiple expressions or constants, separated by commas, can be added to the definition, but cannot 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 the length attribute tells the number of bytes (allocation units) reserved.

The following statements show DB directives with symbols:

```
005F 43502F4D2073   TEXT     DB      'CP/M system',0
     79737465D00
008B E1             AA       DB      'a' + 80H
008C 0102030405     X        DB      1,2,3,4,5
                                     .
                                     .
                                     .
0071 B80C00                  MOV     CX,LENGTH TEXT
```

3.9 The DW Directive

Syntax:

[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 as DB to initialize storage. 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
```

3.10 The DD Directive

Syntax:

[symbol] DD numeric expression[,address 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 is stored in the two upper bytes. Otherwise, DD follows the same procedure as DB. For example,

```
1234                           CSEG  1234H
                                     .
                                     .
                                     .
0000 8CC134128FC1 LONG_JMPTAB  DD        ROUT1,ROUT2
     3412
0008 72C1341275C1              DD        ROUT3,ROUT4
     3412
```

3.11 The RS Directive

Syntax:

[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. For example,

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

If an RS statement is the last statement in a segment, you must follow it with a DB statement in order for GENCMD to allocate the memory space.

3.12 The RB Directive

Syntax:

[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 gives the byte attribute.

3.13 The RW Directive

Syntax:

[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. For example,

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

3.14 The TITLE Directive

Syntax:

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. For example,

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

If the title is too long, the ASM-86 page number overwrites the title.

3.15 The PAGESIZE Directive

Syntax:

PAGESIZE numeric expression

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

3.16 The PAGEWIDTH Directive

Syntax:

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, unless the listing is routed directly to the terminal, when the default page width is 78.

3.17 The EJECT Directive

Syntax:

EJECT

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

3.18 The SIMFORM Directive

Syntax:

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 out on a printer unable to interpret the form-feed character.

3.19 The NOLIST and LIST Directives

Syntax:

NOLIST
LIST

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

3.20 The IFLIST and NOIFLIST Directives

Syntax:

IFLIST
NOIFLIST

The NOIFLIST directive suppresses the printout of the contents of IF-ENDIF blocks that are not assembled. The IFLIST directive resumes printout of IF-ENDIF blocks.

End of Section 3

Section 4
The ASM-86 Instruction Set

4.1 Introduction

The ASM-86 instruction set includes all 8086 machine instructions. This section briefly describes ASM-86 instructions; these descriptions are organized into functional groups. The general syntax for instruction statements is given in Section 2.8.

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; its operands are its 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. In 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
reg16	a 16-bit general purpose register, not segment register
segreg	any segment register: CS, DS, SS, or ES

Table 4-1. (continued)

Symbol	*Operand Type*
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 that evaluates to a label
lab8	any label that is within ± 128 bytes distance from the instruction

The 8086 CPU has nine single-bit Flag registers that reflect the state of the CPU. The user cannot access these registers directly, but the user can test them to determine the effects of an executed instruction upon 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

Symbol	*Meaning*
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

4.2 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 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.
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.

Table 4-3. (continued)

<table>
<tr><th colspan="2">Syntax</th><th>Result</th></tr>
<tr><td>MOV</td><td>reg,mem|reg</td><td>Move memory or register to register.</td></tr>
<tr><td>MOV</td><td>mem|reg,reg</td><td>Move register to memory or register.</td></tr>
<tr><td>MOV</td><td>mem|reg,numb</td><td>Move immediate data to memory or register.</td></tr>
<tr><td>MOV</td><td>segreg,mem|reg16</td><td>Move memory or register to segment register.</td></tr>
<tr><td>MOV</td><td>mem|reg16,segreg</td><td>Move segment register to memory or register.</td></tr>
<tr><td>OUT</td><td>numb8|numb16,acc</td><td>Transfer data from accumulator to output port (0-255) given by numb8 or numb16.</td></tr>
<tr><td>OUT</td><td>DX,acc</td><td>Transfer data from accumulator to output port (0-0FFFFH) given by DX register.</td></tr>
<tr><td>POP</td><td>mem|reg16</td><td>Move top stack element to memory or register.</td></tr>
<tr><td>POP</td><td>segreg</td><td>Move top stack element to segment register. Note that CS segment register is not allowed.</td></tr>
<tr><td>POPF</td><td></td><td>Transfer top stack element to flags.</td></tr>
<tr><td>PUSH</td><td>mem|reg16</td><td>Move memory or register to top stack element.</td></tr>
<tr><td>PUSH</td><td>segreg</td><td>Move segment register to top stack element.</td></tr>
<tr><td>PUSHF</td><td></td><td>Transfer flags to top stack element.</td></tr>
<tr><td>SAHF</td><td></td><td>Transfer the AH register to flags.</td></tr>
<tr><td>XCHG</td><td>reg,mem|reg</td><td>Exchange register and memory or register.</td></tr>
<tr><td>XCHG</td><td>mem|reg,reg</td><td>Exchange memory or register and register.</td></tr>
<tr><td>XLAT</td><td>mem|reg</td><td>Perform table lookup translation, table given by mem|reg, which is always BX. Replaces AL with AL offset from BX.</td></tr>
</table>

4.3 Arithmetic, Logical, and Shift Instructions

The 8086 CPU performs the four basic mathematical operations in several different ways. It supports both 8- and 16-bit operations and also 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 Bit	*Result*
CF	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.
AF	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	set if the result of the operation is zero. Otherwise, ZF is cleared.
SF	set if the result is negative.
PF	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	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) 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.
