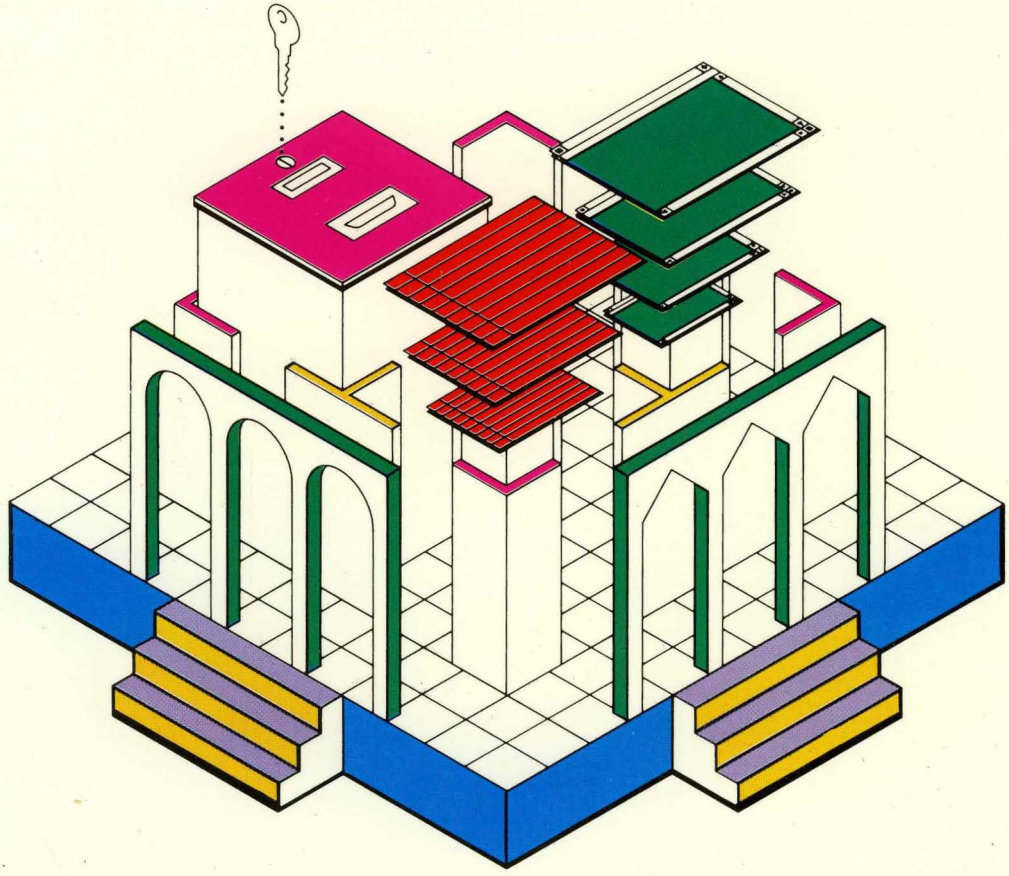
Concurrent™ DOS

Multiuser/Multitasking Operating System

PROGRAMMERS UTILITIES GUIDE



DIGITAL RESEARCH®

PROGRAMMER'S UTILITIES GUIDE

FOR CONCURRENT™ DOS 86 EXPANDED MEMORY (XM)

First Edition: November 1986

1065-2043-001

Foreword

The Programmer's Utilities Guide for Concurrent™ DOS 86 (cited as the Programmer's Utilities Guide) assumes that you are familiar with the Concurrent environment. It also assumes that you are familiar with the basic elements of 8086 and 80286 assembly language programming.

The Programmer's Utilities Guide describes the operation of the following Concurrent DOS 86 native mode (CMD files) and PC DOS mode (EXE files) utilities:

RASM-86™	Assembler for 8086, 80186, and 80286 assembly language that produces relocatable code in the Intel® Object Module Format.
XREF-86™	Utility used with RASM-86 to produce a cross reference listing of the symbols used in a program.
LINK-86™	Linkage editor that combines relocatable object modules and libraries to create executable files.
LIB-86™	Utility that creates and maintains libraries of object modules for use with LINK-86.
SID-86™	Symbolic Instruction Debugger used to test and debug object module code.

The operation of the native mode and PC DOS mode versions of each utility are nearly identical; any specific differences are clearly documented. Since Concurrent DOS 86 supports both its native mode operating system calls, and PC DOS system calls, it is imperative that you not intermix the two modes in an executable module. That is, a PC DOS program (.EXE file) should not contain any calls to Concurrent DOS 86 native mode system calls. Likewise, a program written for the native mode (.CMD file) should not contain any PC DOS system calls.

Typographical Conventions

This guide uses the following notation to describe commands:

command parameter [option]

A **command** is any of the commands described in this guide. A **parameter** can be a filename, an address location, or any specifier that is particular to the command. Optional items, such as command options or additional filenames, appear inside square brackets.

Words joined by an underscore (_) represent a single command item or field.

Examples of specific usage of a command are preceded by an A> prompt, and the user's input appears in **bold print**. For example:

```
A>rasm86 test
```

illustates a specific usage of the RASM86 command.

Characters used to represent values or variables in a command or instruction syntax may also appear in **bold print** in the text in which they are described.

Contents

6 XREF-86 Cross-Reference Utility

7 LINK-86 Linkage Editor

8 LIB-86 Library Utility

9 SID-86 Operation

10 SID-86 Expressions

Figures

Tables

The RASM-86 Assembler

1.1 Introduction

This section describes RASM-86 operation and its command syntax. Sections 2 through 5 detail the characteristics and uses of the RASM-86 components. A sample RASM-86 source file is provided in Appendix A.

1.2 RASM-86 Operation

The RASM-86 assembler converts source files containing 8086, 8087, 80186, 80286, and 80287 instructions into machine language object files. RASM-86 processes an assembly language source file in three passes and can produce three output files from one source file. The three files have the same filename as the source file. For example, if the name of the source file is BIOS88.A86, RASM-86 produces the files BIOS88.OBJ, BIOS88.LST, and BIOS88.SYM.

The LST list file contains the assembly language listing with any error messages. The OBJ object file contains the object code in Intel 8086 and 80286 relocatable object format. The SYM symbol file lists any user-defined symbols.

1.3 RASM-86 Command Syntax

Invoke RASM-86 with the following command form:

```
RASM86 [d:]filename[.typ] [$ run-time parameters]
```

where **filename** is the name of the source file. The filename can be any valid filename of 1 to 8 characters.

The **d:** is an optional drive specification denoting the source file's location. The drive specification is not needed if the source is on the current drive.

The **typ** is the optional filetype, which can be any valid filetype of 1 to 3 characters. If no filetype is specified, filetype A86 is assumed. The **run-time parameters** are described below in Section 1.3.1.

1.3.1 RASM-86 Run-Time Parameters

The dollar sign character, $, denotes an optional string of run-time parameters. A run-time parameter is followed by a device or file specification.

Table 1-1 contains a summary of the RASM-86 run-time parameters, described in detail in the following sections.

Table 1-1. RASM-86 Run-time Parameters

Parameter	Specifies	Valid Arguments
A	Source file device	A, B, C, ... P
IFILENAME	Include **filename** into assembly at beginning of module	
L	Local symbols in object file	O
O	Object file device	A ... P, Z
NC	No case conversion	
P	List file device	A ... P, X, Y, Z
S	Symbol file device	A ... P, X, Y, Z
186	Permit 186 opcodes	
286	Permit 286 opcodes	

All run-time parameters are optional, and you can enter them 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. However, no space is permitted between a parameter and its device name.

If you specify an invalid parameter in the parameter list, RASM-86 displays

```
SYNTAX ERROR
```

and echoes the command tail up to the point where the error occurs, then prints a question mark. (Appendix E contains the complete list of RASM-86 error messages.)

A, O, P, and S Parameters

These run-time parameters associate a filetype with a device. The file parameters: A, O, P, and S specify the type of file. Each of these parameters is followed by a device specifier: A - P, X, Y, Z. For example:

```
$ AA
```

specifies the source file on drive A.

The A, O, P, and S parameters have the following definitions:

A	specifies the Source File
O	specifies the Object File
P	specifies the List File
S	specifies the Symbol File

A device name must follow each of these parameters. The devices are defined as follows:

A - P	Specify disk drives A through P, respectively.
X	specifies the user console, CON:
Y	specifies the list device, LST:
Z	suppresses output, NUL:

If you direct the output to the console, you can temporarily stop the display by typing CTRL-S, then restart it by typing CTRL-Q.

IFILENAME Parameter

If a filename is preceded by an upper case I, RASM-86 includes the contents of the file at the beginning of the module being assembled. If no filename extension is specified, RASM-86 assumes an extension of A86.

L Parameter

The L parameter directs RASM-86 to include local symbols in the object file so that they appear in the SYM file created by LINK-86. Otherwise, only public symbols appear in the SYM file. You can use the SYM file with the Symbolic Instruction Debugger, SID-86™, to simplify program debugging.

NC Parameter

The NC parameter directs RASM-86 to distinquish between uppercase and lowercase in symbol names. Thus, when you specify the NC parameter, the symbols ABC and abc are treated as two unique symbols. If the NC parameter is not specified, RASM-86 would consider ABC and abc to be the same symbol. This parameter is useful when writing programs to be linked with other programs whose symbols might contain lowercase characters, such as "c".

186 and 286 Parameters

The 186 parameter specifies that 80186 opcodes are to be assembled.

The 286 parameter specifies that 80286 opcodes are to be assembled.

1.3.2 RASM-86 Command Line Examples

The following are some examples of valid RASM-86 commands:

Command Line	Result
`A>rasm86 io`	Assembles file IO.A86 and produces IO.OBJ, IO.LST, and IO.SYM, all on the default drive.
`A>rasm86 io.asm $ ad sz`	Assembles file IO.ASM on drive D and produces IO.LST and IO.OBJ. Suppresses the symbol file.
`A>rasm86 io $ py sx`	Assembles file IO.A86, produces IO.OBJ, and sends listing directly to printer. Also outputs symbols on console.

Command Line Result

`A>rasm86 io $ Ifirst`
Assembles file IO.A86 with the contents of the file, first.a86, appearing at the beginning of the module. Then produces IO.OBJ, IO.LST, and IO.SYM, all on the default drive.

`A>rasm86 io $ lo`
Includes local symbols in IO.OBJ.

Once you invoke RASM-86, it displays a sign-on message and then attempts to open the source file. If the file does not exist on the designated drive or does not have the correct filetype, RASM-86 displays:

NO FILE

and stops processing.

By default, RASM-86 creates the output files on the current disk drive. However, you can redirect the output files by using the optional parameters, or by a drive specification in the source filename. In the latter case, RASM-86 directs the output files to the drive specified in the source filename.

When the assembly is complete, RASM-86 displays the message:

END OF ASSEMBLY. NUMBER OF ERRORS: n USE FACTOR: pp%

where **n** represents the number of errors encountered during assembly. The **Use Factor** indicates how much of the available Symbol Table space was actually used during the assembly. The Use Factor is expressed as a decimal percentage ranging from 0 to 99.

1.4 Stopping RASM-86

You can halt both the native mode and PC DOS mode assembler by pressing Ctrl-C. You can also stop the native-mode assembler, by pressing any key on the console keyboard. When you press a key, RASM-86 responds:

```
STOP RASM-86 (Y/N)?
```

If you type Y, RASM-86 immediately stops processing, and returns control to the operating system. Type N to resume processing.

End of Section 1

Elements of RASM-86 Assembly Language

2.1 Introduction

This section describes the following elements of RASM-86 assembly language:

- RASM-86 Character Set
- Tokens and Separators
- Delimiters
- Constants
- Identifiers
- Operators
- Expressions
- Statements

2.2 RASM-86 Character Set

RASM-86 recognizes a subset of the ASCII character set. Valid RASM-86 characters are the letters A through Z (both uppercase and lowercase) and the numerals 0-9.

Valid special characters are:

```
+ - * / = ( ) [ ] ; ' . ! , _ : $ ?
```

Valid nonprinting characters are:

space, tab, carriage return, and line-feed

Usually RASM-86 treats lowercase letters as uppercase, except within strings. You can use the NC parameter described in Section 1.3.1 to make RASM-86 distinguish between lower and upper case. Only alphanumerics, special characters, and spaces can appear in a string.

2.3 Tokens and Separators

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

2.4 Delimiters

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

Table 2-1 describes RASM-86 separators and delimiters. Some delimiters are also operators. Operators are described in Section 2.7.

Table 2-1. Separators and Delimiters

Character	Name	Use
20H	space	separator
09H	tab	legal in source files, expanded in list files
CR	carriage return	terminates source lines
LF	line-feed	legal after CR; if in source lines, it is interpreted as a space
	semicolon	starts comment field
	colon	identifies a label; used in segment override specification

Table 2-1. (continued)

Character	Name	Use
	period	forms variables from numbers
$	dollar sign	notation for present value of location counter; legal, but ignored in identifiers or numbers
+	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

2.5 Constants

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

2.5.1 Numeric Constants

A numeric constant is a 16-bit integer value expressed in one of several bases. The base, called the radix of the constant, is denoted by a trailing radix indicator. Table 2-2 shows the radix indicators.

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

RASM-86 assumes that any numeric constant not terminating with a radix indicator is a decimal constant. Radix indicators can be uppercase or lowercase.

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 zeros and ones. 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). The leading character of a hexadecimal constant must be a decimal digit so RASM-86 cannot confuse a hex constant with an identifier. The following are valid numeric constants:

```
1234     1234D    1100B    1111000011110000B
1234H    0FFEH    3377O    13772Q
33770    0FE3H    1234d    0ffffh
```

2.5.2 Character String Constants

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

The numeric value of a character is its ASCII code. RASM-86 does not translate case in character strings, so you can use both uppercase and lowercase letters. Note that RASM-86 allows only alphanumerics, special characters, and spaces in character strings.

A DB directive is the only RASM-86 statement that can contain strings longer than two characters (see Section 3.7.1). The string cannot exceed 255 bytes. If you want to include an apostrophe in the string, you must enter it twice. RASM-86 interprets two apostrophes together as a single apostrophe. Table 2-3 shows valid character strings and how they appear after processing.

Table 2-3. String Constant Examples

String in source text	As processed by RASM-86
'a'	a
'Ab''Cd'	Ab'Cd
'I like CP/M'	I like CP/M
''''	'
'ONLY UPPER CASE'	ONLY UPPER CASE
'only lower case'	only lower case

2.6 Identifiers

The following rules apply to all identifiers:

- Identifiers can be up to 80 characters long.
- The first character must be alphabetic or one of these special characters: ?, @, or _
- Any subsequent characters can be either alphabetic, numeric, or one of these special characters: ?, @, _, or $. RASM-86 ignores the special character $ in identifiers, so that you can use it to improve readability in long identifiers. For example, RASM-86 treats the identifier interrupt$flag as interruptflag.

There are two types of identifiers:

Keywords

Symbols

Keywords have predefined meanings to RASM-86. Symbols are identifiers you define yourself.

2.6.1 Keyword Identifiers

Keywords are reserved for use by RASM-86; you cannot define an identifier identical to a keyword. Appendix C lists the keywords.

RASM-86 recognizes five types of keywords:

- instructions
- directives
- operators
- registers
- predefined numbers

Section 4 defines the 8086, 8087, 80186, 80286, and 80287 instruction mnemonic keywords and the actions they initiate; Section 3 discusses RASM-86 directives, and Section 2.7 defines operators. Table 2-4 lists the RASM-86 keywords that identify the processor registers.

Three keywords, BYTE, WORD, and DWORD, are predefined numbers. The values of these numbers are 1, 2, and 4, respectively. RASM-86 also associates a Type attribute 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 (bytes)	Numeric Value	Meaning
AH	1	100 B	Accumulator High Byte
BH	1	111 B	Base Register High Byte
CH	1	101 B	Count Register High Byte
DH	1	110 B	Data Register High Byte
AL	1	000 B	Accumulator Low Byte
BL	1	011 B	Base Register Low Byte
CL	1	001 B	Count Register Low Byte
DL	1	010 B	Data Register Low Byte
AX	2	000 B	Accumulator (full word)
BX	2	011 B	Base Register (full word)
CX	2	001 B	Count Register (full word)
DX	2	010 B	Data Register (full word)
BP	2	101 B	Base Pointer
SP	2	100 B	Stack Pointer
SI	2	110 B	Source Index
DI	2	111 B	Destination Index
CS	2	01 B	Code Segment Register
DS	2	11 B	Data Segment Register
SS	2	10 B	Stack Segment Register
ES	2	00 B	Extra Segment Register
ST	10	000 B	8087 stack top register
ST0	10	000 B	8087 stack top register
ST1	10	001 B	8087 stack top - 1 is stack top
--	--	--	--
ST7	10	111 B	8087 stack top - 7 is stack top

2.6.2 Symbol Identifiers

A symbol is a user-defined identifier with 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 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 containing its starting address (see "Segment Control Directives" in Section 3.3). The segment's starting address is a number between 0 and 0FFFFH (65,535D). This number indicates the paragraph in memory to which the current segment is assigned, either when the program is assembled, linked, or loaded.

The offset of a variable is the address of the variable relative to the starting address of the segment. The offset can be any number between 0 and 0FFFFH.

A variable has one of the following type attributes:

BYTE	one-byte variable
WORD	two-byte variable
DWORD	four-byte variable

The data definition directives: DB, DW, and DD, define a variable as one of these three types (see Section 3). For example, the variable, my_variable, is defined when it appears as the name for a data definition directive:

```
my_variable db 0
```

You can also define a variable as the name for an EQU directive referencing another variable, as shown in the following example:

```
another_variable equ my_variable
```

Labels

Labels identify locations in memory containing 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. A label is defined when it precedes an instruction. A colon separates the label from instruction. For example,

```
my_label:   add  ax,bx
```

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

```
another_label   equ   my_label
```

Numbers

You can also define numbers as symbols. RASM-86 treats a number symbol as though you have explicitly coded the number it represents. For example,

```
number_five    equ   5
mov   al,Number_five
```

is equivalent to the following:

```
mov   al,5
```

Section 2.7 describes operators and their effects on numbers and number symbols.

2.6.3 Example Identifiers

The following are valid identifiers:

```
NOLIST
WORD
AH
Mean_streets
crashed
variable number 1234567890
```

2.7 Operators

This section describes the available RASM-86 operators. RASM-86 operators define the operations forming the values used in the final assembly instruction.

RASM-86 operators fall into the following categories:

- arithmetic
- logical
- relational
- segment override
- variable manipulation
- variable creation

Table 2-5 summarizes the available RASM-86 operators and the number of the section where each operator is explained in detail.

Table 2-5. RASM-86 Operator Summary

Operator	Description	Section
+	addition or unary positive	2.7.1
-	subtraction or unary negative	2.7.1
*	multiplication	2.7.1
/	unsigned division	2.7.1

Table 2-5. (Continued)

Operator	Description	Section
.	create variable, assign offset	2.7.6
$	create label, offset = location counter	2.7.6
AND	logical AND	2.7.2
EQ	Equal to	2.7.3
GE	Greater than or equal to	2.7.3
GT	Greater than	2.7.3
LAST	compare LENGTH of variable to 0	2.7.5
LE	Less than or equal to	2.7.3
LENGTH	create number from variable length	2.7.5
LT	Less than	2.7.3
MOD	return remainder of division	2.7.1
NE	Not Equal to	2.7.3
NOT	logical NOT	2.7.2
OFFSET	create number from variable offset	2.7.5
OR	logical OR	2.7.2
PTR	create variable or label, assign type	2.7.6
seg:addr	override segment register	2.7.4
SEG	create number from variable segment	2.7.5
SHR	shift right	2.7.1
SHL	shift left	2.7.1
TYPE	create number from variable type	2.7.5
XOR	logical eXclusive OR	2.7.2

The following sections define the RASM-86 operators in detail. Where the syntax of the operator is illustrated, **a** and **b** represent two elements of the expression. Unless otherwise specified, a and b represent absolute numbers, such as numeric constants, whose value is known at assembly-time. A relocatable number, on the other hand, is a number whose value is unknown at assembly-time, because it can change during the linking process. For example, the offset of a variable located in a segment that will be combined with some other segments at link-time is a relocatable number.

2.7.1 Arithmetic Operators

Addition and Subtraction

Addition and-subtraction operators compute the arithmetic sum and difference of two operands. The first operand (**a**) can be a variable, label, an absolute number, or a relocatable number. For addition, the second operand (**b**) must be a number. For subtraction, the second operand can be a number, or it can be a variable or label in the same segment as the first operand.

When a number is added to a variable or label, the result is a variable or label with an offset whose numeric value is the second operand plus the offset of the first operand. Subtraction from a variable or label returns a variable or label whose offset is the first operand's offset, decremented by the number specified in the second operand.

Syntax:

a + b	returns the sum of **a** and **b**. Where **a** is a variable, label, absolute number, or relocatable number.
a - b	returns the difference of **a** and **b**. Where **a** and **b** are variables, labels, absolute numbers, or relocatable numbers in the same segment.

Example:

```
  0002              count   equ   2
  0005              displ   equ   5
000A FF             flag    db    offh
                            .
                            .
                            .
000B 2EA00B00               mov   al,flag+1
000F 2E8A0E0F00             mov   cl,flag+displ
0014 B303                   mov   bl,displ-count
```

Multiplication and Division

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

Syntax:

a * b	unsigned multiplication of **a** and **b**
a / b	unsigned division of **a** and **b**
a MOD b	return remainder of **a** / **b**
a SHL b	returns the value resulting from shifting **a** to left by the amount specified by **b**
a SHR b	returns the value resulting from shifting **a** to the right by an the amount specified by **b**

Example:

```
0016 BE5500                     mov     si,256/3
0019 B310                       mov     bl,64/4
  0050          buffersize      equ     80
001B B8A000                     mov     ax,buffersize * 2
```

Unary

Unary operators specify a number as either positive or negative. RASM-86 unary operators accept both signed and unsigned numbers.

Syntax:

+ a	gives **a**
- a	gives 0 - **a**

Example:

```
001E B123                mov   cl,+35
0020 B007                mov   al,2--5
0022 B2F4                mov   dl,-12
```

2.7.2 Logical Operators

Logical operators accept only numbers as operands. They perform the Boolean logic operations AND, OR, XOR, and NOT.

Syntax:

a XOR b	bit-by-bit logical EXCLUSIVE OR of **a** and **b**
a OR b	bit-by-bit logical OR of **a** and **b**
a AND b	bit-by-bit logical AND of **a** and **b**
NOT a	logical inverse of **a**: all 0s become 1s, 1s become 0s. (**a** is a 16-bit number.)

Example:

```
  00FC          mask    equ   0fch
  0080          signbit equ   80h
0000 B180               mov   cl,mask and signbit
0002 B003               mov   al,not mask
```

2.7.3 Relational Operators

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.

Syntax:

In all of the operators below, **a** and **b** are unsigned numbers; or they are labels, variables, or relocatable numbers defined in the same segment.

a EQ b	returns 0FFFFH if **a** = **b** otherwise 0
a LT b	returns 0FFFFH if **a** < **b**, otherwise 0
a LE b	returns 0FFFFH if **a** <= **b**, otherwise 0
a GT b	returns 0FFFFH if **a** > **b**, otherwise 0
a GE b	returns 0FFFFH if **a** >= **b**, otherwise 0
a NE b	returns 0FFFFH if **a**<> **b**, otherwise 0

Example:

```
000A            limit1  equ   10
0019            limit2  equ   25
                        .
                        .
                        .
0004 B8FFFF             mov   ax,limit1 lt limit2
0007 B80000             mov   ax,limit1 gt limit2
```

2.7.4 Segment Override Operator

When manipulating variables, RASM-86 decides which segment register to use. You can override this choice by specifying a different register with the segment override operator.