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		Decimal adjust for addition; adjusts AL.
DAS		Decimal adjust for subtraction; adjusts AL.

Table 4-5. (continued)

Syntax		Result
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.
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. Logical and Shift Instructions

Syntax		Result
AND	reg,mem|reg	Perform bitwise logical AND of a register and memory or register.
AND	mem|reg,reg	Perform bitwise logical AND of memory or register and register.
AND	mem|reg,numb	Perform bitwise logical AND of memory or register and data constant.
NOT	mem|reg	Form one's complement of memory or register.
OR	reg,mem|reg	Perform bitwise logical OR of a register and memory or register.
OR	mem|reg,reg	Perform bitwise logical OR of memory or 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. (continued)

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 of memory register and data constant; set condition flags, but do not change destination.
XOR	reg,mem|reg	Perform bitwise logical exclusive OR of a register and memory or register.

Table 4-6. (continued)

Syntax		Result
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.

4.4 String Instructions

String instructions take zero, 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	Subtract source from destination; affect flags, but do not return result.
CMPSB		An alternate mnemonic for CMPS, which assumes a byte operand.
CMPSW		An alternate mnemonic for CMPS, which assumes a word operand.
LODS	mem|reg	Transfer a byte or word from the source operand to the accumulator.
LODSB		An alternate mnemonic for LODS, which assumes a byte operand.
LODSW		An alternate mnemonic for LODS, which assumes a word operand.

Table 4-7. (continued)

<table>
<tr><th colspan="2">Syntax</th><th>Result</th></tr>
<tr><td>MOVS</td><td>mem|reg,mem|reg</td><td>Move 1 byte (or word) from source to destination.</td></tr>
<tr><td>MOVSB</td><td></td><td>An alternate mnemonic for MOVS, which assumes a byte operand.</td></tr>
<tr><td>MOVSW</td><td></td><td>An alternate mnemonic for MOVS, which assumes a word operand.</td></tr>
<tr><td>SCAS</td><td>mem|reg</td><td>Subtract destination operand from accumulator (AX or AL); affect flags, but do not return result.</td></tr>
<tr><td>SCASB</td><td></td><td>An alternate mnemonic for SCAS, which assumes a byte operand.</td></tr>
<tr><td>SCASW</td><td></td><td>An alternate mnemonic for SCAS, which assumes a word operand.</td></tr>
<tr><td>STOS</td><td>mem|reg</td><td>Transfer a byte or word from accumulator to the destination operand.</td></tr>
<tr><td>STOSB</td><td></td><td>An alternate mnemonic for STOS which assumes a byte operand.</td></tr>
<tr><td>STOSW</td><td></td><td>An alternate mnemonic for STOS which assumes a word operand.</td></tr>
</table>

Table 4-8 defines prefixes for string instructions. A prefix repeats its 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.

Table 4-8. Prefix Instructions

Syntax	*Result*
REP	Repeat until CX register is zero.
REPE	Equal to REPZ.
REPNE	Equal to REPNZ.
REPNZ	Repeat until CX register is zero and zero flag (ZF) is zero.
REPZ	Repeat until CX register is zero and zero flag (ZF) is not zero.

4.5 Control Transfer Instructions

There are four classes of control transfer instructions:

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

All control transfer instructions cause program execution to continue at some new location in memory, possibly in a new code segment. The transfer can be absolute or it can depend upon a certain condition. Table 4-9 defines control transfer instructions. In the definitions of conditional jumps, above and below refer to the relationship between unsigned values. 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.
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), and jump to the target label.
CALLF	mem	Push CS register on the stack, push the offset address of the next instruction on the stack, and 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, and transfer control with an indirect call through any one of the 256 interrupt-vector elements. 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, and 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 and restore saved flag registers, as well as CS and IP. 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).
JBE	lab8	Jump if below or equal or not above ((CF or ZF) = 1).

Table 4-9. (continued)

Syntax		Result
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.
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.

Table 4-9. (continued)

Syntax		Result
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.
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.

Table 4-9. (continued)

Syntax		Result
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.

4.6 Processor Control Instructions

Processor control instructions manipulate the flag registers. Moreover, some of these instructions 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 pointers.
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 coprocessor). numb8 must be in the range 0, 63.
HLT		8086 processor enters halt state until an interrupt is recognized.

Table 4-10. (continued)

Syntax	*Result*
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 can precede any other instruction. Buslock prevents coprocessors from gaining the bus; this is useful for shared-resource semaphores.
NOP	No operation is performed.
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.

4.7 Mnemonic Differences

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 4-11. Mnemonic Differences

Mnemonic Function	*CP/M*	*Intel*
Intrasegment short jump:	JMPS	JMP
Intersegment jump:	JMPF	JMP
Intersegment return:	RETF	RET
Intersegment call:	CALLF	CALL

End of Section 4

Section 5
Code-macro Facilities

5.1 Introduction to Code-macros

A macro simplifies using the same block of instructions over and over again throughout a program. ASM-86 does not support traditional assembly-language macros, but it does allow you to define your own instructions by using the Code-macro directive. An ASM-86 Code-macro sends a bit stream to the output file, adding a new instruction to the assembler.

Like traditional macros, Code-macros are assembled wherever they appear in assembly language code, but there the similarity ends. Traditional macros contain assembly language instructions, but a Code-macro contains only Code-macro directives. Macros are usually defined in the user's symbol table; ASM-86 Code-macros are defined in the assembler's symbol table.

Because ASM-86 treats a Code-macro as an instruction, you can start Code-macros by using them as instructions in your program. The example below shows how to start MAC™, an instruction defined by a Code-macro.