In the syntax below, seg:addr represents the segment register (seg) and the address of the expression (addr).

Syntax:

seg:addr	overrides segment register selected by assembler. seg can be: CS, DS, SS, or ES

Example:

```
0024 368B472D          mov   ax,ss:wordbuffer[bx]
0028 268B0E5B00        mov   cx,es:array
002D 26A4              movs  byte ptr [di],es:[si]
```

2.7.5 Variable Manipulation Operators

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 zero. If LENGTH is greater than zero, LAST decrements LENGTH by one. If LENGTH equals zero, LAST leaves it unchanged. Variable manipulators accept only variables as operators.

Syntax:

SEG a	creates a number whose value is the segment value of the variable or label **a**.
OFFSET a	creates a number whose value is the offset value of the variable or label **a**.
TYPE a	creates a number. If the variable **a** is of type BYTE, WORD or DWORD, the value of the number created is 1, 2, or 4, respectively.
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.
LAST a	if LENGTH **a** > 0, then LAST **a** = LENGTH **a** - 1; if LENGTH **a** = 0, then LAST **a** = 0.

Example:

```
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
```

2.7.6 Variable Creation Operators

Three RASM-86 operators are used to create variables. These are the PTR, period, and dollar sign operators described below.

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

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 dollar sign operator ($) creates a label with an offset attribute equal to the current value of the location counter. The label segment value is the same as the current segment. This operator takes no operand.

Syntax:

a PTR b	creates virtual variable or label with type of **a** and attributes of **b**. **a** can be a BYTE, WORD, or DWORD; **b** is the address of the expression.
.a	creates variable with an offset attribute of **a**. Segment attribute is current data segment.

$	creates label with offset equal to current value of location counter; segment attribute is current segment.

Examples:

```
0044 C60705              mov   byte ptr [bx], 5
0047 8A07                mov   al,byte ptr [bx]
0049 FF04                inc   word ptr [si]


004B A10000              mov   ax, .0
004E 268B1E0040          mov   bx, es:.4000h


0053 E9FDFF              jmp   $
0056 EBFE                jmps  $
0058 E9FD2F              jmp   $+3000h
```

2.7.7 Operator Precedence

Expressions combine variables, labels, or numbers with operators. RASM-86 allows several kinds of expressions (see Section 2.8). This section defines the order that RASM-86 performs operations if more than one operator appear in an expression.

RASM-86 evaluates expressions from left to right, but evaluates operators with higher precedence before operators with lower precedence. When two operators have equal precedence, RASM-86 evaluates the leftmost operator first. Table 2-6 shows RASM-86 operators in order of increasing precedence

You can use parentheses to override the precedence rules. RASM-86 first evaluates the part of an expression enclosed in parentheses. If you nest parentheses, RASM-86 evaluates the innermost expressions first.

For example,

```
15/3 + 18/9 = 5 + 2 = 7
15/(3 + 18/9) = 15/(3 + 2) = 15/5 = 3
(20*4) + ((27/9 - 4/2)) = (20*4)+(3-2) = 80+1 = 81
```

Note that RASM-86 allows five levels of nested parentheses.

Table 2-6. Precedence of Operations in RASM-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	$

2.8 Expressions

RASM-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. You can combine numbers with operators such as PTR to make an address expression.

A numeric expression evaluates to a number. It contains no 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.

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

```
mov  variable[bx],0
mov  ax,[bx+di]
mov  ax,[si]
mov  bl,[si]
```

Since all of the above instructions are memory references, the current DS Segment Selector is implied. The first instruction moves the value of 0 hex into the word location specified by the sum of the base register BX and the displacement VARIABLE. The second instruction moves the word found at the location specified by the sum of the base register BX and the index register DI into the location specified by the word register AX. The third instruction moves the word found at the location specified by index register SI into the location specified by the word register AX. The last instruction moves the byte found at the location specified by the index register SI into the location specified by the byte register BL.

2.9 Statements

Statements can be instructions or directives. RASM-86 translates instructions into 8086 and 80286 machine language instructions. RASM-86 does not translate directives into machine code. Directives tell RASM-86 to perform certain functions.

You must terminate each assembly language statement with a carriage return (CR) and line-feed (LF), or exclamation point. RASM-86 treats these as an end-of-line. You can write multiple assembly language statements without comments on the same physical line and separate them with exclamation points. Only the last statement on a line can have a comment because the comment field extends to the physical end of the line.

2.9.1 Instruction Statements

The following is the syntax for an instruction statement:

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

The fields are defined as follows:

label	A symbol followed by a colon 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 RASM-86 or by an EQU directive. This field is optional unless preceded by a prefix instruction. If you omit this field, no operands can be present, although the other fields can appear. Section 4 describes the RASM-86 mnemonics.
operand(s)	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. This field is optional, but you should use comments to facilitate program maintenance and debugging.

Section 3 describes the RASM-86 directives.

2.9.2 Directive Statements

The following is the syntax for a directive statement:

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

The fields are defined as follows:

name — Names are legal for CSEG, DSEG, ESEG, SSEG, GROUP, DB, DW, DD, RB, RW, RD, RS, and EQU directives. The name is required for the EQU and GROUP directives, but it is optional for the other directives. Unlike the label field of an instruction, the name field of a directive is never terminated with a colon.

directive — One of the directive keywords defined in Section 3.

operand(s) — Analogous to the operands for instruction mnemonics. Some directives, such as DB and DW allow any operand; others have special requirements.

comment — Exactly as defined for instruction statements in Section 2.9.1.

End of Section 2

RASM-86 Directives

3.1 Introduction

RASM-86 directives control the assembly process by performing functions such as assigning portions of code to logical segments, requesting conditional assembly, defining data items, allocating memory, specifying listing file format, and including source text from external files.

RASM-86 directives are grouped into the following categories:

- segment control
- linkage control
- conditional assembly
- symbol definition
- data definition and memory allocation
- output listing control
- 8087 control
- miscellaneous

3.2 Assembler Directive Syntax

The following is the general syntax for a directive statement:

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

The fields are defined as follows:

name — Is a symbol that retains the value assigned by the directive. A name is required for the EQU and GROUP directives, but it is optional for the other directives. Unlike the label field of an instruction, the name field of a directive is never terminated with a colon. Names can be used with the CSEG, DSEG, ESEG, SSEG, GROUP, DB, DW, DD, RB, RW, RD, RS, and EQU directives.

directive	One of the directive keywords defined in Sectio 3.3 through 3.10.
operand(s)	Analogous to the operands for instructic mnemonics. Some directives, such as DB and D' allow any operand; others have speci requirements.
comment	Exactly as defined for instruction statements i Section 2.9.1.

The following sections describe each RASM-86 directive. The synta for each directive follows each section heading.

3.3 Segment Control Directives

This section describes the RASM-86 directives used to assign specific attributes to segments. These attributes affect the way the segments are handled during the link process. The available segment control directives are:

CSEG
DSEG
ESEG
SSEG
GROUP

In order to utilize these directives, you must understand the segmented architecture of the 8086 and 80286 processors. The following section summarizes the general characteristics of the 8086/80286 segmented architecture.

3.3.1 The 8086/80286 Segmented Architecture

The address space of an 8086 or an 80286 processor can be subdivided into an arbitrary number of units called segments. Each segment is comprised of contiguous memory locations, up to 64k bytes in length, making up logically independent and seperately addressable units.

Each segment must have a base address specifying its starting location in the memory space. Each segment base address must begin on a paragraph boundary, a boundary divisible by 16.

Every location in the memory space has a physical address and a logical address. A physical address is a value specifying a unique byte location within the memory space. A logical address is the combination of a segment base value and an offset value. The offset value is the address relative to the base of the segment. At run-time, every memory reference is the combination of a segment base value and an offset value that produces a physical address. A physical address can be contained in more than one logical segment.

The CPU can access four segments at a time. The base address of each segment is contained in a segment register. The CS register points to the current code segment that contains instructions. The DS register points to the current data segment usually containing program variables. The SS register points to the current stack segment where stack operations such as temporary storage or parameter passing are performed. The ES register points to the current Extra Segment that typically also contains data.

RASM-86 segment directives allow you to divide your assembly language source program into segments corresponding to the memory segments where the resulting object code is loaded at run-time.

The size, type, and number of segments required by a program defines which memory model the operating system should use to allocate memory. Depending on which model you use, you can intermix all of the code and data in a single 64K segment, or you can have separate Code and Data Segments, each up to 64K in length. The RASM-86 segment directives described below, allow you to create an arbitrary number of Code, Data, Stack, and Extra Segments to more fully use the address space of the processor. You can have more than 64K of code or data by using several segments and managing the segments with the assembler directives.

3.3.2 CSEG, DSEG, ESEG, and SSEG Directives

Every instruction and variable in a program must be contained in a segment. Segment directives allow you to specify the attributes of a segment or a group of segments of the same type.

Create a segment and name it by using the segment directive syntax:

[seg_name] seg_directive [align_type] [combine_type] ['class_name']

where **seg_directive** is one of the following:

CSEG (Code Segment)
DSEG (Data Segment)
ESEG (Extra Segment)
SSEG (Stack Segment)

The optional parameters are described below; note that **class-name** is applicable only to native mode programs. Examples illustrating how segment directives are used are provided at the end of this section.

seg_name

The segment name can be any valid RASM-86 identifier. If you do not specify a segment name, RASM-86 supplies a default name, as shown in Table 3-1.

Table 3-1. Default Segment Names

Segment Directive	Default Segment Name
CSEG	CODE
DSEG	DATA
ESEG	EXTRA
SSEG	STACK

Once you use a segment directive, RASM-86 assigns statements to the specified segment until it encounters another segment directive. RASM-86 combines all segments with the same segment name even if they are not contiguous in the source code.

align_type

The align type allows you to specify to the linkage editor a particular boundary for the segment. The linkage editor uses this alignment information when combining segments to produce an executable file. You can specify one of four different align types:

- BYTE (byte alignment)
- WORD (word alignment)
- PARA (paragraph alignment)
- PAGE (page alignment)

If you specify an align type, it must be with the first definition of the segment. You can omit the align type on subsequent segment directives that name the same segment, but you cannot change the original value. If you do not specify an align type, RASM-86 supplies a default value based on the type of segment directive used. Table 3-2 shows the default values.

Table 3-2. Default Align Types

Segment Directive	Default Align Type
CSEG	BYTE
DSEG	WORD
ESEG	WORD
SSEG	WORD

BYTE alignment means that the segment begins at the next byte following the previous segment.

WORD alignment means that the segment begins on an even boundary. An even boundary is a hexadecimal address ending in 0, 2, 4, 6, 8, A, C, or E. In certain cases, WORD alignment can increase execution speed because the CPU takes only one memory cycle when accessing word-length variables within a segment aligned on an even boundary. Two cycles are needed if the boundary is odd.

PARA (paragraph) alignment means that the segment begins on a paragraph boundary, that is, an address whose four low-order bits are zero.

PAGE alignment means that the segment begins on a page boundary, an address whose low order byte is zero.

combine_type

The combine type determines how the linkage editor can combine the segment with other segments with the same segment name. You can specify one of five different combine types:

- PUBLIC
- COMMON
- STACK
- LOCAL
- nnnn (absolute segment)

If you specify a combine type, it must be in the first segment directive for that segment type. You can omit the combine type on subsequent segment directives for the same segment type, but you cannot change the original combine type. If you do not specify a combine type, RASM-86 supplies the PUBLIC combine type by default; except for SSEG, which uses the STACK combine type by default.

The RASM-86 combine types are defined as follows:

PUBLIC — means that the linkage editor can combine the segment with other segments having the same name. All such segments with combine type PUBLIC are concatenated in the order they are encountered by the linkage editor, with gaps, if any, determined by the align type of the segment.

COMMON — means that the segment shares identical memory locations with other segments of the same name. Offsets inside a COMMON segment are absolute unless the segment is contained in a GROUP (see "Group Directive" in this section).

STACK is similar to PUBLIC, in that the storage allocated for STACK segments is the sum of the STACK segments from each module. However, instead of concatenating segments with the same name, the linkage editor overlays STACK segments against high memory, because stacks grow downward from high addresses to low addresses when the program runs.

LOCAL means that the segment is local to the program being assembled, and the linkage editor will not combine it with any other segments.

ABSOLUTE SEGMENT
causes RASM-86 to determine the load-time position of the segment during assembly, rather than allowing its position to be determined by the linkage editor, or at load time.

class_name (native-mode only)

The class name can be any valid RASM-86 identifier. The class name identifies segments to be placed in the same section of the CMD file created by LINK-86. Unless overridden by a GROUP directive or an explicit command in the LINK-86 command line, LINK-86 places segments into the CMD file it creates as shown in Table 3-3.

Table 3-3. Default Class Name for Segments

Segment Directive	Default Class Name	Section of CMD file
CSEG	CODE	CODE
DSEG	DATA	DATA
ESEG	EXTRA	EXTRA
SSEG	STACK	STACK

Examples:

The following are examples of segment directives:

```
CSEG
DSEG
CSEG          PAGE
DATASEG       DSEG        PARA        'DATA'
CODE1         CSEG        BYTE
XYZ           DSEG        WORD        COMMON
```

The example RASM-86 source file in Appendix A illustrates how segment directives are used.

3.3.3 GROUP Directive

```
group_name GROUP segment_name1, segment_name2, ...
```

The GROUP directive instructs RASM-86 to combine the named segments into a collection called a group whose length can be up to 64K bytes. When segments are grouped together, LINK-86 treats the group as it would a single segment by making the offsets within the segments of a group relative to the beginning of the group rather than to the beginning of the individual segments.

The order of the segment names in the directive is the order that LINK-86 arranges the segments in the executable file.

Use of groups can result in more efficient code, because a number of segments can be addressed from a single segment register without having to change the contents of the segment register.

See Section 7.12 for more information on the grouping and other link processes.

3.4 Linkage Control Directives

Linkage control directives modify the link process. The available linkage control directives are:

END
NAME
PUBLIC
EXTRN

3.4.1 END Directive

END [start_label]

The END directive marks the end of a source file. RASM-86 ignores any subsequent lines. The END directive is optional, and if omitted, RASM-86 processes the source file until it finds an end-of-file character (1AH).

The optional start label serves two purposes. First it defines the current module as the main program. When LINK-86 links modules together, only one can be a main program. Second, start label indicates where the program is to start executing after it is loaded. If start label is omitted, program execution begins at the beginning of the first CSEG from the files linked.

3.4.2 NAME Directive

NAME module_name

The NAME directive assigns a name to the object module generated by RASM-86. The module name can be any valid identifier based on the guidelines described in Section 2.6. If you do not specify a module name with the NAME directive, RASM-86 assigns the source filename to the object module. Both LINK-86 and LIB-86 use NAME directives to identify object modules.

3.4.3 PUBLIC Directive

PUBLIC name [, name, ...]

The PUBLIC directive instructs RASM-86 that the names defined as PUBLIC can be referenced by other programs linked together. Each name must be a label, variable, or a number defined within the program being assembled.

3.4.4 EXTRN Directive

```
EXTRN    external_id [,external_id, ...]
```

The EXTRN directive tells RASM-86 that each external id can be referenced in the program being assembled but is defined in some other program. The external id consists of two parts: a symbol and a type.

The external id uses the form:

```
symbol:type
```

where "symbol" is a variable, label, or number and "type" is one of the following:

- Variables: BYTE, WORD, or DWORD
- Labels: NEAR or FAR
- Numbers: ABS

For example,

```
EXTRN    FCB:BYTE,BUFFER:WORD,INIT:FAR,MAX:ABS
```

RASM-86 determines the Segment attribute of external variables and labels from the segment containing the EXTRN directive. Thus, an EXTRN directive for a given symbol must appear within the same segment as the module in which the symbol is defined.

3.5 Conditional Assembly Directives

Conditional assembly directives are used to set up conditions controlling the instruction sequence. The available conditional assembly directives are:

```
IF
ELSE
ENDIF
```

3.5.1 IF, ELSE, and ENDIF Directives

```
IF      numeric expression
        source line 1
        source line 2
                .
                .
                .
        source line n

ELSE
        alternate source line 1
        alternate source line 2
                .
                .
                .
        alternate source line n

ENDIF
```

The IF and ENDIF directives allow you to conditionally include or exclude a group of source lines from the assembly. The optional ELSE directive allows you to specify an alternative set of source lines. You can use these conditional directives to assemble several different versions of a single source program. You can nest IF directives to five levels.

When RASM-86 encounters an IF directive, it evaluates the numeric expression following the IF keyword. You must define all elements in the numeric expression before you use them in the IF directive. If the value of the expression is nonzero, then RASM-86 assembles source line 1 through source line n. If the value of the expression is zero, then RASM-86 lists all the lines, but does not assemble them.

If the value of the expression is zero, and you specify an ELSE directive between the IF and ENDIF directives, RASM-86 assembles alternative source lines 1 through alternative source lines n.

3.6 Symbol Definition Directive

The available symbol definition directive is:

EQU

3.6.1 EQU Directive

```
symbol_name  EQU  numeric_expression
symbol_name  EQU  address_expression
symbol_name  EQU  register
symbol_name  EQU  instruction_mnemonic
```

The EQU (equate) directive assigns values and attributes to user-defined symbols. Do not put a colon after the symbol name. Once you define a symbol, you cannot redefine the symbol with a subsequent EQU or another directive. You must also define any elements used in numeric expressions or an address expression before using the EQU directive.

The first form of the EQU directive assigns a numeric value to the symbol. The second form assigns a memory address. The third form assigns a new name to an 8086 or 80286 register. The fourth form defines a new instruction (sub)set. The following are examples of these four EQU forms.

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

3.7 Data and Memory Directives

Data definition and memory allocation directives define the storage format used for a specified expression or constant. The available data definition and memory allocation directives are:

DB
DW
DD
RS
RB
RW
RD

3.7.1 DB Directive

[symbol] DB numeric_expression [,numeric_expression...]
[symbol] DB string_constant [,string_constant...]

The DB directive defines initialized storage areas in byte format. RASM-86 evaluates numeric expressions to 8-bit values and sequentially places them in the object file. RASM-86 places string constants in the object file according to the rules defined in Section 2.5.2. Note that RASM-86 does not perform translation from lower- to uppercase within strings.

The DB directive is the only RASM-86 statement that accepts a string constant longer than two bytes. You can add multiple expressions or constants, separated by commas, to the definition if it does not exceed the physical line length.

Use an optional symbol to reference the defined data area throughout the program. The symbol has four attributes:

- segment
- offset
- type
- length

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 reserved.

The following listing shows examples of DB directives and the resulting hexadecimal values:

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

3.7.2 DW Directive

[symbol] DW numeric_expression [,numeric_expression...]
[symbol] DW string_constant [,string_constant...]

The DW directive initializes two-byte words of storage. The DW directive initializes storage the same way as the DB directive, except that each numeric expression, or string constant initializes two bytes of memory with the low-order byte stored first. The DW directive does not accept string constants longer than two characters.

The following are examples of DW directives:

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

3.7.3 DD Directive

[symbol] DD address_expression [,address_expression...]

The DD directive initializes four bytes of storage. DD follows the same procedure as DB, except that the offset attribute of the address expression is stored in the two lower bytes and the segment attribute is stored in the two upper bytes. For example,

```
                                CSEG

0000 6CC100006FC1 LONG_JMPTAB   DD      ROUT1,ROUT2
     0000
0008 72C1000075C1               DD      ROUT3,ROUT4
     0000
```

3.7.4 RS Directive

[symbol] RS numeric_expression

The RS directive allocates storage in memory but does not initialize it. The numeric expression gives the number of bytes to reserve. Note that the RS directive just allocates memory without specifying byte, word, or long attributes. For example,

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

3.7.5 RB Directive

[symbol] RB numeric_expression

The RB directive allocates byte storage in memory without any initialization. The RB directive is identical to the RS directive except that it gives the byte attribute. For example,

```
4061                BUF     RB      48
4161                        RB      4000H
C161                        RB      1
```

3.7.6 RW Directive

[symbol] RW numeric_expression

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

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

3.7.7 RD Directive

[symbol] RD numeric_expression

The RD directive reserves a double word (four bytes) of storage but does not initialize it. For example,

```
C163            DWTAB   RD      4
C173                    RD      1
```

3.8 Output Listing Control Directives

Output listing control directives modify the list file format. The available output listing control directives are:

EJECT
IFLIST
NOIFLIST
LIST
NOLIST
PAGESIZE
PAGEWIDTH
SIMFORM
TITLE

3.8.1 EJECT Directive

EJECT

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

3.8.2 NOIFLIST/IFLIST Directives

NOIFLIST
IFLIST

The NOIFLIST directive suppresses the printout of the contents of conditional assembly blocks that are not assembled The IFLIST directive resumes printout of these blocks.

3.8.3 NOLIST and LIST Directives

NOLIST
LIST

The NOLIST directive suppresses the printout of lines following the directive. The LIST directive restarts the listing.

3.8.4 PAGESIZE Directive

PAGESIZE numeric_expression

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

3.8.5 PAGEWIDTH Directive

PAGEWIDTH numeric_expression

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

3.8.6 SIMFORM Directive

SIMFORM

The SIMFORM directive replaces a form-feed (FF) character in the list file with the correct number of line-feeds (LF). Use this directive when directing a list file to a printer unable to interpret the form-feed character.

3.8.7 TITLE Directive

TITLE string_constant

RASM-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 can be up to 30 characters in length. For example,

TITLE 'CP/M monitor'

3.9 8087 Control Directives

An Intel 8087 coprocessor is not available on all systems. When writing programs using 8087 opcodes, you can use the 8087 control directives to instruct RASM-86 to either generate actual 8087 opcodes or to emulate the 8087 routines in software. The available 8087 control directives are:

HARD8087
AUTO8087

3.9.1 HARD8087 Directive

HARD8087

When an 8087 processor is available on your system and you do not want to emulate the 8087 routines in software, you can use the HARD8087 directive to instruct RASM-86 to generate 8087 opcodes. Using this option saves about 16K bytes of space that would otherwise be used by the emulation routines.

3.9.2 AUTO8087 Directive

AUTO8087

You can use the AUTO8087 option to create programs that decide at runtime whether or not to use the 8087 processor. AUTO8087 is the default option. When you use this option, LINK-86 includes in the command file the 8087 emulation routines and a table of fixup records that point to the 8087 opcodes.

If you use the AUTO8087 option and the system has an 8087, the 8087 fixup table is ignored and the space occupied by the emulation routines is released to the program for heap space. If the system does not have an 8087, the initialization routine replaces all the 8087 opcodes with interrupts that vector into the 8087 emulation routines.

Note that, in order to emulate 8087 routines, you must have a runtime library from a Digital Reasearch high-level language, such as DR C or CBASIC present on your disk and it must be specified on the LINK-86 command line.

3.10 Miscellaneous Directives

Additional RASM-86 directives are:

INCLUDE
ORG

3.10.1 INCLUDE Directive

```
INCLUDE    filename
```

The INCLUDE directive includes another RASM-86 source file in the source text. For example, to include the file EQUALS in your text, you would enter:

```
INCLUDE  EQUALS.A86
```

You can use the INCLUDE directive when the source program is large and resides in several files. Note that you cannot nest INCLUDE directives; a source file called by an INCLUDE directive cannot contain another INCLUDE directive.