```
        .
        .
        .
XCHG    BX,WORD3
MAC     PAR1,PAR2
MUL     AX,WORD4
        .
        .
        .
```

Note that MAC accepts two operands. When MAC was defined, these two operands were also classified by type, size, and so on by defining MAC's formal parameters. The names of formal parameters are not fixed. They are stand-ins that are replaced by the names or values supplied as operands when the Code-macro starts. Thus, formal parameters hold the place and indicate where and how to use the operands.

The definition of a Code-macro starts with a line specifying its name and any formal parameters:

CODEMACRO name [formal parameter list]

where the optional formal parameter list is defined:

formal name : specifier letter [modifier letter][range]

The formal name is not fixed, but represent a place holder. If formal parameter list is present, the specifier letter is required and the modifier letter is optional. Possible specifiers are A, C, D, E, M, R, S, and X. Possible modifier letters are b, d, w, and sb. The assembler ignores case except within strings, but this section shows specifiers in upper-case and modifiers in lower-case. Following sections describe specifiers, modifiers, and the optional range in detail.

The body of the Code-macro describes the bit pattern and formal parameters. Only the following directives are legal within Code-macros:

SEGFIX
NOSEGFIX
MODRM
RELB
RELW
DB
DW
DD
DBIT

These directives are unique to Code-macros. Those that appear to duplicate ASM-86 directives (DB, DW, and DD) have different meanings in Code-macro context. These directives are detailed in later sections. The definition of a Code-macro ends with a line:

```
EndM
```

CodeMacro, EndM, and the Code-macro directives are all reserved words. Code-macro definition syntax is defined in Backus-Naur-like form in Appendix G. The following examples are typical Code-macro 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
```

5.2 Specifiers

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

Table 5-1. Code-macro 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).
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.

5.3 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 modifier requires the operand to be of type b for byte, w for word, d for double-word, and sb for signed byte. For numbers, the modifiers require the number to be of a certain size: b for –256 to 255 and w for other numbers. Table 5-2 summarizes Code-macro modifiers.

Table 5-2. Code-macro Operand Modifiers

Variables		*Numbers*	
Modifier	*Type*	*Modifier*	*Size*
b	byte	b	–256 to 255
w	word	w	anything else
d	dword		
sb	signed byte		

5.4 Range Specifiers

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

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

Numberb is 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 last example specifies that the opcode is to be immediate data and ranges from 0 to 63, inclusive:

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

5.5 Code-macro Directives

Code-macro directives define the bit pattern and make further requirements on how the operand is to be treated. Directives are reserved words. Those that appear to duplicate assembly language instructions have different meanings in a Code-macro definition. Only the nine directives defined here are legal in Code-macro definitions.

5.5.1 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, it 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 that represents the memory address. Because it represents a memory address, the formal parameter must have one of the specifiers E, M, or X.

5.5.2 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, and STOS. The form of NOSEGFIX is

NOSEGFIX segreg,formal name

where segreg is one of the segment registers ES, CS, SS, or DS and formal 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 NOSEGFIX use:

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

5.5.3 MODRM

This directive instructs the assembler to generate the MODRM byte that follows the opcode byte in many 8086 instructions. The MODRM byte contains either the indexing type or the register number to be used in the instruction. It also specifies the register 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 on 8086 instructions and their bit patterns, see the Intel *8086 Assembly Language Programming Manual* and the Intel *8086 Family User's Manual*.