If the file named in the INCLUDE directive does not have a filetype, RASM-86 assumes the filetype to be A86. If you do not specify a drive name with the file, RASM-86 uses the drive containing the source file.

3.10.2 ORG Directive

ORG numeric_expression

The ORG directive sets the offset of the location counter in the current segment to a value specified by the numeric expression. You must define all elements of the expression before using the ORG directive, and the expression must evaluate to an absolute number.

The offset specified by the numeric expression is relative to the offset specified by the location counter within the segment at load-time. Thus, if you use an ORG statement in a segment that the linkage editor does not combine with other segments at link-time, such as LOCAL or absolute segments, then the numeric expression indicates the actual offset within the segment.

If the segment is combined with others at link-time, such as PUBLIC segments, then numeric expression is not an absolute offset. It is relative to the beginning address of the segment, from the program being assembled.

When using the ORG directive, never assume the align type. The desired align type should always be explicitly declared. For example, if you use the command:

ORG 0

The segments must be aligned on a paragraph boundary. Therefore, the PARAGRAPH align type must have been specifically declared.

End of Section 3

RASM-86 Instruction Set

4.1 Introduction

The RASM-86 instruction set includes all 8086, 8087, 80186, and 80286 machine instructions. The general syntax for instruction statements is described in Section 2.9. This section defines 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 the Intel assembly language reference manual for the processor you are using. For descriptions of the instruction bit patterns and operations, see the Intel user's manual for the processor you are using.

The instruction-definition tables present RASM-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.

4.2 RASM-86 Instruction Set Summary

Table 4-1 summarizes the complete RASM-86 instruction set in alphabetical order. The following tables categorize these instructions into functional groups in which each instruction is defined in detail.

Table 4-1. RASM-86 Instruction Summary

Mnemonic	Description	Section
AAA	ASCII adjust for Addition	4.3.4
AAD	ASCII adjust for Division	4.3.4
AAM	ASCII adjust for Multiplication	4.3.4
AAS	ASCII adjust for Subtraction	4.3.4
ADC	Add with Carry	4.3.4
ADD	Add	4.3.4
AND	And	4.3.4
ARPL	Adjust Priviledge level	4.3.10
BOUND	Check Array Index Against Bounds	4.3.9
CALL	Call (intra segment)	4.3.6
CALLF	Call (inter segment)	4.3.6
CBW	Convert Byte to Word	4.3.4
CLC	Clear Carry	4.3.7
CLD	Clear Direction	4.3.7
CLI	Clear Interrupt	4.3.7
CMC	Complement Carry	4.3.7
CMP	Compare	4.3.4
CMPS	Compare Byte or Word (of string)	4.3.5
CMPSB	Compare Byte (of string)	4.3.5
CMPSW	Compare Word (of string)	4.3.5
CTS	Clear Task Switched Flag	4.3.10
CWD	Convert Word to Double Word	4.3.4
DAA	Decimal Adjust for Addition	4.3.4
DAS	Decimal Adjust for Subtraction	4.3.4
DEC	Decrement	4.3.4
DIV	Divide	4.3.4
ENTER	Procedure Entry	4.3.9
ESC	Escape	4.3.7
F2XM1	2^x-1	4.3.8
FABS	Absolute Value	4.3.8
FADD	Add Real	4.3.8
FADD32	Add Real, 32-bit	4.3.8
FADD64	Add Real, 64-bit	4.3.8

Table 4-1. (Continued)

Mnemonic	Description	Section
FADDP	Add Real and Pop	4.3.8
FBLD	Packed Decimal Load	4.3.8
FBSTP	Packed Decimal Store and Pop	4.3.8
FCHS	Change Sign	4.3.8
FCLEX/FNCLEX	Clear Exceptions	4.3.8
FCOM	Compare Real	4.3.8
FCOM32	Compare Real, 32-bit	4.3.8
FCOM64	Compare Real, 64-bit	4.3.8
FCOMP	Compare Real and Pop	4.3.8
FCOM32P	Compare Real and Pop, 32-bit	4.3.8
FCOM64P	Compare Real and Pop, 64-bit	4.3.8
FCOMPP	Compare Real and Pop Twice	4.3.8
FDECSTP	Decrement Stack Pointer	4.3.8
FDISI/FNDISI	Disable Interrupts	4.3.8
FDIV	Divide Real	4.3.8
FDIV32	Divide Real, 32-bit	4.3.8
FDIV64	Divide Real, 64-bit	4.3.8
FDIVR	Divide Real Reversed	4.3.8
FDIVR32	Divide Real Reversed, 32-bit	4.3.8
FDIVR64	Divide Real Reversed, 64-bit	4.3.8
FDIVRP	Divide Real Reversed and Pop	4.3.8
FDUP	Duplicate Top of Stack	4.3.8
FENI/FNENI	Enable Interrupts	4.3.8
FFREE	Free Register	4.3.8
FIADD16	Integer Add, 16-bit	4.3.8
FIADD32	Integer Add, 32-bit	4.3.8
FICOM16	Integer Compare, 16-bit	4.3.8
FICOM32	Integer Compare, 32-bit	4.3.8
FICOM16P	Integer Compare and Pop, 16-bit	4.3.8
FICOM32P	Integer Compare and Pop, 32-bit	4.3.8
FIDIV16	Integer Divide, 16-bit	4.3.8
FIDIV32	Integer Divide, 32-bit	4.3.8
FIDIVR16	Integer Divide Reversed, 16-bit	4.3.8

Table 4-1. (Continued)

Mnemonic	Description	Section
FIDIVR32	Integer Divide Reversed, 32-bit	4.3.8
FILD16	Integer Load, 16-bit	4.3.8
FILD32	Integer Load, 32-bit	4.3.8
FILD64	Integer Load, 64-bit	4.3.8
FIMUL16	Integer Multiply, 16-bit	4.3.8
FIMUL32	Integer Multiply, 32-bit	4.3.8
FINCSTP	Increment Stack Pointer	4.3.8
FINIT/FNINIT	Initialize Processor	4.3.8
FIST16	Integer Store, 16-bit	4.3.8
FIST32	Integer Store, 32-bit	4.3.8
FIST16P	Interger Store and Pop, 16-bit	4.3.8
FIST32P	Interger Store and Pop, 32-bit	4.3.8
FIST64P	Interger Store and Pop, 64-bit	4.3.8
FISUB16	Integer Subtract, 16-bit	4.3.8
FISUB32	Integer Subtract, 32-bit	4.3.8
FISUBR16	Integer Subtract Reversed, 16-bit	4.3.8
FISUBR32	Integer Subtract Reversed, 32-bit	4.3.8
FLD	Load Real	4.3.8
FLD32	Load Real, 32-bit	4.3.8
FLD64	Load Real, 64-bit	4.3.8
FLD80	Load Real, 80-bit	4.3.8
FLDCW	Load Control Word	4.3.8
FLDENV	Load Environment	4.3.8
FLDZ	Load + 0.0	4.3.8
FLD1	Load + 1.0	4.3.8
FLDPI	Load 80-bit value for pi.	4.3.8
FLDL2T	Load $\log_2 10$	4.3.8
FLDL2E	Load $\log_2 e$	4.3.8
FLDLG2	Load $\log_{10} 2$	4.3.8
FLDLN2	Load $\log_e 2$	4.3.8
FMUL	Multiply Real	4.3.8
FMUL32	Multiply Real, 32-bit	4.3.8
FMUL64	Multiply Real, 64-bit	4.3.8

Table 4-1. (Continued)

Mnemonic	Description	Section
FMULP	Multiply Real and Pop	4.3.8
FNOP	No Operation	4.3.8
FPATAN	Partial Arctangent	4.3.8
FPOP	same as FSTP ST0	4.3.8
FPREM	Partial Remainder	4.3.8
FPTAN	Partial Tangent	4.3.8
FRNDINT	Round to Integer	4.3.8
FRSTOR	Restore State	4.3.8
FSAVE/FNSAVE	Save State	4.3.8
FSCALE	Scale	4.3.8
FST	Store Real	4.3.8
FST32	Store Real, 32-bit	4.3.8
FST64	Store Real, 64-bit	4.3.8
FSTP	Store Real and Pop	4.3.8
FST32P	Store Real and Pop, 32-bit	4.3.8
FST64P	Store Real and Pop, 64-bit	4.3.8
FSTENV/FNSTENV	Store Environment	4.3.8
FSTCW/FNSTCW	Store Control Word	4.3.8
FSTSW/FNSTSW	Store Status Word	4.3.8
FSQRT	Square Root	4.3.8
FSUB	Subtract Real	4.3.8
FSUB32	Subtract Real, 32-bit	4.3.8
FSUB64	Subtract Real, 64-bit	4.3.8
FSUBP	Subtract Real and Pop	4.3.8
FSUBR	Subtract Real Reversed	4.3.8
FSUBR32	Subtract Real Reversed, 32-bit	4.3.8
FSUBR64	Subtract Real Reversed, 64-bit	4.3.8
FSUBRP	Subtract Real Reversed and Pop	4.3.8
FTST	Test	4.3.8
FWAIT	CPU Wait	4.3.8
FXAM	Examine	4.3.8
FXCH	Exchange Registers	4.3.8
FXCHG	same as FXCH ST1	4.3.8

Table 4-1. (Continued)

Mnemonic	Description	Section
FXTRACT	Extract Exponent and Significand	4.3.8
FYL2X	Y * $\log_2 X$	4.3.8
FYL2XP1	Y * $\log_2(X + 1)$	4.3.8
HLT	Halt	4.3.7
IDIV	Integer Divide	4.3.4
IMUL	Integer Multiply	4.3.4
IN	Input Byte or Word	4.3.3
INC	Increment	4.3.4
INSB	Input Byte from Port to String	4.3.9
INSW	Input Word from Port to String	4.3.9
INT	Interrupt	4.3.6
INTO	Interrupt on Overflow	4.3.6
IRET	Interrupt Return	4.3.6
JA	Jump on Above	4.3.6
JAE	Jump on Above or Equal	4.3.6
JB	Jump on Below	4.3.6
JBE	Jump on Below or Equal	4.3.6
JC	Jump on Carry	4.3.6
JCXZ	Jump on CX Zero	4.3.6
JE	Jump on Equal	4.3.6
JG	Jump on Greater	4.3.6
JGE	Jump on Greater or Equal	4.3.6
JL	Jump on Less	4.3.6
JLE	Jump on Less or Equal	4.3.6
JMP	Jump (intra segment)	4.3.6
JMPF	Jump (inter segment)	4.3.6
JMPS	Jump (8 bit dispiacement)	4.3.6
JNA	Jump on Not Above	4.3.6
JNAE	Jump on Not Above or Equal	4.3.6
JNB	Jump on Not Below	4.3.6
JNBE	Jump on Not Below or Equal	4.3.6
JNC	Jump on Not Carry	4.3.6
JNE	Jump on Not Equal	4.3.6

Table 4-1. (Continued)

Mnemonic	Description	Section
JNG	Jump on Not Greater	4.3.6
JNGE	Jump on Not Greater or Equal	4.3.6
JNL	Jump on Not Less	4.3.6
JNLE	Jump on Not Less or Equal	4.3.6
JNO	Jump on Not Overflow	4.3.6
JNP	Jump on Not Parity	4.3.6
JNS	Jump on Not Sign	4.3.6
JNZ	Jump on Not Zero	4.3.6
JO	Jump on Overflow	4.3.6
JP	Jump on Parity	4.3.6
JPE	Jump on Parity Even	4.3.6
JPO	Jump on Parity Odd	4.3.6
JS	Jump on Sign	4.3.6
JZ	Jump on Zero	4.3.6
LAHF	Load AH with Flags	4.3.3
LAR	Load Access Rights	4.3.10
LDS	Load Pointer into DS	4.3.3
LEA	Load Effective Address	4.3.3
LEAVE	High Level Procedure Exit	4.3.9
LES	Load Pointer into ES	4.3.3
LGDT	Load Global Descriptor Table Register	4.3.10
LIDT	Load Interrupt Descriptor Table Register	4.3.10
LLDT	Load Local Descriptor Table Register	4.3.10
LMSW	Load Machine Status Word	4.3.10
LOCK	Lock Bus	4.3.7
LODS	Load Byte or Word (of string)	4.3.5
LODSB	Load Byte (of string)	4.3.5
LODSW	Load Word (of string)	4.3.5
LOOP	Loop	4.3.6
LOOPE	Loop While Equal	4.3.6
LOOPNE	Loop While Not Equal	4.3.6
LOOPNZ	Loop While Not Zero	4.3.6
LOOPZ	Loop While Zero	4.3.6

Table 4-1. (Continued)

Mnemonic	Description	Section
LSL	Load Segment Limit	4.3.10
LTR	Load Task Register	4.3.10
MOV	Move	4.3.3
MOVS	Move Byte or Word (of string)	4.3.5
MOVSB	Move Byte (of string)	4.3.5
MOVSW	Move Word (of string)	4.3.5
MUL	Multiply	4.3.4
NEG	Negate	4.3.4
NOP	No Operation	4.3.7
NOT	Not	4.3.4
OR	Or	4.3.4
OUT	Output Byte or Word	4.3.3
OUTSB	Output Byte Pointer [si] to DX	4.3.9
OUTSW	Output Word Pointer [si] to DX	4.3.9
POP	Pop	4.3.3
POPA	Pop all General Registers	4.3.9
POPF	Pop Flags	4.3.3
PUSH	Push	4.3.3
PUSHA	Push all General Registers	4.3.9
PUSHF	Push Flags	4.3.3
RCL	Rotate through Carry Left	4.3.4
RCR	Rotate through Carry Right	4.3.4
REP	Repeat	4.3.5
REPE	Repeat While Equal	4.3.5
REPNE	Repeat While Not Equal	4.3.5
REPNZ	Repeat While Not Zero	4.3.5
REPZ	Repeat While Zero	4.3.5
RET	Return (intra segment)	4.3.6
RETF	Return (inter segment)	4.3.6
ROL	Rotate Left	4.3.4
ROR	Rotate Right	4.3.4
SAHF	Store AH into Flags	4.3.3
SAL	Shift Arithmetic Left	4.3.4

Table 4-1. (Continued)

Mnemonic	Description	Section
SAR	Shift Arithmetic Right	4.3.4
SBB	Subtract with Borrow	4.3.4
SCAS	Scan Byte or Word (of string)	4.3.5
SCASB	Scan Byte (of string)	4.3.5
SCASW	Scan Word (of string)	4.3.5
SGDT	Store Global Descriptor Table Register	4.3.10
SHL	Shift Left	4.3.4
SHR	Shift Right	4.3.4
SIDT	Store Interrupt Descriptor Table Register	4.3.10
SLDT	Store Local Descriptor Table Register	4.3.10
SMSW	Store Machine Status Word	4.3.10
STC	Set Carry	4.3.7
STD	Set Direction	4.3.7
STI	Set Interrupt	4.3.7
STOS	Store Byte or Word (of string)	4.3.5
STOSB	Store Byte (of string)	4.3.5
STOSW	Store Word (of string)	4.3.5
STR	Store Task Register	4.3.10
SUB	Subtract	4.3.4
TEST	Test	4.3.4
VERR	Verify Read Access	4.3.10
VERW	Verify Write Access	4.3.10
WAIT	Wait	4.3.7
XCHG	Exchange	4.3.3
XLAT	Translate	4.3.3
XOR	Exclusive Or	4.3.4

4.3 Instruction-definition Tables

4.3.1 Symbol Conventions

The instruction-definition tables organize RASM-86 instructions into functional groups. In each table, the instructions are listed alphabetically. Table 4-2 shows the symbols used in the instruction-definition tables to define operand types.

Table 4-2. Operand Type Symbols

Symbol	Operand Type
numb	any numeric expression
numb8	any numeric expression that evaluates to an 8-bit number
acc	accumulator register, AX or AL
reg	any general purpose register not a segment register
reg16	a 16-bit general purpose register not a segment register
segreg	any segment register: CS, DS, SS, or ES

Table 4-2. (Continued)

Symbol	Operand Type
mem	any address expression with or without base- and/or index-addressing modes, such as the following: 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 the following: 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 within +/- 128 bytes distance from the instruction

4.3.2 Flag Registers

The 8086 and 80286 CPUs have nine single-bit Flag registers that can be displayed to reflect the state of the processor. You cannot access these registers directly, but you 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 4-3 to represent the nine Flag registers.

Table 4-3. 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

4.3.3 8086 Data Transfer Instructions

There are four classes of data transfer operations:

- general purpose
- accumulator specific
- address-object
- flag

Only SAHF and POPF affect flag settings. Note in Table 4-4 that if acc = AL, a byte is transferred, but if acc = AX, a word is transferred.

Table 4-4. 8086 Data Transfer Instructions

	Syntax	Result
IN	acc,numb8	transfer data from input port given by numb8 (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
MOV	reg,mem|reg	move memory or register to register
MOV	mem|reg,reg	move register to memory or register

Table 4-4. (Continued)

	Syntax	Result
MOV	mem|reg,numb	move immediate data to memory or register
MOV	segreg,mem|reg16	move memory or register to segment register
MOV	mem|reg16,segreg	move segment register to memory or register
OUT	numb8,acc	transfer data from accumulator to output port (0-255) given by numb8
OUT	DX,acc	transfer data from accumulator to output port (0-0FFFFH) given by DX register
POP	mem|reg16	move top stack element to memory or register
POP	segreg	move top stack element to segment register; note that CS segment register is not allowed
POPF		transfer top stack element to flags
PUSH	mem|reg16	move memory or register to top stack element
PUSH	segreg	move segment register to top stack element

Table 4-4. (Continued)

Syntax		Result
PUSHF		transfer flags to top stack element
SAHF		transfer the AH register to flags
XCHG	reg,mem|reg	exchange register and memory or register
XCHG	mem|reg,reg	exchange memory or register and register
XLAT	mem|reg	perform table lookup translation, table given by mem|reg, which is always BX. Replaces AL with AL offset from BX

4.3.4 8086 Arithmetic, Logical, and Shift Instructions

The 8086 and 80286 CPUs perform addition, subtraction, multiplication, and division in several ways. Both CPUs support 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 4-5 summarizes the effects of arithmetic instructions on flag bits. Table 4-6 defines arithmetic instructions. Table 4-7 defines logical and shift instructions.

Table 4-5. Effects of Arithmetic Instructions on Flags

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

Table 4-6. 8086 Arithmetic Instructions

Syntax		Result
AAA		adjust unpacked BCD (ASCII) for addition - adjusts AL
AAD		adjust unpacked BCD (ASCII) for division - adjusts AL
AAM		adjust unpacked BCD (ASCII) for multiplication - adjusts AX
AAS		adjust unpacked BCD (ASCII) for subtraction - adjusts AL
ADC	reg,mem|reg	add (with carry) memory or register to register
ADC	mem|reg,reg	add (with carry) register to memory or register
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 AX by sign extension

Table 4-6. (Continued)

Syntax		Result
CMP	reg,mem|reg	compare memory or register with register
CMP	mem|reg,reg	compare register with memory or register
CMP	mem|reg,numb	compare data constant with memory or register
CWD		convert word in AX to double word in DX/AX by sign extension
DAA		decimal adjust for addition, adjusts AL
DAS		decimal adjust for subtraction, adjusts AL
DEC	mem|reg	subtract 1 from memory or register
DIV	mem|reg	divide (unsigned) accumulator (AX or AL) by memory or register. If byte results, AL = quotient, AH = remainder. If word results, AX = quotient, DX = remainder
IDIV	mem|reg	divide (signed) accumulator (AX or AL) by memory or register - quotient and remainder stored as in DIV

Table 4-6. (Continued)

Syntax		Result
IMUL	mem|reg	multiply (signed) memory or register by accumulator (AX or AL). If byte, results in AH, AL. If word, results in DX, AX.
INC	mem|reg	add 1 to memory or register
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 4-7. 8086 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 or 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

Table 4-7. (Continued)

	Syntax	Result
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 bit
SAL	mem|reg,CL	shift memory or register left, number of bits given by CL register, shift in low-order zero bits

Table 4-7. (Continued)

	Syntax	Result
SAR	mem|reg,1	shift memory or register 1 bit right, shift in high-order bit 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 bit. 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 bit
SHR	mem|reg,CL	shift memory or register right, number of bits given by CL register, shift in high-order zero bits

Table 4-7. (Continued)

Syntax	Result
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 or register and register - set condition flags, but do not change destination.
TEST mem|reg,numb	perform bitwise logical AND of memory or 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
XOR mem|reg,reg	perform bitwise logical exclusive OR of memory or register and register
XOR mem|reg,numb	perform bitwise logical exclusive OR of memory or register and data constant

4.3.5 8086 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 reside in the Extra Segment (ES) and source operands addressed by SI must reside in the Data Segment (DS).

The source operand is normally addressed by the DS register. However, you can designate a different register by using a segment override. For example,

```
MOVS    WORD PTR[DI], CS:WORD PTR[SI]
```

writes the contents of the address at CS:[SI] into ES:[DI].

Table 4-8. 8086 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 that assumes a byte operand
CMPSW	an alternate mnemonic for CMPS that 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 that assumes a byte operand
LODSW	an alternate mnemonic for LODS that assumes a word operand
MOVS mem|reg,mem|reg	move 1 byte (or word) from source to destination
MOVSB	an alternate mnemonic for MOVS that assumes a byte operand
MOVSW	an alternate mnemonic for MOVS that assumes a word operand

Table 4-8. (Continued)

Syntax	Result
SCAS mem|reg	subtract destination operand from accumulator (AX or AL), affect flags, but do not return result
SCASB	an alternate mnemonic for SCAS that assumes a byte operand
SCASW	an alternate mnemonic for SCAS that assumes a word operand
STOS mem|reg	transfer a byte or word from accumulator to the destination operand
STOSB	an alternate mnemonic for STOS that assumes a byte operand
STOSW	an alternate mnemonic for STOS that assumes a word operand

Table 4-9 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 4-9. 8086 Prefix Instructions

Syntax	Result
REP	repeat until CX register is zero
REPE	repeat until CX register is zero, or zero flag (ZF) is not zero
REPNE	repeat until CX register is zero, or zero flag (ZF) is zero
REPNZ	equal to REPNE
REPZ	equal to REPE

4.3.6 8086 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 can depend upon a certain condition. Table 4-10 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 4-10. 8086 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), jump to the target label
CALLF mem	push CS register on the stack, push the offset address of the next instruction on the stack, jump to location indicated by contents of specified double word in memory
INT numb8	push the flag registers (as in PUSHF), clear TF and IF flags, transfer control with an indirect call through any one of the 256 interrupt-vector elements - uses three levels of stack

Table 4-10. (Continued)

Syntax		Result
INTO		if OF (the overflow flag) is set, push the flag registers (as in PUSHF), clear TF and IF flags, transfer control with an indirect call through interrupt-vector element 4 (location 10H). If the OF flag is cleared, no operation takes place
IRET		transfer control to the return address saved by a previous interrupt operation, restore saved flag registers, 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	lab8T	jump if "below or equal" or "not above" ((CF or ZF)=1)
JC	lab8	same as JB
JCXZ	lab8	jump to target label if CX register is zero
JE	lab8	jump if "equal" or "zero" (ZF=1)

Table 4-10. (Continued)

Syntax		Result
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)

Table 4-10. (Continued)

Syntax	Result
JNG lab8	same as JLE
JNGE lab8	same as JL
JNL lab8	same as JGE
JNLE lab8	same as JG
JNO lab8	jump if "not overflow" (OF=0)
JNP lab8	jump if "not parity" or "parity odd" (PF=0)
JNS lab8	jump if "not sign" (SF=0)
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"

Table 4-10. (Continued)

Syntax	Result
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 address pushed by a previous CALL instruction, increment stack pointer by 2
RET numb	return to the address pushed by a previous CALL, increment stack pointer by 2+numb
RETF	return to the address pushed by a previous CALLF instruction, increment stack pointer by 4
RETF numb	return to the address pushed by a previous CALLF instruction, increment stack pointer by 4+numb

4.3.7 8086 Processor Control Instructions

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

Table 4-11. 8086 Processor Control Instructions

Syntax		Result
CLC		clear CF flag
CLD		clear DF flag, causing string instructions to auto-increment the operand registers
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		cause 8086 processor to enter halt state until an interrupt is recognized
LOCK		PREFIX instruction, cause the 8086 processor to assert the bus-lock signal for the duration of the operation caused by the following instruction. The LOCK prefix instruction can precede any other instruction. Bus_lock prevents coprocessors from gaining the bus; this is useful for shared-resource semaphores

Table 4-11. (Continued)

Syntax	Result
NOP	no operation is performed
STC	set CF flag
STD	set DF flag, causing string instructions to auto-decrement the operand registers
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

4.3.8 8087 Instruction Set

RASM-86 supports 8087 opcodes. However, RASM-86 only allows 8087 opcodes in byte, word, and double word format. The form of the RASM-86 instructions differ slightly from the Intel convention to support 8087 instructions.

All 8087 memory reference instructons have two characters appended to the end of the opcode name. The two characters represent the number of bits referenced by the instruction. For example:

```
FADD64  byte ptr  my_var
```

This instruction assumes MY_VAR contains 64 bits (8 bytes). This convention applies to all 8087 instructions referencing user memory, except those that always reference the same number of bits, as is the FSTCW instruction, for example.

Another difference between RASM-86 and the standard Intel convention is that the number of bits referenced by the instruction is placed before the "P" on instructions in which the stack is to be popped. For example:

```
FSUB80P  byte ptr  my_var;  sub  and  pop  temp  real
```

Many of the following 8087 operations are described in terms of the stack registers: ST0, ST1, ... STi (where "i" represents any register on the stack). The stack register where the resulting value is stored is also described for many operations. It is important to remember that when a POP occurs at the end of an 8087 operation, the stack register containing the value is decremented by 1.

For example, if, during an 8087 operation, the result is put in ST3 and a POP occurs at the end of the operation, the result ends up in ST2.

Table 4-12. 8087 Data Transfer Instructions

Syntax	Result
	Real Transfers
FLD	Load a number in IEEE floating point format into 8087 top stack element ST0
FLD32	Load a number in IEEE 32-bit floating point format into 8087 top stack element ST0
FLD64	Load a number in IEEE 64-bit floating point format into 8087 top stack element ST0
FLD80	Load a number in IEEE 80-bit floating point format into 8087 top stack element ST0
FDUP	Duplicate top of stack (FLD ST0)
FST	Store Real
FST32	Store Real (32-bit operands)
FST64	Store Real (64-bit operands)
FSTP	Store Real and Pop
FST32P	Store Real and Pop (32-bit operands)
FST64P	Store Real and Pop (64-bit operands)
FPOP	same as FSTP ST0
FXCH	Exchange Registers
FXCHG	same as FXCH ST1

Table 4-12. (Continued)

Syntax	Result
	Integer Transfers
FILD16	Integer Load (16-bit operands)
FILD32	Integer Load (32-bit operands)
FILD64	Integer Load (64-bit operands)
FIST16	Integer Store (16-bit operands)
FIST32	Integer Store (32-bit operands)
FIST16P	Interger Store and Pop (16-bit operands)
FIST32P	Interger Store and Pop (32-bit operands)
FIST64P	Interger Store and Pop (64-bit operands)
	Packed Decimal Transfers
FBLD	Packed Decimal (BCD) Load
FBSTP	Packed Decimal (BCD) Store 10 bytes and Pop

Table 4-13. 8087 Arithmetic Instructions

Syntax	Operands	Result
		Addition
FADD		Add Real ST0 to ST1, store result in ST1 and Pop
FADD	STi,ST0	Add Real ST0 to STi, store result in STi
FADD32	mem	Add Real mem to ST0, store result in ST0 (32-bit operands)
FADD64	mem	Add Real mem to ST0, store result in ST0 (64-bit operands)
FADDP	STi,ST0	Add Real ST0 to STi, store result in STi and Pop
FIADD16	mem	Integer Add mem to ST0, store result in ST0 (16 bit-operands)
FIADD32	mem	Integer Add mem to ST0, store result in ST0 (32 bit-operands)
		Subtraction
FSUB		Subtract Real ST0 from ST1, store result in ST1 and Pop
FSUB	STi,ST0	Subtract Real ST0 from STi, store result in STi
FSUB	ST0,STi	Subtract Real STi from ST0, store result in ST0

Table 4-13. (Continued)

Syntax	Operands	Result
FSUB32	mem	Subtract Real mem from ST0, store result in ST0 (32-bit operands)
FSUB64	mem	Subtract Real mem from ST0, store result in ST0 (64-bit operands)
FSUBP	STi,ST0	Subtract Real ST0 from STi, store result in STi and Pop
FISUB16	mem	Integer Subtract mem from ST0, store result in ST0 (16-bit operands)
FISUB32	mem	Integer Subtract mem from ST0, store result in ST0 (32-bit operands)
FSUBR		Subtract Real ST1 from ST0, store result in ST1 and Pop
FSUBR	STi,ST0	Subtract Real STi from ST0, store result in STi
FSUBR	ST0,STi	Subtract Real ST0 from STi, store result in ST0
FSUBR32	mem	Subtract Real mem from ST0, store result in ST0 (32-bit operands)
FSUBR64	mem	Subtract Real mem from ST0, store result in ST0 (64-bit operands)

Table 4-13. (Continued)

Syntax	Operands	Result
FSUBRP	STi,ST0	Subtract Real STi from ST0, store result in STi and Pop
FISUBR16	mem	Integer Subtract ST0 from mem, store result in ST0 (16-bit operands)
FISUBR32	mem	Integer Subtract ST0 from mem, store result in ST0 (32-bit operands)
	Multiplication	
FMUL		Multiply Real ST1 by ST0, store result in ST1 and Pop
FMUL	STi,ST0	Multiply Real STi by ST0, store result in STi
FMUL	ST0,STi	Multiply Real ST0 by STi, store result in ST0
FMUL32	mem	Multiply Real ST0 by mem, store result in ST0 (32-bit operands)
FMUL64	mem	Multiply Real ST0 by mem, store result in ST0 (64-bit operands)
FMULP	STi,ST0	Multiply Real STi by ST0, store result in STi and Pop
FIMUL16	mem	Integer Multiply ST0 by mem, store result in ST0 (16-bit operands)

Table 4-13. (Continued)

Syntax	Operands	Result
FIMUL32	mem	Integer Multiply ST0 by mem, store result in ST0 (32-bit operands)
		Division
FDIV		Divide Real ST1 by ST0, store result in ST1 and Pop
FDIV	STi,ST0	Divide Real STi by ST0, store result in STi
FDIV	ST0,STi	Divide Real ST0 by STi, store result in ST0
FDIV32	mem	Divide Real ST0 by mem, store result in ST0 (32-bit operands)
FDIV64	mem	Divide Real ST0 by mem, store result in ST0 (64-bit operands)
FDIVP	STi,ST0	Divide Real STi by ST0, store result in STi and Pop
FIDIV16	mem	Integer Divide ST0 by mem, store result in ST0 (16-bit operands)
FIDIV32	mem	Integer Divide ST0 by mem, store result in ST0 (32-bit operands)
FDIVR		Divide Real ST0 by ST1, store result in ST1 and Pop
FDIVR	STi,ST0	Divide Real ST0 by STi, store result in STi
FDIVR	ST0,STi	Divide Real STi by ST0, store result in ST0

Table 4-13. (Continued)

Syntax	Operands	Result
FDIVR32	mem	Divide Real mem by ST0, store result in ST0 (32-bit operands)
FDIVR64	mem	Divide Real mem by ST0, store result in ST0 (64-bit operands)
FDIVRP	STi,ST0	Divide Real STi by ST0, store result in STi and Pop
FIDIVR16	mem	Integer Divide mem by ST0, store result in ST0 (16-bit operands)
FIDIVR32	mem	Integer Divide mem by ST0, store result in ST0 (32-bit operands)
	Other Operations	
FSQRT		Square Root
FSCALE		Interpret ST1 as an integer and add to exponent of ST0
FPREM		Partial Remainder
FRNDINT		Round to Integer
FXTRACT		Extract Exponent and Significand
FABS		Absolute Value
FCHS		Change Sign

Table 4-14. 8087 Comparison Instructions

Syntax	Operands	Result
FCOM		Compare Real ST0 and ST1
FCOM32	mem	Compare Real mem and ST0 (32-bit operands)
FCOM64	mem	Compare Real mem and ST0 (64-bit operands)
FCOMP		Compare Real ST0 and ST1 and Pop
FCOM32P	mem	Compare Real mem and ST0 and Pop (32-bit operands)
FCOM64P	mem	Compare Real mem and ST0 and Pop (64-bit operands)
FCOMPP		Compare Real ST0 and ST1, then Pop ST0 and ST1
FICOM16	mem	Integer Compare mem and ST0 (16-bit operands)
FICOM32	mem	Integer Compare mem and ST0 (32-bit operands)
FICOM16P	mem	Integer Compare mem and ST0 and Pop (16-bit operands)
FICOM32P	mem	Integer Compare mem and ST0 and Pop (32-bit operands)
FTST		Test ST0 by comparing it to zero
FXAM		Report ST0 as either positive or negative

Table 4-15. 8087 Transcendental Instructions

Syntax	Result
FPTAN	Partial Tangent
FPATAN	Partial Arctangent
F2XM1	2^X-1
FYL2X	$Y * \log_2 X$
FYL2XP1	$Y * \log_2(X + 1)$

Table 4-16. 8087 Constant Instructions

Syntax	Result
FLDZ	Load + 0.0
FLD1	Load + 1.0
FLDPI	Load 80-bit value for pi.
FLDL2T	Load $\log_2 10$
FLDL2E	Load $\log_2 e$
FLDLG2	Load $\log_{10} 2$
FLDLN2	Load $\log_e 2$

Table 4-17. 8087 Processor Control Instructions

Syntax	Operands	Result
FINIT/FNINIT		Initialize Processor
FDISI/FNDISI		Disable Interrupts
FENI/FNENI		Enable Interrupts
FLDCW	mem	Load Control Word
FSTCW/FNSTCW	mem	Store Control Word
FSTSW/FNSTSW	mem	Store Status Word
FCLEX/FNCLEX		Clear Exceptions
FSTENV/FNSTENV	mem	Store Environment
FLDENV	mem	Load Environment
FSAVE/FNSAVE		Save State
FRSTOR		Restore State
FINCSTP		Increment Stack Pointer
FDECSTP		Decrement Stack Pointer
FFREE		Free Register
FNOP		No Operation
FWAIT		CPU Wait

4.3.9 Additional 186 and 286 Instructions

The following Instructions are specific to both the 80186 and 80286 processors. In addition to the instructions below, other 80186 and 80286 instructions are the same as 8086 instructions except they allow a rotate or shift. These instructions are: SAR, SAL, SHR, SHL, ROR, and ROL.

Table 4-18. Additional 186 and 286 Instructions

Syntax	Result
BOUND	Check Array Index Against Bounds
ENTER	Make Stack Frame for Procedure Parameters
INSB	Input Byte from Port to String
INSW	Input Word from Port to String
LEAVE	High Level Procedure Exit
OUTSB	Output Byte Pointer [si] to DX
OUTSW	Output Word Pointer [si] to DX
POPA	Pop all General Registers
PUSHA	Push all General Registers

4.3.10 Additional 286 Instructions

The following instructions are specific to the 80286 processor.

Table 4-19. Additional 286 Instructions

Syntax	Result
CTS	Clear Task Switched Flag
ARPL	Adjust Priviledge level
LGDT	Load Global Descriptor Table Register

Table 4-19. (Continued)

Syntax	Result
SGDT	Store Global Descriptor Table Register
LIDT	Load Interrupt Descriptor Table Register
SIDT	Store Interrupt Descriptor Table Register
LLDT	Load Local Descriptor Table Register from Register/Memory
SLDT	Store Local Descriptor Table Register to Register/Memory
LTR	Load Task Register from Register/Memory
STR	Store Task Register to Register/Memory
LMSW	Load Machine Status Word from Register/Memory
SMSW	Store Machine Status Word
LAR	Load Access Rights from Register/Memory
LSL	Load Segment Limit from Register/Memory
ARPL	Adjust Required Privilege Level from Register/Memory
VERR	Verify Read Access; Register/Memory
VERW	Verify Write Access

End of Section 4

RASM-86 Code-Macro Facilities

5.1 Introduction

RASM-86 allows you to define your own instructions using the Code-macro directive. RASM-86 code-macros differ from traditional assembly-language macros in the following ways:

- Traditional assembly-language macros contain assembly-language instructions, but a RASM-86 code-macro contains only code-macro directives.
- Traditional assembly-language macros are usually defined in the Symbol Table, while RASM-86 code-macros are defined in the assembler's internal Symbol Table.
- A traditional macro simplifies the repeated use of the same block of instructions throughout a program, but a code-macro sends a bit stream to the output file, and in effect, adds a new instruction to the assembler.

5.2 Invoking Code-macros

RASM-86 treats a code-macro as an instruction, so you can invoke code-macros by using them as instructions in your program. The following example shows how to invoke MYCODE, an instruction defined by a code-macro.

```
MYCODE  PARM1,PARM2
```

Note that MYCODE accepts two operands as formal parameters. When you define MYCODE, RASM-86 classifies these two operands according to type, size, and so forth.

5.3 Defining Code-macros

A code-macro definition takes the general form:

```
CodeMacro name  [ formal parameter list ]
[ list of code-macro directives ]
EndM
```

where name is any string of characters you select to represent the code-macro. The optional formal parameter and code-macro directive lists are described in the following sections. Example code-macro definitions are provided in Section 5.3.3

5.3.1 Formal Parameter List

When you define a code macro, you can specify one or more optional formal parameter lists. The parameters specified in the formal parameter list are used as placeholders to indicate where and how the operands are to be used. The formal parameter list is created using the following syntax:

```
formal_name : specifier_letter [ modifier_letter ] [ range ]
```

formal_name

You can specify any formal_name to represent the formal parameters in your list. RASM-86 replaces the formal_names with the names or values supplied as operands when you invoke the code-macro.

specifier_letter

Every formal parameter must have a specifier letter to indicate what type of operand is 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 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	General register only.
S	Segment register only.
X	Direct memory reference.

modifier_letter

The optional modifier_letter in a code-macro definition 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 be a certain type:

- b for byte
- w for word
- d for double-word
- sb for signed byte

For numbers, the modifiers require the number be 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		

range

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

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

Numberb is an 8-bit number, not an address.

5.3.2 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, and those that appear to duplicate assembly language instructions have different meanings within a code-macro definition.

The following are legal code-macro directives:

SEGFIX	DW
NOSEGFIX	DD
MODRM	DBIT
RELB	IF
RELW	ELSE
DB	ENDIF

These directives are unique to code-macros. The code-macro directives DB, DW, and DD that appear to duplicate the RASM-86 directives of the same names have different meanings in code-macro context. These directives are discussed in greater detail in Section 3.7.1.

CodeMacro, EndM, and the code-macro directives are all reserved words. The formal definition syntax for a code-macro is defined in Backus-Naur-like form in Appendix D.

SEGFIX

SEGFIX instructs RASM-86 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, RASM-86 takes no action. SEGFIX has the following form:

```
SEGFIX formal_name
```

The formal_name is the name of a formal parameter representing the memory address. Because it represents a memory address, the formal parameter must have one of the specifiers E, M, or X.

NOSEGFIX

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

```
NOSEGFIX  segreg, form_name
```

The segreg is one of the segment registers ES, CS, SS, or DS, and form_name is the name of the memory-address formal parameter that 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 in a code-macro directive:

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

MODRM

This directive instructs RASM-86 to generate the MODRM byte following the opcode byte in many of the 8086 and 80286 instructions. The MODRM byte contains either the indexing type or the register number to be used in the instruction. It also specifies which register is to be used, or gives more information to specify an instruction.

The MODRM byte carries the information in three fields:

```
    fields:      mod     reg     reg_mem
MODRM byte:      _ _   _ _ _   _ _ _ _ _
```

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 **reg_mem**, or 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 earlier.

For further information about 8086 and 80286 instructions and their bit patterns, see the Intel assembly language programming manual and the Intel user's manual for your processor.

MODRM has the forms:

```
MODRM  form_name,form_name
MODRM  NUMBER7,form_name
```

NUMBER7 is a value 0 to 7 inclusive, and form_name is the name of a formal parameter. The following examples show how MODRM is used in a code-macro directive:

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

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

RELB and RELW

These directives, used in IP-relative branch instructions, instruct RASM-86 to generate a 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 have the following forms:

```
RELB form_name
RELW form_name
```

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

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

DB, DW and DD

These directives define a number, or a parameter as either a byte, word, or double-word. These directives differ from those occuring outside code-macros.

The directives have the following forms:

```
DB form_name | NUMBERB
DW form_name | NUMBERW
DD form_name
```

NUMBERB is a single-byte number, NUMBERW is a two-byte number, and form_name is a name of a formal parameter. For example,

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

DBIT

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

```
DBIT field_description [,field_description]
```

The field_description has two forms:

```
number  combination
number (form_name (rshift))
```

The number ranges from 1 to 16, and specifies the number of bits to be set. The 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 contains form_name, a formal parameter name that instructs the assembler to put a certain number in the specified position. This number normally refers to the register specified in the first line of the code-macro. The numbers used in this special case for each register are the following:

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

The 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 is 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 is present in the definition

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

then the register number is shifted right once, and the erroneous result is 48H + 1H = 49H.

IF, ELSE, and ENDIF

The IF and ENDIF directives allow you to conditionally include or exclude a group of source lines from the assembly. The optional ELSE directive allows you to specify an alternative set of source lines. These code-macro directive operate in the same manner as the RASM-86 conditional assembly directives described in Section 3.5.1.

5.3.3 Example Code-Macro Definitions

In order to clearly distinguish specifiers from modifiers, the examples in this section show specifiers in uppercase and modifiers in lowercase.

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

Defines a code-macro, named IN, specifying that the input port must be identified by the DX register.

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

Defines a code-macro, named ROR, specifying that the CL register is to contain the count of rotation.

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

Defines a code macro named ESC, specifying that the value represented by the opcode parameter is to be immediate data, with a range from 0 to 63 bytes. ESC also specifies that the value represented by the adds parameter is a byte to be used as an effective address.

```
CodeMacro AAA
  DB 37H
EndM
```

Defines a code macro, named AAA, as the value 37H. (This is the normal opcode value of the AAA instruction.)

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

Defines a code macro, named NESC. The value represented by the opcode parameter is defined as data, with a range from 0 to 63 bytes. The value represented by the src parameter is defined as a byte to represent either a memory address or a register.

The SEGFIX directive checks to see if src is in the current segment (data segment) and, if not, to override with the correct segment.

The DBIT directive creates a byte. the upper five bits of this byte contain 1BH; the lower 3 bits are derived from the value of opcode, shifted right by 3.

The MODRM directive generates modrm bytes, based on the values of the opcode and src parameters.

End of Section 5

XREF-86 Cross-Reference Utility

6.1 Introduction

XREF-86 is an assembly language cross-reference utility program that creates a cross-reference file showing the use of symbols throughout the program. XREF-86 accepts two input files created by RASM-86. XREF-86 assumes these input files have filetypes of LST and SYM respectively, and they both reside on the same disk drive. XREF-86 creates one output file with the filetype XRF.

6.2 XREF-86 Command Syntax

XREF-86 is invoked using the command form:

XREF86 [drive:] filename

XREF-86 reads FILENAME.LST line by line, attaches a line number prefix to each line, and writes each prefixed line to the output file, FILENAME.XRF. During this process, XREF-86 scans each line for any symbols existing in the file FILENAME.SYM.

After completing this copy operation, XREF-86 appends to FILENAME.XRF a cross-reference report listing all the line numbers where each symbol in FILENAME.SYM appears. XREF-86 flags with a # character each line number reference where the referenced symbol is the first token on the line.

XREF-86 also lists the value of each symbol, as determined by RASM-86 and placed in the Symbol Table file, FILENAME.SYM.

When you invoke XREF-86, you can include an optional DRIVE: specification with the filename. When you invoke XREF-86 with a DRIVE: name preceding the FILENAME, XREF-86 searches for the input files and create the output file on the specified drive. If DRIVE: is not specified, XREF-86 associates the files with the default drive.

For example, to search for the file BIOS on the Drive C, enter:

```
xref86 c:bios
```

XREF-86 also allows you to direct the output file to the default list device instead of to FILENAME.XRF. To redirect the output, add the string $p to the command line. For example,

```
A>xref86 bios $p
```

End of Section 6

LINK-86 Linkage Editor

7.1 Introduction

LINK-86 is the linkage editor that combines relocatable object files to form either a native-mode (CMD) or PC DOS mode (EXE) command file that runs under Concurrent DOS 86. The object files can be produced by any 8086 or 80286 language translators that produce object files using a compatible subset of the Intel 8086/80286 object module format.

7.2 LINK-86 Operation

LINK-86 accepts three types of files.

Object (OBJ) File — A language source file processed by the language translator into the relocatable object code used by the microprocessor. This type of file contains one or more object modules.

Library (L86) File — An indexed library of commonly used object modules. A library file is generated by the library manager, LIB-86, in the processor's relocatable object format.

Input (INP) File — A file consisting of filenames and options like a command line entered from the console. For a detailed explanation of the input file, see Section 7.10.

LINK-86 produces the following types of files:

Command (CMD or EXE) File
Contains executable code loadable by Concurrent. The filetype depends on which version of the linker you use (native mode or PC DOS mode).

For simplicity, examples in this guide use the CMD filetype.

Symbol Table (SYM) File
: Contains a list of symbols from the object files and their offsets. This file is suitable for use with SID-86.

Line Number (LIN) File
: Contains line number symbols, which can be used by SID-86 for debugging. This file is created only if the compiler puts line number information into the object files being linked.

Map (MAP) File
: Contains segment information about the layout of the command file.

During processing, LINK-86 displays any unresolved symbols at the console. Unresolved symbols are symbols referenced but not defined in the files being linked. Such symbols must be resolved before the program can run.

Upon completion of processing, LINK-86 displays the size of each section of the command file and the Use Factor, which is a decimal percentage indicating the amount of available memory used by LINK-86.

See Section 7.12 for a complete explanation of the link process.

7.3 LINK-86 Command Syntax

You invoke LINK-86 with a command of the form:

```
LINK86 [filespec =] filespec_1 [,filespec_2,...,filespec_n]
```

where **filespec** is a Concurrent file specification, consisting of an optional drive specification and a filename with optional filetype.