The forms of MODRM are:

MODRM formal name, formal name
MODRM NUMBER7, formal name

where NUMBER7 is a value 0 to 7 inclusive, and formal name is the name of a formal parameter. The following examples show how to use 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
```

5.5.4 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 supplied as an operand. RELB generates one byte and RELW two bytes of displacement. The directives take the following forms:

> RELB formal name
> RELW formal name

where formal name is the name of a formal parameter with a C (code) specifier. For example,

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

5.5.5 DB, DW, and DD

These directives differ from those that occur outside of Code-macros. The forms of the directives are

DB formal name | NUMBERB
DW formal name | NUMBERW
DD formal name

where NUMBERB is a single-byte number, NUMBERW is a two-byte number, and formal 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
```

5.5.6 DBIT

This directive manipulates bits in combinations of a byte or less. The form is

DBIT field description[,field description]

where a field description has two forms:

number combination
number (formal name(rshift))

number ranges from 1 to 16 and specifies the number of bits to be set. Combination 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 formal name, a formal parameter name instructing the assembler to put a certain number in the specified position. This number usually refers to the register specified in the first line of the Code-macro. The numbers used in this special case for each register are

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

A rshift, 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 following definition 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 instruction

```
DEC      DX
```

is assembled and DX has a value of 2H, then 48H + 2H = 4AH, 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.

End of Section 5

Section 6
DDT-86

6.1 DDT-86 Operation

The DDT-86 program allows you to test and debug programs interactively in a Concurrent CP/M-86 environment. You should be familiar with the 8086 processor, ASM-86, and the Concurrent CP/M-86 operating system before using DDT-86.

6.1.1 Starting DDT-86

Start DDT-86 by entering a command in one of the following forms:

DDT86
DDT86 filename

The first command simply loads and executes DDT-86. After displaying its sign-on message and the 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 it loads the file specified by filename. If the filetype is omitted from the filename, .CMD is assumed. Note that DDT-86 *cannot* load a file of type .H86. The second form of the starting command is equivalent to the sequence:

```
A>DDT86
DDT86 x.x
-E filename
```

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

6.1.2 DDT-86 Command Conventions

When DDT-86 is ready to accept a command, it prompts the operator with a hyphen (-). In response, you can type a command line, or a CTRL-C to end the debugging session. See Section 6.1.4. A command line can have up to 64 characters and must terminate with a carriage return. While entering the command, use standard CP/M line-editing functions, such as CTRL-X, CTRL-H, and CTRL-R, to correct typing errors. DDT-86 does not process the command line until you enter a carriage return.

The first character of each command line determines the command action. Table 6-1 summarizes DDT-86 commands. DDT-86 commands are defined individually in Section 6.2.

Table 6-1. DDT-86 Command Summary

Command	*Action*
A	Enter assembly language statements.
B	Compare blocks of memory.
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	Hexadecimal arithmetic.
I	Set up File Control Block and command tail.
L	List memory using 8086 mnemonics.
M	Move memory block.
QI	Read I/O port.
QO	Write I/O port.
R	Read disk file into memory.
S	Set memory to new values.
SR	Search for string.
T	Trace program execution.
U	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. These can be hexadecimal values, filenames, 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.

6.1.3 Specifying a 20-Bit Address

Most DDT-86 commands require one or more addresses as operands. Because the 8086 can address up to 1 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 can be a 16-bit hexadecimal value or the name of a segment register. If a segment register name is specified, the value of ssss is the contents of that register in the user's CPU state, as indicated by the X command. If omitted, the value of ssss is a default value appropriate to the command being executed, as described in Section 6.3.

6.1.4 Terminating DDT-86

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

6.1.5 DDT-86 Operation with Interrupts

DDT-86 operates with interrupts enabled or disabled and preserves the interrupt state of the program being executed under DDT-86. When DDT-86 has control of the CPU, either when it starts, 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 started, except for a few critical regions where interrupts are disabled. 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.

6.2 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.

6.2.1 The A (Assemble) Command

The A command assembles 8086 mnemonics directly into memory. The form is

As

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 the operator enters assembly language statements as described in Section 2.8. When a statement is entered, DDT-86 converts it to binary, places the values 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.

6.2.2 The B (Block Compare) Command

The B command compares two blocks of memory and displays any differences on the screen. The form is

Bs1,f1,s2

where s1 is the 20-bit address of the start of the first block; f1 is the offset of the final byte of the first block, and s2 is the 20-bit address of the start of the second block. If the segment is not specified in s2, the same value is used that was used for s1.

Any differences in the two blocks are displayed at the screen in the following form:

s1:o1 b1   s2:o2 b2

where s1:o1 and s2:o2 are the addresses in the blocks; b1 and b2 are the values at the indicated addresses. If no differences are displayed, the blocks are identical.

6.2.3 The D (Display) Command

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

```
D
Ds
Ds,f
DW
DWs
DWs,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 in 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.

6.2.4 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 forms:

```
E filename
E
```

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 blocks of memory allocated by previous E or R commands or by programs executed under DDT-86. Thus only one file at a time can be loaded for execution.

When the load is complete, DDT-86 displays the start and end 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. Files are closed after an E command.

E with no filename frees all memory allocations made by DDT-86, without loading a file.

6.2.5 The F (Fill) Command

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

```
Fs,f,b
FWs,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 in 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 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.

6.2.6 The G (Go) Command

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

```
G
G,b1
G,b1,b2
Gs
Gs,b1
Gs,b1,b2
```

where s is a 20-bit address where program execution is to start, and b1 and b2 are 20-bit addresses of breakpoints. 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, so DDT-86 derives the 20-bit address from the user's CS and IP registers. The first form transfers control to your program without setting any breakpoints. The next two forms set one and two breakpoints, respectively, before passing control to your program. The next three forms are analogous to the first three, except that your CS and IP registers are first set to s.

Once control has been transferred to the program under test, it executes in real time 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.

6.2.7 The H (Hexadecimal Math) Command

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

Ha,b

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

ssss dddd

6.2.8 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 I command takes the form:

I command tail

where command tail is a character string which 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 command tail are also copied into the default command buffer at 0080H. The length of command tail is stored at 0080H, followed by the character string ending 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 16-bit value at absolute memory location 0:6. 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.

6.2.9 The L (List) Command

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

L
Ls
Ls,f

where s is a 20-bit address where 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 and 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.

Long displays can be aborted by typing any key during the list process. Or, enter CTRL-S to halt the display temporarily.

6.2.10 The M (Move) Command

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

```
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 same value is used that was used for s. Note that if 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.

6.2.11 The QI, QO (Query I/O) Commands

The QI and QO commands allow access to any of the 65,536 input/output ports. The QI command reads data from a port; the QO command writes data to a port. The forms of the QI command are

```
QIn
QIWn
```

where n is the 16-bit port number. In the first case, DDT-86 displays the 8-bit value read from port n. In the second case, DDT-86 displays a 16-bit value from port n.

The forms of the QO command are

QOn,v
QOWn,v

where n is the 16-bit port number, and v is the value to output. In the first case, the 8-bit value v is written to port n. If v is greater than 255, DDT-86 responds with a question mark. In the second case, the 16-bit value v is written to port n.

6.2.12 The R (Read) Command

The R command reads a file into a contiguous block of memory. The forms are

R filename
R filename,s

where filename is the name and type of the file to be read, and s is the location to which the file is read. The first form lets DDT-86 determine the memory location into which the file is read.

The second form tells DDT-86 to read the file into the memory segment beginning at s. This address can have the standard form (ssss:0000). The low-order four bits of s are assumed to be zero, so DDT-86 reads files on a paragraph boundary. If the memory at s is not available, DDT-86 issues the message:

```
MEMORY REQUEST DENIED
```

DDT-86 reads the file into memory and displays the start and end addresses of the block of memory occupied by the file. 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. Thus a number of files can be read into memory without overlapping.

If the file does not exist or there is not enough memory to load the file, DDT-86 issues an error message. Files are closed after an R command, even if an error occurs.

The following are examples of the R command, followed by a brief explanation.

`rddt86.cmd`	Read file DDT86.CMD into memory.
`rtest`	Read file TEST into memory.
`rtest,1000:0`	Read file TEST into memory, starting at location 1000:0.

6.2.13 The S (Set) Command

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

Ss
SWs

where s is the 20-bit address where 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 is

ssss:oooo bb

In response to the second form, the display is

ssss:oooo wwww

where bb and wwww are the contents of memory in byte and word formats, respectively.

In response to one of the above displays, the operator can choose to alter the memory location or to leave it unchanged. If a valid hexadecimal value is entered, 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.

6.2.14 The SR (Search) Command

The SR (Search) command searches a block of memory for a given pattern of numeric or ASCII values and lists the addresses where the pattern occurs. The form is

SRs,f,pattern

where s is the 20-bit starting address of the block to be searched, f is the offset of the final address of the block, and pattern is a list of one or more hexadecimal values and/or ASCII strings. ASCII strings are enclosed in double quotes and can be any length. For example,

```
SR200,300,"The form",0d,0a
```

For each occurrence of pattern, DDT-86 displays the 20-bit address of the first byte of the pattern, in the form:

ssss:oooo

If no addresses are listed, pattern was not found.

6.2.15 The T (Trace) Command

The T command traces program execution for 1 to 0FFFFH program steps. The forms are

T
Tn
TS
TSn

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. A long trace can be aborted before n steps have been executed by pressing any character at the console.

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 because 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.

6.2.16 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 forms are

```
U
Un
US
USn
```

where n is the number of instructions to execute before returning control to the console. The U command can be aborted before n steps have been executed by pressing any key at the console.

6.2.17 The V (Value) Command

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