Note: The PC DOS version does not support pathnames in the command line.

Each **filespec** can be followed by one or more of the command options described in Section 7.5. If you enter a filename to the left of the equal sign, LINK-86 creates the output files with that name and the appropriate filetypes. For example, if the files PARTA, PARTB, and PARTC are written in 8086 or 80286 assembly code, the command:

```
A>link86 myfile = parta,partb,partc
```

creates MYFILE.CMD and MYFILE.SYM. The files PARTA, PARTB, and PARTC can be a combination of object files and library files. If no filetype is specified, the linker assumes a filetype of OBJ.

If you do not specify an output filename, LINK-86 creates the output files using the first filename in the command line. For example, the command

```
A>link86 parta,partb,partc
```

creates the files PARTA.CMD and PARTA.SYM. If you specify a library file in your link command, do not enter the library file as the first file in the command line.

You can also instruct LINK-86 to read its command line from a file, thus making it possible to store long or commonly used link commands on disk (see Section 7.10).

The following are examples of LINK-86 commands:

```
A>link86 myfile = parta,partb

A>link86 a:myfile.286 = parta,partb,transvec

A>link86 b:myfile.cmd = parta,partb
```

The available LINK-86 command options are described in Section 7.5.

7.4 Stopping LINK-86

To stop LINK-86 during processing, press the console interrupt character, usually Control-C.

7.5 LINK-86 Command Options

When you invoke LINK-86, you can specify command options that control the link operation.

When specifying command options, enclose them in square brackets immediately following a filename. A command option is specified using the following command form:

```
A>link86 file[option]
```

For example, to specify the command option MAP for the file TEST1 and the NOLOCALS option for the file TEST2, enter:

```
A>link86 test1[map],test2[nolocals]
```

You can use spaces to improve the readability of the command line, and you can put more than one option in square brackets by separating them with commas. For example:

```
A>link86 test1 [map, nolocals], test2 [locals]
```

specifies that the MAP and NOLOCALS options be used for the TEST1 file and the option LOCALS for the TEST2 file.

LINK-86 command options are grouped into the following categories:

- Command File Options
- SYM File Options
- LIN File Options
- MAP File Options
- L86 File Options
- INPUT File Options
- I/O File Options

Table 7-1 summarizes the available LINK-86 command options. The following sections describe the function and syntax in detail for each command option.

Table 7-1. LINK-86 Command Options

Option	Abbreviation	Meaning
CODE	C	controls contents of CODE section of command file
DATA	D	controls contents of DATA section of command file
EXTRA	E	controls contents of EXTRA section of command file
STACK	ST	controls contents of STACK section of command file
FILL	F	zero fill and include uninitialized data in command file
NOFILL	NOF	do not include uninitialized data in command file
HARD8087	HA	create a command file requiring an 8087 coprocessor.
LIBSYMS	LI	include symbols from library files in SYM file
NOLIBSYMS	NOLI	do not include symbols from library files in SYM file
LOCALS	LO	include local symbols in SYM file
NOLOCALS	NOLO	do not include local symbols in SYM file
LINES	LIN	create LIN file with line number symbols

Table 7-1. (Continued)

Option	Abbreviation	Meaning
NOLINES	NOLIN	do not create LIN file
MAP	M	create a MAP file
SEARCH	S	search library and only link referenced modules
INPUT	I	read command line from disk file
ECHO	ECHO	echo contents of INP file on console
X1*	X1	controls contents of X1 section of CMD file
X1*	X1	controls contents of X1 section of CMD file
X1*	X1	controls contents of X1 section of CMD file
X1*	X1	controls contents of X1 section of CMD file
CODESHARED*	CODES	mark group as shared in CMD file header
NO PREFIX**	NOP	Do not generate prefix code at beginning of EXE file

* Native mode only
** PC DOS mode only

7.6 Command File Options

Most command file options can appear after any filename in the command line. The only exception is the HARD8087 option which if used, must appear after the first filename.

7.6.1 Command File Formats

A native-mode command file consists of a 128-byte header record followed by up to eight sections, each of which can be up to 64K in length. These sections are called CODE, DATA, STACK, EXTRA, X1, X2, X3, and X4. Each of these sections correspond to a LINK-86 command option of the same name. The header contains information such as the length of each section of the command file, its minimum and maximum memory requirements, and its load address. Concurrent uses this information to properly load the file.

A PC DOS mode command file contains two part: a **header** and an **executable** code module. The header contains 28 bytes of control information, if required. The executable code module begins on a sector boundary immediately following the header in the memory image created by LINK-86. PC DOS mode command files can contain up to four sections, each of which can be up to 64K in length. Each of these four sections (CODE, DATA, STACK, and EXTRA), correspond to a LINK-86 command option, which allows you to identify a section in a command file. The parameters described below allow you to alter the information in that section.

File Section Option Parameters

Each of the options identifying the command file sections must be followed by one or more parameters enclosed in square brackets.

LINK-86 option parameters are specified using the form:

```
link86 file [option [parameter] ]
```

Table 7-2 shows the file section option parameters, their abbreviations, and their meanings.

Table 7-2. Command File Option Parameters

Parameter	Abbreviation	Meaning
GROUP	G	groups to be included in command file section
CLASS	C	classes to be included in command file section
SEGMENT	S	segments to be included in command file section
ABSOLUTE	AB	absolute load address for command file section
ADDITIONAL	AD	additional memory allocation for the command file section
MAXIMUM	M	maximum memory allocation for command file section
ORIGIN	O	origin of first segment in command file section

GROUP, CLASS, SEGMENT

The GROUP, CLASS, and SEGMENT parameters each contain a list of groups, classes, or segments that you want LINK-86 to put into the indicated section of the command file. For example, the command

```
A>link86 test [code [segment [code1, code2], group [xyz]]]
```

instructs LINK-86 to put the segments CODE1, CODE2, and all the segments in group XYZ into the CODE section of the file TEST.CMD (or TEST.EXE).

ABSOLUTE, ADDITIONAL, MAXIMUM

The ABSOLUTE, ADDITIONAL, and MAXIMUM parameters tell LINK-86 the values to put in the command file header. These parameters override the default values normally used by LINK-86. Table 7-3 shows the default values.

Each parameter is a hexadecimal number enclosed in square brackets.

The ABSOLUTE parameter indicates the absolute paragraph address where the operating system loads the indicated section of the command file at runtime. A paragraph consists of 16 bytes.

The ADDITIONAL parameter indicates the amount of additional memory, in paragraphs, required by the indicated section of the command file. The program can use this memory for Symbol Tables or I/O buffers at runtime.

The MAXIMUM parameter indicates the maximum amount of memory needed by the indicated section of the command file.

For example, the command

```
A>link86 test [data [add [100], max [1000]], code [abs[40]]]
```

creates the file TEST.CMD whose header contains the following information:

- The DATA section requires at least 100H paragraphs in addition to the data in the command file.
- The DATA section can use up to 1000H paragraphs of memory.
- The CODE section must load at absolute paragraph address 40H.

ORIGIN

The ORIGIN parameter is a hexadecimal value that indicates the byte offset where the indicated section of the command file should begin. LINK-86 assumes a default ORIGIN value of 0 for each section except the DATA section, which has a default value of 100H to reserve space for the Base Page in a native-mode program or the Program Segment Prefix (PSP) in a PC DOS mode program.

Table 7-3 summarizes the default values for each of the command options and parameters.

Table 7-3. Default Values

for Command File Options and Parameters

OPTION	GROUP	CLASS	SEGMENT	ABS	ADD	MAX	ORG
CODE	CGROUP	CODE	CODE	0	0	0	0
DATA	DGROUP	DATA	DATA	0	0	1000H*	100H
STACK	-	STACK	STACK	0	0	0	0
EXTRA	-	EXTRA	EXTRA	0	0	0	0
X1	-	X1	X1	0	0	0	0**
X2	-	X2	X2	0	0	0	0**
X3	-	X3	X3	0	0	0	0**
X4	-	X4	X4	0	0	0	0**

* If there is a DGROUP; otherwise 0H.
** Native-mode only

7.6.2 FILL / NOFILL

The FILL and NOFILL options tell LINK-86 what to do with any uninitialized data at the end of a section of the command file. The FILL option, which is active by default, directs LINK-86 to include this uninitialized data in the command file and fill it with zeros. The NOFILL option directs LINK-86 to omit the uninitialized data from the command file. Note that these options apply only to uninitialized data at the end of a section of the command file. Uninitialized data that is not at the end of a section is always zero filled and included in the command file.

7.6.3 HARD8087

You must use the HARD8087 option if the program contains 8087 instructions. There are no 8087 software emulation routines provided with LINK-86. A program that contains 8087 instructions must always run on a system with an 8087 coprocessor.

7.6.4 CODESHARED (Native-mode only)

The CODESHARED option marks the group in the CMD file header with a group descriptor type 09H (shared code). The default code group descriptor is 01H (non-shared code).

7.7 SYM File Options

The following command options affect the contents of the SYM file created by LINK-86:

- LOCALS
- NOLOCALS
- LIBSYMS
- NOLIBSYMS

These options must appear in the command line after the specific file or files to which they apply. When you specify one of these options, it remains in effect until you specify another. Therefore, if a command line contains two options, the leftmost option affects all of the specified files until the second option is encountered, which affects all of the remaining files specified on the command line.

7.7.1 LOCALS / NOLOCALS

The LOCALS option directs LINK-86 to include local symbols in the SYM file if they are present in the object files being linked. The NOLOCALS option directs LINK-86 to ignore local symbols in the object files. The default is LOCALS. For example, the command

```
A>link86 test1 [nolocals], test2 [locals], test3
```

creates a SYM file containing local symbols from TEST2.OBJ and TEST3.OBJ, but not from TEST1.OBJ

7.7.2 LIBSYMS / NOLIBSYMS

The LIBSYMS option directs LINK-86 to include in the SYM file any symbols coming from a library searched during the link operation. The NOLIBSYMS option directs LINK-86 not to include those symbols in the SYM file.

Typically, such a library search involves the runtime subroutine library of a high-level language such as C. Because the symbols in such a library are usually of no interest to the programmer, the default is NOLIBSYMS.

7.8 MAP File Option

The MAP option directs LINK-86 to create a MAP file containing information about the segments in the command file. The amount of information LINK-86 puts into the MAP file is controlled by the following optional parameters

OBJMAP	NOOBJMAP
L86MAP	NOL86MAP
ALL	NOCOMMON

These parameters are enclosed in brackets following the MAP option. The OBJMAP parameter directs LINK-86 to put segment information about OBJ files into the MAP file. The NOOBJMAP parameter suppresses this information. Similarly, the L86MAP switch directs LINK-86 to put segment information from L86 files into the MAP file. The NOL86MAP parameter suppresses this information. The ALL parameter directs LINK-86 to put all the information into the MAP file. The NOCOMMON parameter suppresses all common segments from the MAP file.

Once you instruct LINK-86 to create a MAP file, you can change the parameters to the MAP option at different points in the command line. For example, the command

A>**link86 finance [map[all]],screen.l86,graph.l86[map[nol86map]]**

directs LINK-86 to create a map file containing segment information from FINANCE.OBJ and SCREEN.L86; segment information for GRAPH.L86 is suppressed by the NOI86MAP option.

If you specify the MAP option with no parameters, LINK-86 uses OBJMAP and NOL86MAP as defaults.

7.9 SEARCH Option

The SEARCH option directs LINK-86 to search the preceding library and include in the command file only those modules satisfying external references from other modules. Note that LINK-86 does not search L86 files automatically. If you do not use the SEARCH option after a library file name, LINK-86 includes all the modules in the library file when creating the command file. For example, the command

```
A>link86 test1, test2, math.l86 [search]
```

creates the native-mode file TEST1.CMD by combining the object files TEST1.OBJ, TEST2.OBJ, and any modules from MATH.L86 referenced directly or indirectly from TEST1.OBJ or TEST2.OBJ.

The modules in the library file do not have to be in any special order. LINK-86 makes multiple passes through the library index when attempting to resolve references from other modules.

LINK-86 automatically uses the SEARCH option when linking compiler-requested libraries.

7.10 Input File Options

The following command options determine how LINK-86 uses the input file:

INPUT
ECHO

The INPUT option directs LINK-86 to obtain further command line input from the indicated file. Other files can appear in the command line before the input file, but the input file must be the last filename on the command line. When LINK-86 encounters the INPUT option, it stops scanning the command line, entered from the console. Note that you cannot nest command input files. That is, a command input file cannot contain the input option.

The input file consists of filenames and options just like a command line entered from the console. An input file can contain up to 2048 characters, including spaces. For example, the file TEST.INP might include the lines:

```
MEMTEST=TEST1,TEST2,TEST3,
IOLIB.L86[S],MATH.L82[S],
TEST4,TEST5[LOCALS]
```

To direct LINK-86 to use this file for input, enter the command

```
A>link86 test[input]
```

If no file type is specified for an input file, LINK-86 assumes INP.

The ECHO option causes LINK-86 to display the contents of the INP file on the console as it is read.

7.11 I/O Option

The $ option controls the source and destination devices under LINK-86. The general form of the $ option is:

```
$Tdrive
```

where T is a file type and **drive** is a single-letter drive specifier.

File Types

LINK-86 recognizes five file types:

C - Command File (CMD or EXE)
L - Library File (L86)
M - Map File (MAP)
O - Object File (OBJ or L86)
S - Symbol File (SYM)

Drive Specifications

The **drive** specifier can be a letter in the range A through P, corresponding to one of sixteen logical drives. Alternatively, it can be one of the following special characters:

X - Console
Y - Printer
Z - No Output

When you use the $ option, you cannot separate the Tdrive character pair with commas. You must use a comma to set off any **$** options from other options. For example, the three command lines shown below are equivalent:

A>**link86 part1[$sz,$od,$lb],part2**

A>**link86 part1[$szodlb],part2**

A>**link86 part1[$sz od lb],part2**

The value of a $ option remains in effect until LINK-86 encounters a countermanding option as it processes the command line from left to right.

7.11.1 $C (Command) Option

The $C option uses the form:

$Cdrive

LINK-86 normally generates the command file on the same drive as the first object file in the command line. The $C option instructs LINK-86 to place the command file on the drive specified by the **drive** character following the $C ($CZ suppresses the generation of a command file).

7.11.2 $L (Library) Option

The $L option uses the form:

$Ldrive

LINK-86 normally searches on the default drive for runtime subroutine libraries linked automatically. The $L option directs LINK-86 to search the specified **drive** for these library files.

7.11.3 $M (Map) Option

The $M option uses the form:

$Mdrive

LINK-86 normally generates the Map file on the same drive as the command file. The $M option instructs LINK-86 to place the Map file on the drive specified by the **drive** character following the $M. Specify $MX to send the Map file to the console or $MY to send the MAP file to the printer.

7.11.4 $O (Object) Option

The $O option uses the form:

$Odrive

LINK-86 normally searches for the OBJ or L86 files that you specify in the command line on the default drive, unless such files have explicit drive prefixes. The $O option allows you to specify the drive location of multiple OBJ or L86 files without adding an explicit **drive** prefix to each filename. For example, the command

A>**link86 p[$od],q,r,s,t,u.l86,b:v**

tells LINK-86 that all the object files except the last one are located on drive D. Note that this does not apply to libraries linked automatically (see Section 7.11.2).

7.11.5 $S Symbol Option

The $S option uses the form:

$Sdrive

LINK-86 normally generates Symbol files on the same drive as the command file. The $S option directs LINK-86 to place these files on the drive specified by the **drive** character following the $S. Specifying $SZ directs LINK-86 not to generate the files.

7.12 The Link Process

The link process involves two distinct phases: collecting the segments in the object files, and then positioning them in the command file.

The following terms are used in this section to describe how LINK-86 processes object files and creates the command file.

Segment	A collection of code or data bytes whose length is less than 64K. A segment is the smallest unit that LINK-86 manipulates.
Segment name	Any valid RASM-86 identifier. LINK-86 combines all segments with the same segment name from separate object files.
Class name	Any valid RASM-86 identifier. LINK-86 uses the class name to position the segment in the correct section of the command file.
Align type	Indicates the type of boundary where the segment begins. The Align types are byte, word, paragraph and page.
Combine type	Determines how LINK-86 combines segments with the same name from different files into a single segment. The Combine types are: public, common, stack, absolute, and local.
Group	A collection of segments with different names grouped into a single segment. By grouping segments, you can combine library modules and other modules of similiar type with your object file modules into a single segment. By combining the contents of individual segments into one large segment, the pointer need only be a 16-bit offset into a single segment. This results in shorter and faster code than addressing individual segments with 32-bit pointers.

If your program is written in a high-level language, the compiler automatically assigns the Segment name, Class name, Group, Align type, and Combine type. If your program is written in assembly language, refer to Section 3 for a description of how to assign these attributes.

7.12.1 Phase 1 - Collection

In Phase 1, LINK-86 first collects all segments from the separate files being linked, and then combines them into the output file according to the combine type, align type, and group type specified in the object module.

Combine Types

The combine type determines how the data and code segments of the individual object files are combined together into segments in the final executable file. There are 5 combine types:

- Public
- Common
- Stack
- Local
- nnnn (absolute segment)

When the Public Combine type is used, LINK-86 combines segments by concatenating them together, leaving the appropriate space between the segments as indicated by the Align type (see below). Public is the most common Combine type, and RASM-86, as well as most high-level language compilers, use it by default.

For example, suppose there are three object files: FILEA.OBJ, FILEB.OBJ, and FILEC.OBJ, and each file defines a data segment, named Dataseg, with the public combine type. Figure 7-1 illustrates how LINK-86 combines this segment using the default combine type, public.

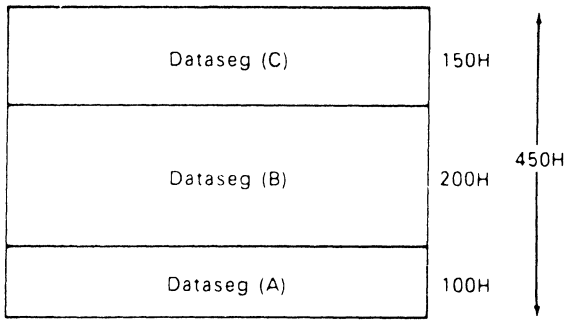


Figure 7-1. Combining Segments with the Public Combine Type

Figure 7-2 illustrates the Common Combine type. Suppose the three files: FILEA.OBJ, FILEB.OBJ, and FILEC.OBJ each contain a data segment, named Dataseg, with the Common Combine type. LINK-86 combines these data segments so all parts of the segments from the separate files being linked have the same low address in memory. The Common Combine type overlays the data or code from the various object files, making it common to all of the linked routines in the executable file. Note that this corresponds to a common block in high-level languages.

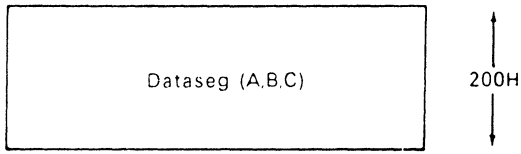


Figure 7-2. Combining Segments with the Common Combine Type

LINK-86 combines segments with the Stack Combine type so the total length of the resulting stack segment is the sum of the input stack segments, including any intersegment gaps specified by the align type.

For example, suppose the three files FILEA.OBJ, FILEB.OBJ, and FILEC.OBJ each contain a segment named Stkseg with the Stack Combine type. Figure 7-3 illustrates how they are combined by LINK-86.

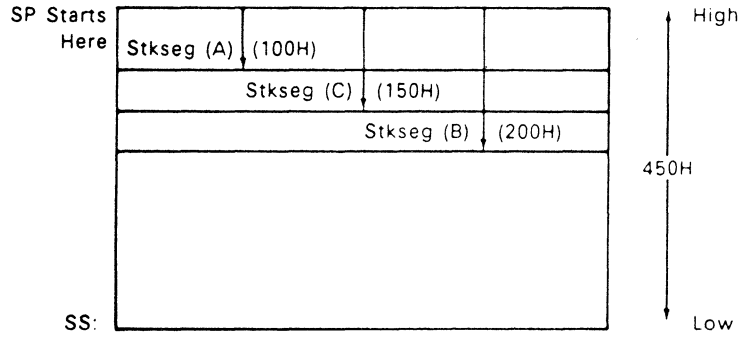


Figure 7-3. Combining Segments with Stack Combination

Segments with the local or absolute combine type cannot be combined. LINK-86 displays an error message if the files being linked contain multiple local segments with the same name.

Align Type

The Align type indicates on what type of boundary the segment begins, and thus determines the amount of space LINK-86 leaves between segments of the same name. When you specify an align type, you determine whether the base address of a segment is to start on a byte, word, paragraph, or page boundary. Four align types can be specified in LINK-86:

- Byte alignment (multiple of 1 byte)
- Word alignment (multiple of 2 bytes)
- Paragraph alignment (multiple of 16 bytes)
- Page alignment (multiple of 256 bytes)

Byte alignment produces the most compact code. When segments are byte aligned, no gap is left between the segments

If the segments are word aligned, LINK-86 adds a one-byte gap, if necessary, to ensure that the next part of the segment begins on a word boundary. Word is the default Align type for Data segments, since the 8086 and 80286 processors perform faster memory accesses for word-aligned data. Word alignment is useful for saving space when a large number of small segments are used. However, the offset of the base of the segment may not be zero.

The gap between paragraph-aligned segments can be up to 15 bytes. Paragraph alignment is used when the offset of the base of the segment must be zero.

Page-aligned segments have up to 255-byte gaps between them. Page alignment is used when creating system applications where the code or data must start on a page boundary.

Suppose the data segment, Dataseg, has the paragraph Align type and has a length of 129H in FILEA, 10EH in FILEB, and 13AH in FILEC. As shown, LINK-86 combines the segments to ensure that each segment begins on a paragraph boundary.

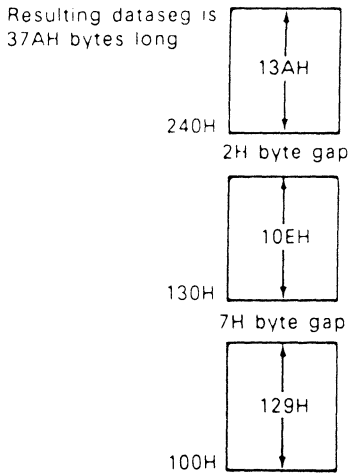
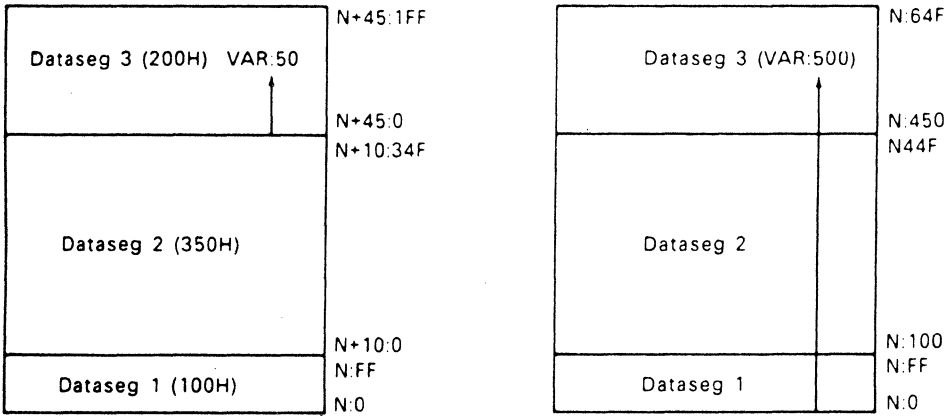


Figure 7-4. Paragraph Alignment

LINK-86 does not align segments having an Absolute combine type because these segments have their load-time memory location determined at translation time.