```
V
```

If the last file was loaded with the E command, the V command displays the start and end 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 start and end 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.

6.2.18 The W (Write) Command

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

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

6.2.19 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 forms are

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 a hyphen, indicating that the corresponding flag is not set (0), or a 1-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.

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 the operator 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 a carriage return is typed, 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 until the last register is displayed.

The third form allows the operator 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 a carriage return is typed, 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 can be examined or altered with each Xf command. Set or reset flags by entering a value of 1 or 0.

After an X command, the type1 and type2 segment values are set to the contents of the CS and DS registers, respectively.

6.3 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 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 type1 segment value if no segment value is specified in the command.

When started, DDT-86 sets the type1 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 type1 segment value to the value of the CS register.
- When a file is read by an R command, DDT-86 sets the type1 segment value to the base segment where the file was read.
- After an X command, the type1 and type2 segment values are set to the contents of the CS and DS registers, respectively.
- When DDT-86 regains control from a user program after a G, T or U command, it sets the type1 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 type1 segment value to the segment value specified.

The second type of command pertains to the Data Segment: B (Block Compare), D (Display), F (Fill), M (Move), S (Set), and SR (Search). These commands use the internal type2 segment value if no segment value is specified in the command.

When started, DDT-86 sets the type2 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 type2 segment value to the value of the DS register.
- When a file is read by an R command, DDT-86 sets the type2 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 type2 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 type2 segment value to the value of the DS register.
- When a segment value is specified explicitly in a B, D, F, M, S, or SR command, DDT-86 sets the type2 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 because 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	
B		x
D		x
E	c	c
F		x
G	c	c
H		
I		
L	x	
M		x
R	c	c
S		x
SR		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.

6.4 Assembly Language Syntax for A and L Commands

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) can appear in one statement, but they all must precede the opcode of the statement. Alternately, a prefix can 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 form:

  ssss:oooo

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

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

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

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

- Operands that 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 forms of register indirect memory operands are

  [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. For example,

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

6.5 DDT-86 Sample Session

In the following sample session, you interactively debug a simple sort program. Comments explain the steps involved.

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                          PAGE 1



                  ;
                  ;      simple sort program
                  ;
                  sort:
0000 BE0000            mov  si,0          ;initialize index
0003 BB0001            mov  bx,offset;bx = base of list
                            nlist
0006 C6060B0100        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 C6060B0101        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 F6060B0101        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
0008 COMP     0029 DONE     001C INCI     0000 SORT
```

Type hex file generated by ASM-86.

```
A>type sort.h86
:04000003000000000FB
:1B0000B1BE0000BB0001C6060B0100BA003A4001760A8840018800C60808016C
:11001B81014BB3FE0B75E9F80B0801017507E9F0FFEE
:080100820308040B1F0604010035
: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 0481 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 08 1F 0B 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  0028
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  AD   [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

```
#sort
    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 [010B],01
047D:0027   JNZ  0000
047D:002B   JMP  0028
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 [010B],01
047D:001C INC     SI
047D:001D CMP     SI,0003
047D:0020 JNZ     000B
047D:0022 TEST    BYTE [010B],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 [010B],01
047D:001C   INC  SI
047D:001D   CMP  SI,000B
047D:0020   JNZ  000B
047D:0022   TEST BYTE [010B],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 08 1F 08 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, let's 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 to patch.)

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

Display unsorted list.

```
-d100
0480:0100 03 08 04 08 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 20 20 20 .............
```

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

```
-g,2B
*047D:002B
```

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,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-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

-d100,10f
0480:0100 03 04 08 06 1F 08 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
--------- 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:0008 MOV     BYTE [010B],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 [010B],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,28
*047D:0028
```

See if list is sorted.

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

Looks better; let's install patch in disk file. To do this, we must read CMD file including header, so we 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
```

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.

```
047D:0000 MOV   SI,0000
047D:0003 MOV   BX,0100
047D:0008 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
```

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................

-g,29
*047D:0029




-d100,10F
0480:0100 03 08 04 06 1F 06 04 01 00 00 00 00 00 00 00 00 ............
-^C
A>
```

End of Section 6

Appendix A
Starting ASM-86

Command: `A>ASM86`

Syntax:

ASM86 filespec [$ parameters]

where

filespec	is the 8086 assembly source file (drive and filetype are optional).
parameters	is a one-letter type followed by a one-letter device from the table below.

Default filetype:

.A86

Parameters:

$ Td where T = type and d = device

Table A-1. Parameter Types and Devices

TYPES: *DEVICES:*	*A*	*H*	*P*	*S*	*F*
A-P	x	x	x	x	
X		x	x	x	
Y		x	x	x	
Z		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

Type	*Function*
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

Name	*Meaning*
A - P	Drives A - P
X	console device
Y	printer device
Z	byte bucket
I	Intel hex format
D	Digital Research hex format

Table A-4. Invocation Examples

Example	*Result*
ASM86 IO	Assembles file IO.A86 and produces IO.H86 IO.LST and IO.SYM.
ASM86 IO.ASM $ AD SZ	Assembles file IO.ASM on device D and produces IO.LST and IO.H86. No symbol file.
ASM86 IO $ PY SX	Assembles file IO.A86, produces IO.H86, routes listing directly to printer, and outputs symbols on console.
ASM86 IO $ FD	Produces Digital Research hex format.
ASM86 IO $FI	Produces Intel hex format.

End of Appendix A

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*
Intrasegment short jump:	JMPS	JMP
Intersegment jump:	JMPF	JMP
Intersegment return:	RETF	RET
Intersegment call:	CALLF	CALL

End of Appendix B

Appendix C
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 Digital adds segment information to hexadecimal records. Output of either format can be input to the 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	ll
3-6	load address	aaaa
7-8	record type	tt
9-(n-1)	data bytes	dd.....d
n-(n+1)	checksum	cc
n+2	carriage return	CR
n+3	line-feed	LF

Table C-2. Hexadecimal Record Formats

Type	*Content*	*Format*
00	Data record	: ll 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	=>	checksum - 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 type (DT and ST) that Digital Research hexadecimal format differs from Intel. 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 Value*	*Digital 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
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

End of Appendix C

Appendix D
Reserved Words

Table D-1. Keywords or Reserved Words

Predefined Numbers				
BYTE	WORD	DWORD		
Operators				
AND	LAST	MOD	OFFSET	SHR
EQ	LE	NE	OR	TYPE
GE	LENGTH	NOT	SEG	XOR
GT	LT	PTR	SHL	
Assembler Directives				
CODEMACRO	EJECT	IF	NOLIST	RS
CSEG	END	IFLIST	ORG	RW
DB	ENDIF	INCLUDE	PAGESIZE	SIMFORM
DD	ENDM	LIST	PAGEWIDTH	SSEG
DSEG	ESEG	NOIFLIST	RB	TITLE
DW	EQ			
Code-macro Directives				
DB	DD	MODRM	SEGFIX	RELW
DBIT	DW	NOSEGFIX	RELB	
8086 Registers				
AH	BL	CL	DI	ES
AL	BP	CS	DL	SI
AX	BX	CX	DS	SP
BH	CH	DH	DX	SS

Instruction Mnemonics – See Appendix E.

End of Appendix D

Appendix E
ASM-86 Instruction Summary

Table E-1. ASM-86 Instruction Summary

Mnemonic	*Description*	*Section*
AAA	ASCII adjust for Addition	4.3
AAD	ASCII adjust for Division	4.3
AAM	ASCII adjust for Multiplication	4.3
AAS	ASCII adjust for Subtraction	4.3
ADC	Add with Carry	4.3
ADD	Add	4.3
AND	And	4.3
CALL	Call (intrasegment)	4.5
CALLF	Call (intersegment)	4.5
CBW	Convert Byte to Word	4.3
CLC	Clear Carry	4.6
CLD	Clear Direction	4.6
CLI	Clear Interrupt	4.6
CMC	Complement Carry	4.6
CMP	Compare	4.3
CMPS	Compare Byte or Word (of string)	4.4
CMPSB	Compare Byte of string	4.4
CMPSW	Compare Word of string	4.4
CWD	Convert Word to Double Word	4.3
DAA	Decimal Adjust for Addition	4.3
DAS	Decimal Adjust for Subtraction	4.3
DEC	Decrement	4.3
DIV	Divide	4.3
ESC	Escape	4.6
HLT	Halt	4.6
IDIV	Integer Divide	4.3
IMUL	Integer Multiply	4.3
IN	Input Byte or Word	4.2
INC	Increment	4.3
INT	Interrupt	4.5
INTO	Interrupt on Overflow	4.5
IRET	Interrupt Return	4.5

Table E-1. (continued)

Mnemonic	*Description*	*Section*
JA	Jump on Above	4.5
JAE	Jump on Above or Equal	4.5
JB	Jump on Below	4.5
JBE	Jump on Below or Equal	4.5
JC	Jump on Carry	4.5
JCXZ	Jump on CX Zero	4.5
JE	Jump on Equal	4.5
JG	Jump on Greater	4.5
JGE	Jump on Greater or Equal	4.5
JL	Jump on Less	4.5
JLE	Jump on Less or Equal	4.5
JMP	Jump (intrasegment)	4.5
JMPF	Jump (intersegment)	4.5
JMPS	Jump (8-bit displacement)	4.5
JNA	Jump on Not Above	4.5
JNAE	Jump on Not Above or Equal	4.5
JNB	Jump on Not Below	4.5
JNBE	Jump on Not Below or Equal	4.5
JNC	Jump on Not Carry	4.5
JNE	Jump on Not Equal	4.5
JNG	Jump on Not Greater	4.5
JNGE	Jump on Not Greater or Equal	4.5
JNL	Jump on Not Less	4.5
JNLE	Jump on Not Less or Equal	4.5
JNO	Jump on Not Overflow	4.5
JNP	Jump on Not Parity	4.5
JNS	Jump on Not Sign	4.5
JNZ	Jump on Not Zero	4.5
JO	Jump on Overflow	4.5
JP	Jump on Parity	4.5
JPE	Jump on Parity Even	4.5
JPO	Jump on Parity Odd	4.5
JS	Jump on Sign	4.5
JZ	Jump on Zero	4.5
LAHF	Load AH with Flags	4.2
LDS	Load Pointer into DS	4.2
LEA	Load Effective Address	4.2
LES	Load Pointer into ES	4.2

Table E-1. (continued)

Mnemonic	*Description*	*Section*
LOCK	Lock Bus	4.6
LODS	Load Byte or Word (of string)	4.4
LODSB	Load Byte of string	4.4
LODSW	Load Word of string	4.4
LOOP	Loop	4.5
LOOPE	Loop While Equal	4.5
LOOPNE	Loop While Not Equal	4.5
LOOPNZ	Loop While Not Zero	4.5
LOOPZ	Loop While Zero	4.5
MOV	Move	4.2
MOVS	Move Byte or Word (of string)	4.4
MOVSB	Move Byte of string	4.4
MOVSW	Move Word of string	4.4
MUL	Multiply	4.3
NEG	Negate	4.3
NOT	Not	4.3
OR	Or	4.3
OUT	Output Byte or Word	4.2
POP	Pop	4.2
POPF	Pop Flags	4.2
PUSH	Push	4.2
PUSHF	Push Flags	4.2
RCL	Rotate through Carry Left	4.3
RCR	Rotate through Carry Right	4.3
REP	Repeat	4.4
RET	Return (intrasegment)	4.5
RETF	Return (intersegment)	4.5
ROL	Rotate Left	4.3
ROR	Rotate Right	4.3
SAHF	Store AH into Flags	4.2
SAL	Shift Arithmetic Left	4.3
SAR	Shift Arithmetic Right	4.3
SBB	Subtract with Borrow	4.3
SCAS	Scan Byte or Word (of string)	4.4
SCASB	Scan Byte of string	4.4
SCASW	Scan Word of string	4.4
SHL	Shift Left	4.3
SHR	Shift Right	4.3

Table E-1. (continued)

Mnemonic	*Description*	*Section*
STC	Set Carry	4.6
STD	Set Direction	4.6
STI	Set Interrupt	4.6
STOS	Store Byte or Word (of string)	4.4
STOSB	Store Byte of string	4.4
STOSW	Store Word of string	4.4
SUB	Subtract	4.3
TEST	Test	4.3
WAIT	Wait	4.6
XCHG	Exchange	4.2
XLAT	Translate	4.2
XOR	Exclusive Or	4.3

End of Appendix E

Appendix F
Sample Program APPF.A86