Grouping

LINK-86 combines segments into groups. When segments are grouped, intersegment gaps are determined using the same Align types as those used to combine segments. Figure 7-5 illustrates how LINK-86 combines segments into groups.



a. Segments Without Groups **b. Segments Within A Group**

Figure 7-5. The Effect of Grouping Segments

In Figure 7-5, N:0 is the base address where the segments are loaded at run-time (paragraph N, offset 0). Figure 7-5a shows that each segment not contained in a group begins at offset zero, and thus can be up to 64K long. The offset of any given location, in this case the variable VAR, is relative to the base of the segment. Thus, in order to access VAR at run-time, the program must load a segment register with the base address of the data segment Dataseg3 and point to an offset of 50H.

In Figure 7-5b, the same segments are combined in a group. The offsets of the segments are now cumulative and thus cannot extend past 64k-bytes (FFFFH). The offset of VAR is 500H relative to the base of the group. At run-time, the program does not need to reload a segment register to point to the base of Dataseg3, but can access VAR directly by pointing to an offset of 500H.

7.12.2 Phase 2 - Create Command File

In Phase 2, LINK-86 assigns each group and segment to a section of the command file as follows:

1. LINK-86 first processes any segments, groups or classes the user placed in a specific section by means of the command line options described in Section 7.5.

2. Segments belonging to the group CGROUP are placed in the CODE section of the command file.

3. Segments belonging to the group DGROUP are placed in the DATA section of the command file. Note that the group names CGROUP and DGROUP are automatically generated by PL/I-86, CB86, and other high-level language compilers.

4. If there are any segments not processed according to (1), (2), and (3), LINK-86 places them in the command file according to their class name, as shown in Table 7-3. This table also shows the RASM-86 segment directives that produce the class names as defaults.

5. Segments not processed by any of the above means are omitted from the command file because LINK-86 does not have sufficient information to position them.

Table 7-4. LINK-86 Usage of Class Names

Class Name	Command File Section	Segment Directive
CODE	CODE	CSEG
DATA	DATA	DSEG
EXTRA	EXTRA	ESEG
STACK	STACK	SSEG
X1*	X1	
X2*	X2	
X3*	X3	
X4*	X4	

* Native-mode only. There is no segment directive in RASM-86 producing this class name as a default; you must supply it explicitly.

See Appendix G for a list of LINK-86 error messages.

End of Section 7

LIB-86 Library Utility

8.1 Introduction

LIB-86 is a utility program for creating and maintaining library files containing 8086 or 80286 object modules. These modules can be produced by any language translator that produces modules in Intel's 8086 or 80286 object module format.

You can use LIB-86 to create libraries, as well as append, replace, select, or delete modules from an existing library. You can also use LIB-86 to obtain information about the contents of library files.

8.2 LIB-86 Operation

When you invoke LIB-86, it reads the indicated files and produces a Library file, a Cross-reference file, or a Module map file as indicated by the command line. When LIB-86 finishes processing, it displays the **Use Factor**, a decimal number indicating the percent of available memory LIB-86 uses during processing.

Table 8-1 shows the filetypes recognized by LIB-86.

Table 8-1. LIB-86 Filetypes

Type	Usage
INP	Input Command File
L86	Library File
MAP	Module Map File
OBJ	Object File
XRF	Cross-reference File

8.3 LIB-86 Command Syntax

LIB-86 uses the command form:

```
LIB86 libraryfile = file1 [options] file2, .... filen
```

LIB-86 creates a Library file with the filename given by LIBRARYFILE. If you omit the filetype, LIB-86 creates the Library file with filetype L86.

LIB-86 reads the files specified by FILE1 through FILEN and produces the library file. If FILE1 through FILEN do not have a specified filetype, LIB-86 assumes a default filetype of OBJ. The files to be included can contain one or more modules; they can be OBJ or L86 files, or a combination of the two.

Modules in a library need not be arranged in any particular order, because LINK-86 searches the library as many times as necessary to resolve references. However, LINK-86 runs much faster if the order of modules in the library is optimized. To do this, remove as many backward references as possible (modules which reference public symbols declared in earlier modules in the library) so LINK-86 can search the library in a single pass.

Module names are assigned by language translators. The method for assigning module names varies from translator to translator, but is generally either the filename or the name of the main procedure.

8.4 Stopping LIB-86

With the native-mode version, you can press any console key to halt LIB-86, which then displays the message

```
STOP LIB-86 (Y/N)?
```

If you type Y, LIB-86 immediately stops processing and returns control to the operating system. Typing N causes LIB-86 to resume processing.

You can stop both the native-mode and PC DOS mode versions by typing Ctrl-C.

8.5 LIB-86 Command Options

When you invoke LIB-86, you can specify optional parameters in the command line controlling the operation. Table 8-2 shows the LIB-86 command options. You can abbreviate each option keyword by truncating on the right, as long as you include enough characters to prevent ambiguity. Thus, EXTERNALS can be abbreviated EXTERN, EXT, EX, or simply, E. The following sub-sections describe the function of each command option.

Table 8-2. LIB-86 Command Line Options

Option	Purpose	Abbreviation
DELETE	Delete a Module from a Library file	D
EXTERNALS	Show EXTERNALS in a Library file	E
ECHO	Echo contents of INP file on console	
INPUT	Read commands from Input file	I
MAP	Create a Module Map	MA
MODULES	Show Modules in a Library file	MO
NOALPHA	Show Modules in order of occurrence	N
PUBLICS	Show PUBLICS in a Library file	P
REPLACE	Replace a Module in a Library file	R
SEGMENTS	Show Segments in a Module	SEG
SELECT	Select a Module from a Library file	SEL
XREF	Create a Cross-reference file	X

8.6 Creating and Updating Libraries

The following sections describe how you create new libraries and update existing libraries.

8.6.1 Creating a New Library

To create a new library, enter the name of the library, then an equal sign followed by the list of the files you want to include, separated by commas. For example,

```
A>lib86 newlib = a,b,c

A>lib86 newlib.l86 = a.obj,b.obj,c.obj

A>lib86 math = add,sub,mul,div
```

The first two examples are equivalent.

8.6.2 Adding to a Library

To add a module or modules to an existing library, specify the library name on both sides of the equal sign in the command line. The library name appears on the left of the equal sign as the name of the library you are creating. The name also appears on the right of the equal sign, with the names of the other file or files to be appended. For example,

```
A>lib86 math = math.l86,sin,cos,tan

A>lib86 math = sqrt,math.l86
```

8.6.3 Replacing a Module

LIB-86 allows you to replace one or more modules without rebuilding the entire library from the individual object files. The command for replacing a module or modules in a library has the general form:

LIB86 newlibrary = oldlibrary [REPLACE [replace list]]

where NEWLIBRARY is the name of the new library file you wish to create; OLDLIBRARY is the name of the existing library file (that can be the same as NEWLIBRARY) containing the module you want to replace; and REPLACE LIST contains one or more module names of the form:

```
modulename = filename
```

For example, the command:

```
A>lib86 math = math.l86 [replace [sqrt=newsqrt] ]
```

directs LIB-86 to create a new file MATH.L86 using the existing MATH.L86 as the source, replacing the module SQRT with the file NEWSQRT.OBJ. If the name of the module being replaced is the same as the file replacing it, you need to enter the name only once. For example, the command:

```
A>lib86 math = math.l86 [replace [sqrt] ]
```

replaces the module SQRT with the file SQRT.OBJ in the Library file MATH.L86.

You can effect multiple replaces in a single command by using commas to separate the names. For example,

```
A>lib86 new = math.l86 [replace [sin=newsin,cos=newcos] ]
```

Note that you cannot use the command options DELETE and SELECT in conjunction with REPLACE.

LIB-86 displays an error message if it cannot find any of the specified modules or files. See Appendix H for a complete list of LIB-86 error messages.

8.6.4 Deleting a Module

The command for deleting a module or modules from a library has the general form:

LIB86 newlibrary = oldlibrary [DELETE [module specifiers]]

where MODULE SPECIFIERS can contain either the names of single modules, or a collection of modules, which are specified using the name of the first and the last modules of the group, separated by a hyphen. For example,

```
A>lib86 math = math.l86 [delete [sqrt] ]

A>lib86 math = math.l86 [delete [add, sub, mul, div] ]

A>lib86 math = math.l86 [delete [add - div] ]
```

You cannot use the command options REPLACE and SELECT in conjunction with DELETE.

LIB-86 displays an error message if it cannot find any of the specified modules in the library (see Appendix H).

8.6.5 Selecting a Module

The command for selecting a module or modules from a library has the general form:

LIB86 newlibrary = oldlibrary [SELECT [module specifiers]]

where MODULE SPECIFIERS can contain either the names of single modules, or groups of modules, which are specified using the name of the first and the last modules of the group, separated by a hyphen. For example,

```
A>lib86 arith = math.l86 [select [add, sub, mul, div] ]
```

```
A>lib86 arith = math.l86 [select [add - div] ]
```

You cannot use the command options DELETE and REPLACE in conjunction with SELECT.

LIB-86 displays an error message if it cannot find any of the specified modules in the library (see Appendix H).

8.7 Displaying Library Information

You can use LIB-86 to obtain information about the contents of a library. LIB-86 can produce two types of listing files: a Cross-reference file and a Library Module Map. Normally, LIB-86 creates these listing files on the default drive, but you can route them directly to the console or the printer by using the command options described in Section 8.5.

8.7.1 Cross-reference File

You can create a file containing the Cross-reference listing of a library with the command:

```
LIB86 libraryname [XREF]
```

LIB-86 produces the file LIBRARYNAME.XRF on the default drive, or you can redirect the listing to the console or the printer.

The Cross-reference file contains an alphabetized list of all Public, External, and Segment name symbols encountered in the library. Following each symbol is a list of the modules in which the symbol occurs. LIB-86 marks the module or modules in which the symbol is defined with a pound sign, #, after the module name. Segment names are enclosed in slashes, as in /CODE/. At the end of the cross-reference listing, LIB-86 indicates the number of modules processed.

8.7.2 Library Module Map

You can create a Module Map of a library using the command:

```
LIB86 libraryname [MAP]
```

LIB-86 produces the file LIBRARYNAME.MAP on the default drive, or you can redirect the listing to the console or the printer.

The Module Map contains an alphabetized list of the modules in the Library file. Following each module name is a list of the segments in the module and their lengths. The Module Map also includes a list of the Public symbols defined in the module, and a list of the External symbols referenced in the module. At the end of the Module Map listing, LIB-86 indicates the number of modules processed.

LIB-86 normally alphabetizes the names of the modules in the Module Map listing. You can use the NOALPHA switch to produce a map listing the modules in the order in which they occur in the library. For example,

```
A>lib86 math.l86 [map,noalpha]
```

8.7.3 Partial Library Maps

You can use LIB-86 to create partial library maps in two ways. First, you can create a map with only module names, Segment names, Public names, or External names using one of the commands:

```
LIB86 libraryname [MODULES]
LIB86 libraryname [SEGMENTS]
LIB86 libraryname [PUBLICS]
LIB86 libraryname [EXTERNALS]
```

You can also combine the SELECT command with any of the map-producing commands described above, or the XREF command. For example,

```
A>lib86 math.l86 [map,noalpha,select [sin,cos,tan] ]

A>lib86 math.l86 [xref,select [sin,cos,tan] ]
```

8.8 LIB-86 Commands on Disk

For convenience, LIB-86 allows you to put long or commonly used LIB-86 command lines in a disk file. Then when you invoke LIB-86, a single command line directs LIB-86 to read the rest of its command line from a file. The file can contain any number of lines consisting of the names of files to be processed and the appropriate LIB-86 command options. The last character in the file must be a normal end-of-file character (1AH).

To direct LIB-86 to read commands from a disk file, use a command of the general form:

```
LIB86 filename [INPUT]
```

If FILENAME does not include a filetype, LIB-86 assumes filetype INP.

As an example, the file MATH.INP might contain the following:

```
MATH = ADD [$OC],SUB,MUL,DIV,
SIN,COS,TAN,
SQRT,LOG
```

Then the command:

```
A>lib86 math [input]
```

directs LIB-86 to read the file MATH.INP as its command line. You can include other command options with INPUT, but no other filenames can appear in the command line after the INP file. For example,

```
A>lib86 math [input,xref,map]
```

The ECHO option causes LIB-86 to display the contents of the INP file on the console as it is read.

8.9 Redirecting I/O

LIB-86 assumes that all the files it processes are on the default drive, so you must specify the drive name for any file not on the default drive. LIB-86 creates the L86 file on the default drive unless you specify a drive name. For example,

```
A>lib86 e:math = math.l86,d:sin,d:cos,d:tan
```

LIB-86 also creates the MAP and XRF files on the same drive as the L86 file it creates, or the same drive as the first object file in the command line if no library is created.

You can override the LIB-86 defaults by using the following command options:

```
$M<drive> - MAP file destination drive
$O<drive> - source OBJ or L86 file location
$X<drive> - XRF file destination drive
```

where <drive> is a drive name (A-P). For the MAP and XRF files, <drive> can be X or Y, indicating console or printer output, respectively. You can also put multiple I/O options after the dollar sign. For example,

```
A>lib86 trig [map,xref,$ocmyxy] = sin,cos,tan
```

The $O switch remains in effect as LIB-86 processess the command line from left to right, until it encounters a new $O switch. This feature can be useful if you are creating a library from a number of files, the first group of which is on one drive, and the remainder on another drive. For example,

```
A>lib86 biglib = a1 [$oc],a2, ...a50 [$od],a51, ...a100
```

End of Section 8

SID-86 Operation

9.1 Introduction

SID-86 is a symbolic debugger designed for use with the Concurrent DOS 86 operating system. SID-86 features:

- Symbolic assembly and disassembly
- Expressions involving hexadecimal, decimal, ASCII, and symbolic values
- Permanent breakpoints with pass counts
- Trace without call

To use SID-86, you should be familiar with the Intel 8088/8086/80286 microprocessor architecture, and Concurrent DOS 86 as described in the Concurrent DOS 86 System Guide and the Concurrent DOS 86 Programmer's Guide.

9.2 Typographical Conventions

The following typographical conventions are used to illustrate SID-86's command and output structures:

- Commands appear in UPPERCASE characters and their arguments appear in lower case characters. This convention is used to distinguish the command from its arguments. Typically, you enter all SID-86 command characters in lower case.
- When an example of a SID-86 command is given, user input is displayed in **bold print**.
- Some of the examples of SID-86 output use horizontal and/or vertical elipses (.....) to illustrate the continuation of an output pattern.

- A <ctrl> sign is used to illustrate the CONTROL (or CTRL) key on your keyboard. For example, <ctrl>-D instructs you to press and hold down the CONTROL key while you press the "D" key.
- [] are used to signify an optional parameter

9.3 Starting SID-86

You start SID-86 by entering a command in one of the following forms:

SID
SID filespec [symfilespec]

The first form loads and executes SID-86. After displaying its sign-on message and prompt character (#) SID-86 is ready to accept commands.

In the second form, **filespec** specifies the name of the file to be debugged. If you do not enter a filetype, SID-86 assumes a CMD filetype (in native mode) or an EXE filetype (in PC DOS mode). **Symfilespec** specifies the optional symbol (SYM) file, with or without file extension.

The following are examples of valid SID-86 command lines:

A>sid Start SID-86

A>sid hello.cmd Start SID-86 and load the command file hello.cmd as the debug process.

A>sid b:hello b:hello
Start SID-86 and load the command file, hello, along with the symbol table file, hello.sym, from the B drive.

9.4 Exiting SID-86

SID-86 does not automatically save any files upon exit. Therefore, to save the modified version of your file, write the file to disk using the W (write) Command described in Section 11.4.19 before exiting SID-86.

You can exit SID-86 by typing Ctrl-C in response to the # prompt (see Section 11.4.12). This returns control to the operating system.

End of Section 9

SID-86 Expressions

10.1 Introduction

SID-86 can reference absolute machine addresses through expressions. Expressions can use names from the program's SYM file, which is created when the program is linked using LINK-86. Expressions can also be literal values in hexadecimal, decimal, or ASCII character string form. You can combine these literal values with arithmetic operators to provide access to subscripted and indirectly-addressed data or program areas.

10.2 Literal Hexadecimal Numbers

SID-86 normally accepts and displays values in hexadecimal. Valid hexadecimal digits consist of the decimal digits 0 through 9 and the hexadecimal digits A, B, C, D, E, and F, which correspond to the decimal values 10 through 15, respectively.

A literal hexadecimal number in SID-86 consists of one or more contiguous hexadecimal digits. If you type four digits, the leftmost digit is most significant and the rightmost digit is least significant. If the number contains more than four digits, the rightmost four are recognized as significant, and the remaining leftmost digits are discarded. The following examples show the hexadecimal and the decimal equivalents of the corresponding input values.

Input Value	Hexadecimal	Decimal
1	0001	1
100	0100	256
fffe	FFFE	65534
10000	0000	0
38001	8001	32769

10.3 Literal Decimal Numbers

Enter decimal numbers by preceding the number with the # symbol. The number following the # symbol must consist of one or more decimal digits (0 through 9), with the most significant digit on the left and the least significant digit on the right. Decimal values are padded or truncated according to the rules of hexadecimal numbers when converted to the equivalent hexadecimal value.

In the following examples, the input values on the left produce the internal hexadecimal values on the right:

Input Value	Hexadecimal Value
#9	0009
#10	000A
#256	0100
#65535	FFFF
#65545	0009

10.4 Literal Character Values

SID-86 accepts one or two printable ASCII characters enclosed in apostrophes as literal values in expressions. Characters remain as typed within the apostrophes (that is, no case translation occurs). The leftmost character is the most significant, and the rightmost character is the least significant. Single character strings are padded on the left with zeros. Strings having more than two characters are not allowed in expressions, except in the S command, as described in Section 11.4.14.

Note that the enclosing apostrophes are not included in the character string, nor are they included in the character count. The only exception is when a pair of contiguous apostrophes is reduced to a single apostrophe and included in the string as a normal graphic character (see examples below).

In the following examples, the strings to the left produce the hexadecimal values to the right. Note that uppercase ASCII alphabetics begin at the encoded hexadecimal value 41; lowercase alphabetics begin at 61; a space is hexadecimal 20 and an apostrophe is hexadecimal 27. #

Input String	Hexadecimal Value
'A'	0041
'AB'	4142
'aA'	6141
''''	0027
''''''	2727
' A'	2041
'A '	4120

10.5 Register Values

You can use the contents of a debug program's register set by specifying a register name wherever a 16-bit number is valid. For example, if you know that at a certain point in the program the BX register points to a data area you want to see, the command

```
DDS:BX
```

displays the desired area of memory. If the current default address segment is DS, you can display the desired area of memory by entering an index register:

```
DBX
```

Note that when assembling instructions using the A command, register names are treated differently than in other expressions. In particular, a register name in an assembly language statement entered in the A (Assemble) command refers to the name of a register, and not its contents.

10.6 Stack References

Elements in the stack can be included in expressions. A caret sign (^) refers to the 16-bit value at the top of the stack, pointed to by the SS and SP registers (SS:SP) in the user's CPU state. A sequence of n carets refers to the nth 16-bit value on the stack. For example, a command having the form:

```
command ^
```

uses the value stored at the top of the stack as its parameter. If two carets are given, the second value stored on the stack is used; three carets specifies the third value on the stack, and so on.

For example, if you wish to display the value located on the top of the stack, you could enter:

```
DSS:^
```

If the address of a segment and the address of a particular offset within that segment are both stored on the stack, carets can be used to specify a complete address. For example, if the third value on the stack is used as a segment address and the first value on the stack is used as the offset within that segment, you can display the complete address using the following command:

```
D^^^:^
```

You can use a stack reference to set a breakpoint on return from a subroutine, even though the actual value is not known.

For example, when callf pushes the current code segment address (CS) onto the stack, followed by the address of the next program instruction (IP), the command

```
G,^^:^
```

transfers control to the program and sets breakpoints at the address contained in the CS and IP registers. This command is the same as:

```
G,CS:IP
```

10.7 Symbolic References

If a symbol table is present during debugging, you can reference values associated with symbols using the following three symbol reference forms:

.s
@s
=s

where **s** represents a sequence of 1 to 31 characters matching a symbol in the table.

The **.s** form gives the 32-bit value associated with the symbol **s** in the symbol table. The **@s** form gives the 16-bit value contained in the word locations pointed to by **s**. The **=s** form gives the 8-bit value at **s** in memory.

For example, given the following excerpt from a SYM table with a segment address of CB0:

```
0000  Variables
0000  Data
0100  Gamma
0102  Delta
```

and given the following memory values:

```
CB0:0100  contains 02
CB0:0101  contains 3E
CB0:0102  contains 4D
CB0:0103  contains 22
```

then the symbol references shown below on the left gives the hexadecimal values shown on the right. Recall that 16-bit 80286 memory values are stored with the least significant byte first. Therefore, the word values at 0100 and 0102 are 3E02 and 224D, respectively.

SYMBOL REFERENCE	HEXADECIMAL VALUE
.GAMMA	CB0:0100
.DELTA	CB0:0102
@GAMMA	3E02
@DELTA	224D
=GAMMA	0002
=DELTA	004D

10.8 Qualified Symbols

Duplicate symbols can occur in the symbol table due to separately assembled or compiled modules that independently use the same name for different subroutines or data areas. Block structured languages allow nested name definitions that are identical, but nonconflicting. Thus, SID-86 allows reference to "qualified symbols" that take the form:

S1/S2/ . . . /Sn

where **S1** through **Sn** represent symbols present in the table during a particular session.

SID-86 always searches the symbol table from the first to last symbol in the order the symbols appear in the symbol file. For a qualified symbol, SID-86 begins by matching the first **S1** symbol, then searches for a match with symbol **S2**, continuing until symbol **Sn** is matched. If this search and match procedure is not successful, SID-86 prints a ? to the console. Suppose, for example, that part of the symbol table has a segment address of D00 appearing in the symbol file as follows:

0100 A 0300 B 0200 A 3E00 C 20F0 A 0102 A

Then the unqualified and qualified symbol references shown below on the left produce the hexadecimal values shown on the right.

Symbol Reference	Hexadecimal Value
.A	D00:0100
@A	2D04
.A/A	D00:0200
.C/A/A	D00:0102
=C/A/A	005E
.B/A/A	D00:20F0

10.9 Expression Operators

Literal numbers, strings, and symbol references can be combined into symbolic expressions using unary and binary "+" and "-" operators. SID-86 evaluates the expression from left to right, producing a 32-bit address at each step. Overflow and underflow are ignored as the evaluation proceeds. The final value becomes the command parameter, whose interpretation depends upon the particular command letter preceding it.

When placed between two operands, the + indicates addition to the previously accumulated value. The sum becomes the new accumulated value in the evaluation.

The - symbol causes SID-86 to subtract the literal number or symbol reference from the 16-bit value accumulated thus far in the symbolic expression. If the expression begins with a minus sign, then the initial accumulated value is taken as zero. That is,

```
-x   is computed as   0-x
```

where **x** is any valid symbolic expression. For example, the address

```
0700-100
```

is the same as the address

```
0600
```

In commands specifying a range of addresses (i.e., B, D, L, F, M and W), the ending address of the range can be indicated as an offset from the starting address. To do this, you can precede the desired offset with a plus sign. For example, the command

```
DFD00,+#512
```

displays the memory from offset address FD00 to FF00. SID-86 does not allow use of the unary plus operator at other times.