```
                        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  *
                        ;       **********************
```

Listing F-1. Sample Program APPF.A86

```
CP/M ASM86 1.08  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:
 0008 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 224708     and  al,readyinmasktab [bx]
 0016 7402       jz   constatout
 0018 B0FF       mov  al,0ffh
```

Listing F-1. (continued)

```
CP/M ASM86 1.08  SOURCE: APPF.A86      Terminal Input/Output          PAGE   3


                    constatout:
 001A 5A580AC0C3            pop dx ! pop bx ! or al,al ! ret
                    ;
                    ;
                    ;       *********
                    ;       * CONIN *
                    ;       *********
                    ;
                    ;       Entry: BL - reg = terminal no
                    ;       Exit:  AL - reg = read character
                    ;
 001F 53E82800      conin:  push bx ! call ckterminal !
 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 ckterminal
 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
```

Listing F-1. (continued)

```
CP/M ASM86 1.08  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 *
                 ;       *************************
```

Listing F-1. (continued)

```
CP/M ASM86 1.08  SOURCE: APPF.A86      Terminal Input/Output          PAGE   5


                   ;

 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
```

Listing F-1. (continued)

End of Appendix F

Appendix G
Code-macro 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>|
                      <nosegfix> | <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

End of Appendix G

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. Diagnostics messages report problems with the syntax and semantics of the program being assembled. The following messages indicate fatal errors ASM-86 encounters during assembly:

```
NO FILE
DISKETTE FULL
DIRECTORY FULL
DISKETTE 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
5	UNDEFINED INSTRUCTION
6	GARBAGE AT END OF LINE - IGNORED
7	OPERANDS MISMATCH INSTRUCTION
8	ILLEGAL INSTRUCTION OPERANDS

Table H-1. (continued)

Number	*Meaning*
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

End of Appendix H

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. This is a fatal error that terminates DDT-86 execution. Take appropriate action after checking to see if the correct disk is in the drive and that the disk is not write-protected.
DISK READ ERROR	The disk file specified in an R command could not be read properly. This is usually the result of an unexpected end-of-file. Correct the problem by regenerating the H86 file.
DISK WRITE ERROR	A disk write operation could not be successfully performed during a W command, probably due to a full disk. Erase files or obtain a disk with greater capacity.
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 can be allocated at a given time.

Table I-1. (continued)

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.

End of Appendix I

Index

U

V

W

X