10.10 Sample Symbolic Expressions

Frequently, the formulation of symbolic expressions is closely related to the program structures in the program being tested. Suppose you want to debug a sorting program containing the following data items:

LIST	Names the base of a table (or array) of byte values to sort, assuming there are no more than 255 elements, denoted by LIST(0), LIST(1), ... , LIST(254).
N	A byte variable that gives the actual number of items in LIST, where the value of N is less than 256. The items to sort are stored in LIST(0) through LIST(N-1).
I	The byte subscript that indicates the next item to compare in the sorting process. LIST(I) is the next item to place in sequence, where I is in the range 0 through N-1.

Given these data areas, the command

```
D.LIST,+#254
```

displays the entire area reserved for sorting as follows:

```
LIST(0), LIST(1), . . . , LIST(254)
```

The command

```
D.LIST,+=I
```

displays the LIST vector up to and including the next item to sort as follows:

```
LIST(0), LIST(1), . . . , LIST(I)
```

The command

```
D.LIST+=I,+0
```

displays only LIST(I).

Finally, the command

```
D.LIST,+=N-1
```

displays only the area of LIST holding active items to sort as follows:

```
LIST(0), LIST(1), . . . , LIST(N-1)
```

End of Section 10

SID-86 Commands

11.1 Command Structure

When SID-86 is ready to accept a command, it prompts you with a pound sign (#), after which you can enter one of the commands described in this section, or type a CTRL-C to end the debugging session.

A valid SID-86 command can have up to 256 characters and must be terminated with a carriage return. A SID-86 command can be followed by one or more arguments. The arguments can be symbolic expressions, filenames, or other information, depending on the command. Arguments are separated from each other by commas or spaces. Several commands (D, G, N, P, S, T, and U) can be preceded by a minus sign. The effect of the minus sign varies among commands. See the commands in Section 11.4.25 for explanations of the effects of the minus sign on each command.

11.2 Specifying an Address

Most SID-86 commands require one or more addresses as operands. Enter an address as follows:

```
ssss:0000
```

where **ssss** represents an optional 16-bit segment number and **0000** is a 16-bit offset. If you omit the segment value, SID-86 uses a default value appropriate to the command being executed, as described in Section 11.4.4.

It is not possible to have a debugged process (read in by the E command or command line) and a file (read in by the R command) simultaneously resident in SID-86. (See Sections 11.4.4 and 11.4.13 for descriptions of the E and R commands.)

11.3 Line Editing Functions

When you enter a command, use standard Concurrent line-editing functions to correct typing errors. These line-editing functions are:

CTRL-X	erase from beginning of line to cursor
CTRL-S	move cursor to left
CTRL-D	move cursor to right

SID-86 does not process the command line until you enter a carriage return.

11.4 SID-86 Commands

This section describes each SID-86 command in alphabetic order. Table 11-2 at the end of this chapter summarizes SID-86 commands.

11.4.1 A (Assemble) Command

The A command assembles 8086/80286 mnemonics directly into memory. It has the form:

As

where s is the address where assembly begins. SID-86 responds to the A command by displaying the address of the memory location where assembly begins. At this point, you can enter assembly language statements as described in Section 4. When you enter a statement, SID-86 converts it to binary, places the value(s) in memory, and displays the address of the next available memory location. This process continues until you press the carriage return without entering any statement or after entering only a period.

SID-86 responds to invalid statements by displaying the message:

```
Bad command or parameter; press ? for help
```

and redisplaying the current assembly address.

Note that wherever a numeric value is valid in an assembly language statement, you can also enter an expression. There is one difference between expressions in assembly language statements and those appearing elsewhere in SID-86: under the A command, references to registers refer to the names of the registers, while elsewhere they refer to the contents of the registers. When you use the A command, you cannot reference the contents of a register in an expression.

The following is an example of the A command:

#a213 Assemble at offset 213 of the current default CS value.

nnnn:0213 **mov ax,#128**
Set AX register to decimal 128.

nnnn:0216 **push ax**
Push AX register on stack.

nnnn:0217 **call .proc1**
Call procedure whose address is the value of the symbol PROC1.

nnnn:021A **test byte [.i/i], 80**
Test the most significant bit of the byte whose address is the value of the second occurrence of the symbol I.

nnnn:021E **jz .done**
Jump if zero flag set to the location whose address is the value of the symbol DONE.

nnnn:0220 . stop assemble process.

11.4.2 B (Block Compare) Command

The B command compares and displays the difference between two blocks of memory loaded by either an R command, E command, or command line. The B command has the form:

Bs1,f1,s2

where **s1** is the address of the start of the first block; **f1** is the offset address that specifies the last byte of the first block, and **s2** is the address of the start of the second block. If the segment is not specified in **s2**, the same value used for **s1** is assumed.

SID-86 displays any differences in the two blocks in the form:

a1 b1 a2 b2

where the **a1** and the **a2** 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.

The following are examples of the B command:

#b40:0,1ff,60:0 Compare 512 (200H) bytes of memory starting at 40:0 and ending at 40:1FF with the block of memory starting at 60:0.

#bes:.array1,+ff,.array2
Compare a 256-byte array starting at offset ARRAY1 in the extra segment with ARRAY2 in the extra segment.

11.4.3 D (Display) Command

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

1. D
2. Ds
3. Ds,f
4. DW
5. DWs
6. DWs,f
7. -Dn

where **s** is the starting address of the display, and **f** is the ending address. If no segment value is given for **f**, then the segment specified by **s** is assumed and the value of **f** is used as the offset within that segment.

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

ssss:0000 bb bb . . . bb aa . . . a

where **ssss** is the segment being displayed, and **0000** is the offset within segment **ssss**. The **bb**'s represent the 8-bit contents of the memory locations in hexadecimal, and the **a**'s represent the contents of memory in ASCII. A period represents any nonprintable ASCII character.

Form 1 displays memory from the current display address for 12 display lines. Form 2 is similar to form 1, except the default display address is changed to address **s**. Form 3 displays the memory block between locations **s** and **f**. Forms 4,5 and 6 are identical to forms 1,2 and 3 except that the contents of memory are displayed as 16-bit words, rather than 8-bit bytes, as follows:

ssss:0000 wwww wwww . . . wwww aaaa . . . aa

where **wwww** represents a 16-bit word in hexadecimal.

You can use Ctrl-S and Ctrl-Q to control scrolling during a long display. In native mode, you can stop the display by typing any key at the console; in PC DOS mode, type Ctrl-Break.

The last address displayed becomes the default starting address for the next display unless another starting address is specified.

By default, the D command displays 176 bytes of memory. Form 7 form changes the default number to **n**, which can be any number between 0 and 65535.

If the number of bytes left in the debugged process is less than the established default value, then only those bytes remaining in the process are displayed.

The following are examples of the D command:

#df00,f23	Display memory bytes from offset F00H through F23H in the current data segment.
#d.array+=i,+#10	Display 10 bytes starting at location ARRAY (i).
#dwss:sp	Display the value at the top of stack in word format.
#d^	Display the value at the top of stack in byte format.
#dw#128,#255	Display memory words from offset 80H through FFH.
#-d10	Set the default number of bytes displayed to 16.

11.4.4 E (Load Program, Symbols for Execution) Command

The E command loads a file into memory so a subsequent G, T, or U command can begin program execution. The E command can also load a symbol table file. The E command has the following forms:

1. [-] Efilespec
2. [-] Efilespec symfilespec
3. E*symfilespec[,symfilespec...]
4. E

Form 1 loads the command file specified by **filespec**. If you do not enter a filetype for the file, SID-86 assumes either a CMD or EXE filetype, depending on which version you are using. SID-86 alters the contents of the CS, DS, ES, and IP registers according to the information in the header of the file loaded. When the file is completely loaded, SID-86 displays the start and end addresses of each segment in the file. You can use the V command to redisplay this information later. See the V Command in Section 11.4.18.

Form 2 loads the command file specified by **filespec** as described above, and then loads a symbol file as specified in **symfilespec**. The default filetype for a symbol file is SYM. SID-86 displays the message:

SYMBOLS

when it begins loading the symbol file. If SID-86 detects an invalid hexadecimal digit or an invalid symbol name, it displays an error message and stops loading the symbol file. You can use the H command to display the symbols loaded when the error occurred to determine the exact location of the error in the SYM file. 64K bytes of memory is available for symbol table storage.

The PC DOS version of SID-86 loads symbols at the top of memory, and adjusts the memory word size in the Program Segment Prefix accordingly.

Form 3 does not load a program but simply loads the specified symbols files. The form E releases all memory being used for symbol table files.

When using the PC DOS version, forms 1 and 2 have an optional - parameter When LINK-86 creates an object file, by default it inserts some prefix code at the beginning of the file to set up the environment in which the program runs. SID-86 automatically executes this prefix code. You cam use the - parameter to direct SID-86 not to execute the prefix code. You can then proceed to the point where the prefix code transfers control to the main program using the following commands:

```
G,102
T
```

When loading a program file with the E command, SID-86 releases any blocks of memory allocated by any previous E or R command or by programs executed under SID-86. Therefore, only one file at a time can be loaded for execution and that file should be loaded before any symbol tables are read. Do not load any symbol files after program execution begins.

SID-86 issues an error messsage if a file does not exist or cannot be successfully loaded in the available memory.

The symbol table file is produced by LINK-86 in the format:

```
nnnn symbol1  nnnn symbol2  . . . .
      .
      .
      .
      .
```

where **nnnn** is a four digit hexadecimal number, and spaces, tabs, carriage returns, and line-feeds serve as delimiters between hexadecimal values and symbol names. Symbol names can be up to 31 characters in length.

The following are examples of the E command (in the native mode):

#etest Load file TEST.CMD

#etest.cmd test.sym
Load file TEST.CMD and symbol table file TEST.SYM

#etest test io Load file TEST.CMD and symbol table files TEST.SYM and IO.SYM

11.4.5 F (Fill) Command

The F command fills an area of memory read into SID-86 using an E command, or R command with a byte or word constant. The F command has the following forms:

1. Fs,f,b
2. FWs,f,w

where **s** is a starting address of the block to be filled and **f** is the address of the final byte of the block. If no segment value is specified by **f**, then the segment value of **s** is used by default. Similarly, if no segment value is specified by **s**, then the current display address is used by default.

Form 1 stores the 8-bit value **b** in locations **s** through **f**. Form 2 stores the 16-bit value **w** in locations **s** through **f** in standard form, the low eight bits first followed by the high eight bits.

If **s** is greater than **f**, or the value **b** is greater than 255, SID-86 responds with the message:

```
Bad command or parameter; press ? for help
```

The following are examples of the F command:

#f100,13f,0	Fill memory at the current default display segment from offsets 100H through 13FH with 0.
#f.array,+255,ff	Fill the 256-byte block starting at ARRAY with the constant FFH.

11.4.6 G (Go) Command

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

1. G
2. G,b1
3. G,b1,b2
4. Gs
5. Gs,b1
6. Gs,b1,b2
7. -G (with all of the above forms)

where **s** is an address where program execution is to start, and **b1** and **b2** are addresses of breakpoints. If you do not supply a segment value for any of these three addresses, the segment value defaults to the contents of the CS register.

For forms 1,2, and 3, no starting address is specified, so SID-86 gives the address from the CS and IP registers. Form 1 transfers control to your program without setting any breakpoints. Forms 2 and 3 set one and two breakpoints respectively before passing control to your program. Forms 4,5 and 6 are identical to 1,2 and 3, except the CS and IP registers are first set to **s**.

If you precede any form of the G command with a minus sign, the intermediate permanent breakpoints set by the P command are not displayed.

Once SID-86 transfers control to the program under test, it executes in real time until a breakpoint is encountered. At this point, SID-86 regains control, clears the breakpoints set by the G command, and displays the address where the executing program is interrupted. This is done using the format:

```
*ssss:0000  .symbol
```

where **ssss** corresponds to the CS register, **0000** corresponds to the IP register where the break occurs, and **.symbol** is the symbol whose value is equal to **0000**, if such a symbol exists. When a breakpoint returns control to SID-86, the instruction at the breakpoint address has not yet been executed.

The following are examples of the G command:

#g	Begin program execution at address given by the CS and IP registers with no breakpoints set.
#g.start,.error	Begin program execution at label START in the code segment, setting a breakpoint at label ERROR.
#g,.error,^	Continue program execution address given by the CS and IP registers, with breakpoints at label ERROR and at the address at the top of the stack.
#-g,34f	Begin execution with a breakpoint at offset 34FH to the current segment value of CS, suppressing intermediate pass point display.

11.4.7 H (Hexadecimal Math) Command

The H command provides several useful arithmetic functions. The H command has the following forms:

1. Ha,b
2. Ha
3. H
4. H .symbol

Form 1 form computes the sum (ssss), difference (dddd), product (pppppppp), and quotient (qqqq) with the remainder (rrrr) of two 16-bit values. The results are displayed in hexadecimal notation as follows:

```
+ ssss  - dddd  * pppppppp  / qqqq  (rrrr)
```

Underflow and overflow are ignored in addition and subtraction.

Form 2 displays the value of the expression **a** in hexadecimal, decimal, and ASCII (if the value has a graphic ASCII equivalent) in the following format:

```
hhhh  #ddddd  'c'
```

Form 3 displays the symbols currently loaded in the SID-86 symbol table. Each symbol is displayed in the following form:

```
nnnn  symbolname
```

You can stop the display by pressing any key at the console (in the native mode version) or Ctrl-Break (in the PC DOS version).

Form 4 allows you to display the address where the specified symbol is defined in the symbol table.

If the symbol is found in the symbol table, SID-86 responds:

```
ssss:0000 #dddd 'c'  .symbol
```

where **ssss:0000** is the address, **#dddd** is the decimal equivalent of 0000, and **c** is the ASCII value of #dddd. If the symbol is not found, SID-86 displays the message:

```
Bad command or parameter; press ? for help
```

The H command uses 16-bit arithmetic with no overflow handling, except for the product in form 1 above. Without overflow handling, the value:

```
ffff + 2
```

equals 1.

The following are examples of the H command:

#h List all symbols and values loaded with the E command(s).

#h@index Show the word contents of the memory location at INDEX in hexadecimal and decimal.

#h5c28,80 Show sum, difference, product, and quotient of 5C28H and 80H.

11.4.8 I (Input Command Tail) Command

In the native-mode version, the I command prepares a File Control Block (FCB) and command tail buffer in SID-86's Base Page and copies this information into the Base Page of the last file loaded with the E command.

In the PC DOS version, the I command prepares a Program Segment Prefix (PSP) and copies this information into the PSP of the last file loaded with the E command.

The I command has the form:

I<command tail>

where **<command tail>** is a character string that usually contains one or more filenames. The first filename is parsed into the default FCB at 005CH. The optional second filename is parsed into the second part of the default FCB at 006CH. The characters in the command tail are also copied into the default command buffer at 0080H. The command tail's length is stored at 0080H, followed by the character string terminated with a binary zero.

If a file has been loaded with the E command, SID-86 copies the FCB and command buffer from the Base Page (or PSP) to the Base Page (or PSP) of the loaded program. The location of SID-86's PSP can be obtained from the 16-bit value at location 0:6. The location of the Base Page (or PSP) of a program loaded with the E command is the value displayed for DS on completion of the program load.

Examples (PC DOS version)

#ifile1.exe Set up a File Control Block at 05CH for FILE1.EXE and put the string "file1.exe" in the buffer at 80H (in the PSP of the last file loaded with the E command).

#ia:file1 b:file2 c:file3 $px

Set up FCB's at 5CH and 6CH for the files A:FILE1 and B:FILE2, and copy the string following the i into the buffer at 80H.

11.4.9 L (List) Command

The L command lists the contents of memory read into SID-86 using the R command, the E command, or the command line in assembly language. The L command has the following forms:

1. L
2. Ls
3. Ls,f
4. -L (with all of the above forms)

where **s** is the address where the list starts and **f** is the address where the list finishes. If no segment value is given for **f**, then the segment value specified by **s** is assumed and the value of **f** is used as the offset within that segment.

Each disassembled instruction takes the form:

label:
ssss:0000 prefixes opcode operands .symbol = memory value

where **label** is the symbol whose value is equal to the offset **0000**, if such a symbol exists; **prefixes** are segment override, lock, and repeat prefixes; **opcode** is the mnemonic for the instruction; **operands** is a field containing 0, 1, or 2 operands, as required by the instruction; and **.symbol** is the symbol whose value is equal to the numeric operand, if there is one and such a symbol exists. If the instruction references a memory location, the L command displays the contents of the location in the **memory value** field as a byte, word, or double word, as indicated by the instruction.

Form 1 lists 12 disassembled instructions from the current list address. Form 2 sets the list address to **s** and then lists 12 instructions. Form 3 form lists disassembled code from **s** through **f**. If you precede any of the L command forms with a minus sign, no symbolic information is displayed (the labels and **symbol** fields are omitted).

This speeds up the listing if many symbols are present and you have no need to display them.

In all forms, the list address is set to the next unlisted location in preparation for a subsequent L command. When SID-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.

You can control display scrolling with Ctrl-S and Ctrl-Q. In native mode, you can stop the list by typing any key; in PC DOS mode, type Ctrl-Break.

The syntax of the assembly language statements produced by the L command is described in Section 5.

If the memory location being disassembled is not a valid 8086/80286 instruction, SID-86 displays:

```
??= nn
```

where **nn** is the hexadecimal value of the contents of the memory location.

The following are examples of the L command:

#l	Disassemble 12 instructions from the current default list address.
#-l	Disassemble 12 instructions, without symbols, from the current default list address.
#l243c,244e	Disassemble instructions from 243CH through 244EH.
#l.find,+20	Disassemble 20H bytes from the label FIND.
#l.err+3	Disassemble 12 lines of code from the label ERR plus 3.
#l.err,.err1	Disassemble from label err to label err1.

11.4.10 M (Move) Command

The M command copies a block of data values read into SID-86 using an E command, R command, or command line from one area of memory to another. The M command has the form:

Ms,f,d

where **s** is the starting address of the block to be moved; **f** is the offset of the final byte within the segment; and **d** is the address of the first byte of the area to receive the data. If you do not specify the segment in **d**, the M command uses the same value used for **s**. Therefore, the data found between the **s** and **f** is copied to a location starting at **d**.

The following are examples of the M command:

#m20:2400,+9,30:100
Move 10 bytes from 20:2400 to 30:100.

#m.array,+#63,.array2
Move 64 bytes from ARRAY to ARRAY2.

11.4.11 P (Permanent Breakpoint) Command

The P command sets, clears, and displays "permanent" breakpoints. The P command has the following forms:

1. Pa,n
2. Pa
3. -Pa
4. -P
5. P

A permanent breakpoint remains in effect until you explicitly remove it, as opposed to breakpoints set with the G command that must be reentered with each G command. Pass points have associated pass counts ranging from 1 to 0FFFFH. The pass count indicates how many times the instruction at the pass point executes before the control returns to the console. SID-86 can set up to 30 permanent breakpoints at a time.

Forms 1 and 2 are used to set pass points. Form 1 sets a pass point at address **a** with a pass count of **n**, where **a** is the address of the pass point, and **n** is the pass count from 1 to 0FFFFH. If a pass point is already active at **a**, the pass count is changed to **n**. SID-86 responds with a question mark if there are already 16 active pass points.

Form 2 sets a pass point at **a** with a pass count of 1. If a pass point is already active at **a**, the pass count is changed to 1. If there are already 16 active pass points, SID-86 responds with the message:

```
Bad command or parameter; press ? for help
```

Forms 3 and 4 are used to clear pass points. Form 3 clears the pass point at location **a**. SID-86 responds with a question mark if there is no pass point set at **a**. Form 4 clears all the pass points.

Form 5 displays all the active pass points using the form:

```
nnnn  ssss:0000  .symbol
```

where **nnnn** is the current pass count for the pass point; **ssss:0000** is the segment and offset of the pass point location, and **.symbol** is the symbolic name of the offset of the pass point, if such a symbol exists. When a pass point is encountered, SID-86 displays the permanent breakpoint information in the form:

```
nnnn  PASS  ssss:0000  .symbol
```

where **nnnn**, **ssss:0000**, and **.symbol** are as previously described. Next, SID-86 displays the CPU state before the instruction at the permanent breakpoint is executed. SID-86 then executes the instruction at the permanent breakpoint. If the pass count is greater than 1, SID-86 decrements the pass count and transfers control back to the program under test.

When the pass count reaches 1, SID-86 displays the break address (that of the next instruction to be executed) in the following form:

```
*ssss:0000  .symbol
```

Once the pass count reaches 1, it remains at 1 until the permanent breakpoint is cleared or the pass count is changed with another P command.

You can suppress the intermediate pass point display with the -G command (see the G Command in Section 11.4.6). When the -G command is used, only the final pass points (when the pass count = 1) are displayed.

You can use permanent breakpoints in conjunction with breakpoints set with the G command.

Normally, SID-86 does not display the segment registers at pass points. You can use the S and -S commands to enable and disable the segment register display (see the S Command in Section 11.4.14).

The following are examples of the P command:

#p	Display active permanent breakpoints.
#p.error	Set permanent breakpoint at label ERROR.
#p.print,17	Set permanent breakpoint at label PRINT with count of 17H.
#-p	Clear all permanent breakpoints.
#-p.error	Clear permanent breakpoint at label ERROR.

11.4.12 QI, QO (Query I/O) Command

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 QI command has two forms:

```
QIn
QIWn
```

where n is the 16-bit port number. The first form displays the 8-bit value read from port n. The second form displays the 16-bit value read from port n.

The QO command has two forms:

```
QOn,v
QOWn,v
```

where **n** is the 16-bit port number, and **v** is the value to output. The first form writes the 8-bit value v to port n. If v is greater than 255, SID-86 responds with a question. The second form writes the 16-bit value v to the port n.

Examples

#qiw20	Displays the 16-bit value of input port 20H.
#qi1024	Displays the 8-bit value of input port 1024.
#qow20,ff7e	Sets the 16-bit output port number 20H to 0FF7EH.
qo#1025,2	Sets the 8-bit output port number 1025 to 2.

11.4.13 R (Read) Command

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

Rfilespec

where **filespec** is the name of the file you want to read. When you use the R command, SID-86 automatically determines the memory location into which the file is read.

When you enter the R command after a process is loaded with the E command, or from the command line during the debugging process, the process being debugged is stopped. Similarly, entering an E command erases the buffered information formed by the R command.

When you enter the R command, SID-86 reads the file into memory, computes, allocates, and displays the start and end addresses of the block of memory occupied by the file. You can use the V command to redisplay this information at a later time. SID-86 sets the default display pointer for subsequent D commands to the start of the block occupied by the file.

The R command does not free any memory previously allocated by another R command. Therefore, you can read a number of files into memory without them overlapping. When the R command is used, files are concatenated together in memory in the same order in which they were read in.

SID-86 issues an error message if the file does not exist or there is not enough memory to load the file.

The following are examples of the R command:

#rbanner.exe	Read file BANNER.EXE into memory.
#rtest	Read file TEST into memory.

11.4.14 S (Set) Command

The S command changes the contents of bytes or words of memory read into SID-86 with the E command, R command, or command line. The forms are as follows:

1. Ss
2. SWs
3. S
4. -S

where **s** is the address where the change occurs.

SID-86 displays the memory address and its current contents on the following line. In response to form 1, the display is:

```
ssss:0000 bb
```

where **bb** is the contents of memory in byte format. In response to form 2, the display is:

```
ssss:0000 wwww
```

where **wwww** is the contents of memory in word format.

You can choose to alter the memory location or to leave it unchanged. If you enter a valid expression, the contents of the byte (or word) in memory is replaced with the value of the expression. If you do not enter a value, the contents of memory are unaffected and the contents of the next address are displayed. In either case, SID-86 continues to display successive memory addresses and values until you enter a period on a line by itself or until SID-86 detects an invalid expression.

With form 1, you can enter a string of ASCII characters, beginning with a quotation mark and ending with a carriage return. The characters between the quotation mark and the carriage return are placed in memory starting at the address displayed. No case conversion takes place. The next address displayed is the address following the character string.

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

Forms 3 and 4 control the display of the segment registers when the CPU state is displayed with the T (Trace) command and at pass points. Form 3 turns on the segment register display and form 4 turns it off. You can turn off the segment register display while debugging to allow the CPU state display to fit on one line.

The following are examples of the S command.

#s.array+3	Begin set at ARRAY (3)
nnnn:1234 **55 0**	Set byte to 0.
nnnn:1235 **55 'abc'**	Set three bytes to a, b, c.
nnnn:1238 **55 #75**	Set byte to decimal 75.
nnnn:1239 **55 .**	Terminate set command.
#s	Enable segment register display in CPU state display.
#-s	Disable segment register display in CPU state display.

11.4.15 SR (Search for String) Command

The SR command searches for a string of characters of values within memory. The SR command has two forms:

```
SRs,f,"string"
SRs,f,value[,value]
```

where **s** is the starting address to begin searching and **f** is the finishing address to end searching.

Form 1 searches for a string of 1 to 30 printable ASCII characters. The "**string**" parameter specifies the string to search for. Note that **string** can use either single (') or double (") quotes.

Form 2 searches for a numerical hex value between 0 and FFH (one byte) in size. The **value** parameter must be a hexadecimal number within the range specified above (leading 0's do not need to be specified). To search for a multiple byte (up to 16 bytes) pattern, separate each byte value with a comma.

Both forms of the SR command can search for the same things, since a numerical value also equals an ASCII value.

The following are examples of the SR command:

#SR56:00,56:1ff,0D,0A
: search memory starting at 56:00 and ending at 56:1FF for a two-byte value consisting of 0Dh (Carriage Return) and 0Ah (line feed).

#SR56:1ff,56:d0ff,"ABCD"
: search memory starting at 56:1FF and ending at 56:d0ff for the character string: ABCD.

#SR56:iff,56:d0ff,41,42,43,44
: search memory starting at 56:1FF and ending at 56:d0ff for a four-byte value consisting of 41 (A), 42 (B), 43 (C), and 44 (D).

11.4.16 T (Trace) Command

The T command traces program execution for 1 to 0FFFFH program steps. After each trace, SID-86 displays the current state of the CPU and the next disassembled instruction to be executed. You must read programs into SID-86 with the E command or command line. The T command has the following forms:

1. T
2. Tn
3. TW
4. TWn
5. -T (with all of the above forms)

where **n** is the number of program steps to execute before returning control to the console. If you do not specify the number of program steps, SID-86 executes a single program step. To stop the trace, pressing any key (in native mode), or Ctrl-Break (in PC DOS mode).

A program step is generally a single instruction, with the following exceptions:

- If a system interrupt instruction (either a native-mode BDOS or PC DOS mode DOS interrupt) is traced, the entire function is treated as one program step and executed in real time. This is because SID-86 makes its own function calls and the parts of Concurrent are not reentrant.
- If the traced instruction is a MOV or POP whose destination is a segment register, the CPU executes the next instruction immediately. This is due to a feature of the microprocessor that disables interrupts, including the Trace Interrupt, for one instruction after a MOV or POP loads a segment register. This allows a sequence such as:

  ```
  MOV  SS, STACKSEGMENT
  MOV  SP, STACKOFFSET
  ```

  to be executed with no chance of an interrupt occurring between the two instructions, at which time the stack is undefined. Such a sequence of MOV or POP instructions, plus one instruction after the sequence is considered one program step.
- If you use forms 3 or 4 and the traced instruction is a CALL, CALLF, or INT, the entire called subroutine or interrupt handler (and any subroutines called therein) is treated as one program step and executes in real time.

After each program step is executed, SID-86 displays the current CPU state, the next disassembled instruction to be executed, the symbolic name of the instruction operand (if any), and the contents of the memory location(s) referenced by the instruction (if appropriate). See the X Command in Section 11.4.20 for a detailed description of the CPU state display.

If a symbol has a value equal to the instruction pointer (IP), the symbol name followed by a colon is displayed on the line preceding the CPU state display. The segment registers are normally not displayed with the T command, which allows the entire CPU state to be displayed on one line. Use the S command, as described in Section 11.4.14, to enable the segment register display. With the segment register display enabled, the display of the CPU state is identical to that of 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 you do not specify the number of program steps, as in form 1, one program step is executed. Otherwise, SID-86 executes **n** program steps and displays the CPU state before each step, as in form 2. You can stop a long trace before **n** steps are executed by typing any character (in native mode) or Ctrl-Break (in PC DOS mode).

After **n** steps are executed, SID-86 displays the address of the next instruction to be executed, along with the symbolic value of the IP register (if there is such a symbol) in the following form:

```
*ssss:0000  .symbol
```

Forms 3 and 4 trace execution without breaking for calls to subroutines. The entire subroutine called from the program being traced is treated as a single program step and executed in real time. This allows tracing at a high level of the program, ignoring subroutines already debugged.

If you precede the command with a minus sign, SID-86 omits symbolic labels and symbolic operands from the CPU state display. This can speed up the display by skipping the symbol table lookup when large symbol tables are loaded.

When a single instruction is being traced, interrupts are disabled for the duration of the instruction.

This prevents SID-86 from tracing through interrupt handlers when debugging on systems in which interrupts occur frequently.

After a T command, SID-86 sets the list address used in the L command at the address of the next instruction to be executed. SID-86 also sets the default segment values to the CS and DS register values.

The following are examples of the T command:

#t	Trace one program step.
#tffff	Trace 65535 steps.
#-t#500	Trace 500 program steps with symbolic lookup disabled.

11.4.17 U Command

The U command, like the T command, is used to trace program execution. The U command functions in the same way as the T command, except that the CPU state is displayed after the last set of program steps have executed, rather than after every step.

The U command works only with programs loaded by the E command or from the command line. The U command has the following forms:

1. U
2. Un
3. UW
4. UWn
5. -U (with all of the above forms)

where **n** is the number of instructions to execute before returning control to the console. You can stop the U command before **n** steps are executed by pressing any key (in native mode) or Ctrl-Break (in PC DOS mode).

Forms 3 and 4 trace execution without calls to subroutines. The entire subroutine called from the program being traced is treated as a single program step and executed in real time. This allows tracing at a high level of the program, ignoring subroutines already debugged.

Preceding any of the U command forms with a minus sign causes SID-86 not to print any symbolic reference information. This allows the program to execute faster.

The following are examples of the U command:

#u200	Trace without display 200H steps. ?r **#-u200** Trace without display 200H steps, suppressing the intermediate pass point display.

11.4.18 V (Value) Command

The V command displays information about the last file loaded with the E or R commands, excluding symbol tables loaded with the E command. The form is:

V

If you load the last file with the R command (or the E command in PC DOS mode), the V command displays the start and end addresses of the file. In native mode, if you read the last file with the E command, the V command displays the start and length in bytes for the code, data, and heap segments.

If an R or E command have not have been used, SID-86 responds to the V command with the message:

```
Bad command or parameter; press ? for help
```

11.4.19 W (Write) Command

The W command writes the contents of a contiguous block of memory to disk. This command requires you to first use the R command to read the data into SID-86. The W command has two forms:

Wfilespec
Wfilespec,s,f

where **filespec** is an optional pathname and the name of the file you want to receive the data. The **s** and **f** arguments are the first and last addresses of the block to be written. If you do not specify the segment in **f**, SID-86 uses the same value used for **s**.

When you use form 1, SID-86 assumes the first and last addresses from the files read with an R command. This causes all of the files read with the R command to be written. If no file is read with an R command, SID-86 responds with the message:

```
Bad command or parameter; press ? for help
```

Use form 1 for writing out files after patches are installed, assuming the overall length of the file is unchanged.

Form 2 allows you to write the contents of a specific memory block. The first address of the memory block is specified by **s** and the last address of the memory block is specified by **f**.

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

The following are examples of the W command:

#wtest.cmd Write to the file TEST.CMD the contents of the memory block read into by the most recent R command.

#wb:test.exe,40:0,3fff
Write the contents of the memory block 40:0 through 40:3FFF to the file TEST.EXE on drive B.

11.4.20 X (Examine CPU State) Command

The X command allows you to examine and alter the CPU state of the program under test. The X command has the following forms:

1. X
2. Xr
3. Xf

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

```
          AX   BX   CX   . . .   SS   ES   IP
---------xxxx xxxx xxxx   . . .  xxxx xxxx xxxx

instruction   symbol name          memory value
```

The nine hyphens at the beginning of the line indicate the state of the nine CPU flags. Each position can be either a hyphen, indicating that the corresponding flag is not set (0), or a single-character abbreviation of the flag name, indicating that the flag is set (1). The abbreviations of the flag names are shown in Table 11-2.

Table 11-1. 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

instruction is the disassembled instruction at the next location to be executed, which is indicated by the CS and IP registers. If the symbol table contains a symbol whose value is equal to one of the operands in **instruction**, the symbol name appears in the **symbol name** field, preceded by a period. If instruction references memory, the contents of the referenced location(s) appear in the **memory value** field, preceded by an equal sign. Either a byte, word, or double word value is shown, depending on the instruction. In addition to displaying the machine state, form 1 changes the values of the default segments back to the CS and DS register values, and the default offset for the L command to the IP register value.

Form 2 allows you to alter the registers in the CPU state of the program being tested. The **r** following the X is the name of one of the 16-bit CPU registers. SID-86 responds by displaying the name of the register followed by its current value. If you type a carriage return, the value of the register does not change. If you type a valid expression, the contents of the register change to the value of the expression. In either case, the next register is then displayed. This process continues until you enter a period or an invalid expression, or the last register is displayed.

Form 3 allows you to alter one of the flags in the CPU state of the program being tested. SID-86 responds by displaying the name of the flag followed by its current state. If you type a carriage return, the state of the flag does not change. If you type a valid value, the state of the flag changes to that value. You can examine or alter only one flag with each Xf command. You set or reset flags by entering a value of 1 or 0.

The following are examples of the X command.

#xbp	Change registers starting with BP.
BP=1000 **2b64**	Change BP to hex 2B64.
SI=2000 **#12345**	Change SI to decimal 12345.
CS=0040 .	Terminate X command.

11.4.21 Z (Print 8087/80287 Registers) Command

The Z command prints out the contents of 8087 and 80287 registers. The form is:

```
Z
```

The output generated by the Z command looks like the following:

```
 CW     SW     TW         IP            OP
037F   4100   FFFF   FFEF   07FF   FFFF   0000

0   EEC0   A6B6   B596   EB8A   BAD5
1   EEC0   A6B6   B55E   EB8B   BAD5
2   EEC0   A6A6   B5D6   EB8A   BAD5
3   EEC0   A6B6   B55E   EB8B   BAD5
4   EEC0   A6B6   B55E   EB8B   BAD5
5   EEC0   A6B6   B55E   EB8B   BAD5
6   EEC0   A6B6   B55E   EB8B   BAD5
7   EEC0   A6B6   B55E   EB8B   BAD5
```

Where:

CW = Control Word Format
SW = Status Word Format (indicates physical register 0)
TW = Tag Word
IP = 8087 or 80287 Instruction Pointer
OP = Pointer for last operand fetched

An error message appears if no 8087 or 80287 processor is present when the Z command is given.

11.4.22 ? (List Commands) Command

The ? command prints a list of available SID-86 commands, similar to the list appearing in Table 1-1. The form is:

```
?
```

11.4.23 ?? (List Commands Format) Command

The ?? command prints a detailed command list that includes the SID-86 commands, and the available command options. The form is:

```
??
```

11.4.24 : (Define Macro) command

The : command defines or redefines a macro. The form is:

```
:name
```

where **name** is the name of the macro. SID-86 responds with the message:

```
"Enter commands one to a line;
terminate with an empty line"
```

Enter the commands you want to execute immediately following the message.

For example, if you wish to create a macro named "s" that, when invoked, prints out the contents of the stack, you would define the macro as follows:

```
#:s
 dw ss:sp <cr>
 <cr>
```

11.4.25 = (Use Macro) Command

The = command causes SID-86 to use a previously defined macro. The forms are:

```
=
= name
```

The first form prints out the list of existing SID-86 macros and their definitions (values).

The second form executes the command associated with **name**.

For example, to invoke the "s" macro defined in the example for the : command, enter:

```
#=s
```

and the contents of the stack will print.

Table 11-2. SID-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 and symbols for execution
F	fill memory block with a constant
G	begin execution with optional breakpoints
H	hexadecimal arithmetic
I	set up program arguments
L	list memory using 8086 mnemonics
M	move memory block
P	set, clear, display pass points
Q	direct I/O request
R	read disk file into memory
S	set memory to new values
SR	search for string within memory
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
Z	dump 80287 register.
?	print list of SID commands
??	print list of SID commands with options
=	use a previously defined macro
:	define a macro

End of Section 11

Default Segment Values

12.1 Introduction

SID-86 has an internal mechanism that keeps track of the current segment value, making segment specification optional when entering a SID-86 command. SID-86 divides the command set into two types according to which segment a command defaults if you do not specify a segment value in the command line.

12.2 Type-1 Segment Value

The A (Assemble), L (List Mnemonics), P (Pass Points), and R (Read) commands use the internal type-1 segment value if you do not specify a segment value in the command.

When started, SID-86 sets the type-1 segment value to 0 and changes it when one of the following actions is taken:

- When an E command loads a file, SID-86 sets the type-1 segment value to the value of the CS:IP register.
- When an R command reads a file, SID-86 sets the type-1 segment value to the base segment where the file was read.
- When an X command changes the value of the CS:IP register, SID-86 changes the type-1 segment value to the new value of the CS:IP register.
- When SID-86 regains control from a user program after a G, T, or U command, it sets the type-1 segment value to the value of the CS:IP register.
- When an A or L command explicitly specifies a segment value, SID-86 sets the type-1 segment value to the segment value specified.

12.3 Type-2 Segment Value

The D (Display), F (Fill), M (Move), S (Set), (and SR (Search) in native mode) commands use the internal type-2 segment value if you do not specify a segment value in the command.

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

- When an E command loads a file, SID-86 sets the type-2 segment value to the value of the DS register.
- When an R command reads a file, SID-86 sets the type-2 segment value to the base segment where the file was read.
- When a D, F, M, S or SR command explicitly specifies a segment value, SID-86 sets the type-2 segment value to the segment value specified.

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

Table 13-1 summarizes the SID-86 default segment values.

Table 12-1. SID-86 Default Segment Values

Command	Default Segment Value
A	Current CS:IP of debugged process
B	Current display address
D	Current display address
E	No default values assumed
F	Current display address
G	Current CS:IP of debugged process
H	No default values assumed
I	No default values assumed
L	Current list address
M	Current display address
P	Current CS:IP of debugged process
Q	No default values assumed
R	Default is beginning address of the file
S	Current display address
SR	Current display address in native mode; no default value assumed for PC DOS mode
T	Current CS:IP of debugged process
U	Current CS:IP of debugged process
V	No default values assumed
W	Default is beginning address of the file
X	No default values assumed
Y	No default values assumed
Z	No default values assumed
:	No default values assumed
=	No default values assumed

End of Section 12

Assembly Language Syntax for A and L Commands

13.1 Assembly Language Exceptions

In general, the SID-86 A and L commands use standard 8086/80286 assembly language syntax. Several minor exceptions are listed below.

- 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 appear on a line by itself.
- The distinction between byte and word string instructions is 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 an absolute address, you enter it in the form:

  ssss:0000

  where **ssss** is the segment and **0000** is the offset of the address.

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

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

  Failure to supply a necessary prefix results in an error message.

- Operands addressing 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 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]
```

End of Section 13

SID-86 Sample Session

14.1 Introduction

The following sample session illustrates the commands and procedures used to interactively debug a simple program.

Begin the session by entering the following source file, TYP.A86. Then assemble and link the sample file to create an executable file.

```
;   RASM-86 sample for Concurrent DOS 86
;   display the contents of an ASCII file at the console

fcb     equ     5ch
eof     equ     1ah
bdosi   equ     224
conoutc equ     2
openc   equ     15
readc   equ     20
setdmac equ     26

start:  mov     dx,fcb
        call    open

loop:   call    getchr
        cmp     al,eof
        jz      done
        call    conout
        jmps    loop

done:   mov     dl,0            ;reset function
        mov     cl,0
        jmp     bdos

getchr: cmp     bptr,bsize      ;see if we need a new buffer full
        jc      get1
        call    filbuf          ;refill buffer from file

get1:   mov     bx,offset buffer
        mov     al,buffer[bx]   ;get next character from buffer
        inc     bptr            ;increment buffer pointer
        ret
```

```
filbuf: mov     dx,offset buffer
        call    setdma
        mov     dx,5ch
        call    read
        mov     bptr,0
        ret

open:   mov     cl,openc
        call    bdos
        cmp     al,0ffh
        jnz     err
        ret

setdma: mov     cl,setdmac
        jmps    bdos

read:   mov     cl,readc
        call    bdos
        cmp     al,0
        jnz     err
        ret

conout: mov     cl,conoutc
        jmps    bdos

printm: mov     cl,9
        jmps    bdos

bdos:   int     bdosi
        ret

err:    mov     dx,offset errorm
        call    printm
        jmp     done

        dseg
        org     100h

errorm  db      'ERROR',0dh,0ah,'$'
bsize   equ     80h
buffer  rs      bsize
bptr    db      bsize

        end
```

Type symbol table file produced by RASM-86.

A>**type typ.sym**

```
0000 VARIABLES
0188 BPTR       0108 BUFFER     0100 ERRORM

0000 NUMBERS
00E0 BDOSI      0080 BSIZE      0002 CONOUTC    001A EOF        005C FCB
000F OPENC      0014 READC      001A SETDMAC

0000 LABELS
0061 BDOS       0059 CONOUT     0012 DONE       0064 ERR        002F FILBUF
0023 GET1       0019 GETCHR     0006 LOOP       0041 OPEN       005D PRINTM
004F READ       004B SETDMA     0000 START
```

Try executing the program with the file TYP.A86 as data.

A>**typ typ.a86**

ERROR

The program doesn't work correctly, so load the executable program and symbol table file to find out why.

A>**sid86 typ.cmd typ.sym**

SID-86 shows the start and end addresses of each segment from the file:

```
     START      END
CS 06DA:0000 06DA:006F
DS 06E1:0000 06E1:015F
SYMBOLS
```

Display all the symbols that SID-86 loaded.

```
#h
0000 VARIABLES
0188 BPTR
0108 BUFFER
0100 ERRORM
0000 NUMBERS
00E0 BDOSI
0080 BSIZE
0002 CONOUTC
001A EOF
005C FCB
000F OPENC
```

```
0014 READC
001A SETDMAC
0000 LABELS
0061 BDOS
0059 CONOUT
0012 DONE
0064 ERR
002F FILBUF
0023 GET1
0019 GETCHR
0006 LOOP
0041 OPEN
005D PRINTM
004F READ
004B SETDMA
0000 START
```

Disassemble the beginning of the code segment.

```
#l
START:
  06DA:0000 MOV     DX,005C .FCB
  06DA:0003 CALL    0041 .OPEN
LOOP:
  06DA:0006 CALL    0019 .GETCHR
  06DA:0009 CMP     AL,1A .EOF
  06DA:000B JZ      0012 .DONE
  06DA:000D CALL    0059 .CONOUT
  06DA:0010 JMPS    0006 .LOOP
DONE:
  06DA:0012 MOV     DL,00 .VARIABLES
  06DA:0014 MOV     CL,00 .VARIABLES
  06DA:0016 JMP     0061 .BDOS
GETCHR:
  06DA:0019 CMP     BYTE [0188],80 .BPTR
  06DA:001E JB      0023 .GET1
```

Set up the default file control block at 5CH with the name of the file to process:

```
#ityp.a86
```

Trace the first two instructions of the program.

```
#t2
           AX   BX   CX   DX   SP   BP   SI   DI   IP
--------- 0000 0000 0000 0000 092C 0000 0000 0000 0000 MOV     DX,005C .FCB
--------- 0000 0000 0000 005C 092C 0000 0000 0000 0003 CALL    0041 .OPEN
```

```
*06DA:0041 .OPEN
```

SID-86 stops execution after two instructions, at the label OPEN.

Display the contents of the default fcb, to make sure it's set up right.

```
#d.fcb,+#35
06E1:005C 00 54 59 50 20 20 20 20 20 41 38 36 00 00 00 00 .TYP     A86....
06E1:006C 00 20 20 20 20 20 20 20 20 20 20 20 00 00 00 00 .           ....
06E1:007C 00 00 00 00 ....
```

The fcb looks ok. Disassemble the next few instructions.

```
#l
OPEN:
  06DA:0041 MOV     CL,0F .OPENC
  06DA:0043 CALL    0061 .BDOS
  06DA:0046 CMP     AL,FF
  06DA:0048 JNZ     0064 .ERR
  06DA:004A RET
SETDMA:
  06DA:004B MOV     CL,1A .EOF
  06DA:004D JMPS    0061 .BDOS
READ:
  06DA:004F MOV     CL,14 .READC
  06DA:0051 CALL    0061 .BDOS
  06DA:0054 CMP     AL,00 .VARIABLES
  06DA:0056 JNZ     0064 .ERR
  06DA:0058 RET
```

Continue program execution with a break point after the open function.

```
#g,46
*4BDD:0046
```

Display the CPU registers.

```
#x
            AX   BX   CX   DX   SP   BP   SI   DI   CS   DS   SS   ES   IP
--------- 0000 0000 0900 C0CF 005A 0000 0000 0000 4BDD 4BE4 381A 4BE4 0046
CMP     AL,FF
```

Registers look ok; trace a few more instructions.

```
#t2
            AX   BX   CX   DX   SP   BP   SI   DI   IP
--I------ 0000 0000 0000 0000 005A 0000 0000 0000 0046 CMP     AL,FF
--I---A-C 0000 0000 0000 0000 005A 0000 0000 0000 0048 JNZ     0064 .ERR
*4BDD:0064 .ERR
```

The code shouldn't be getting to the ERR label - the jump instruction seems to be of the wrong flavor. It should be a JZ. Rather than editing the source, just install a patch, which is easy because the new instruction is the same length as the old one. Read the file into memory (including the header).

```
#rtyp.cmd
  START      END
4C1B:0000 4C1B:02FF
```

The file was read into memory starting at paragraph 4C1B. That is where the header is - the code starts 8 paragraphs later.

```
#xcs
CS 4BDD 4c1b+8
DS 4BE4 cs
SS 381A
```

The code starts in paragraph 4C23. Using that as the base for the L command, all the symbol values will be correct.

```
#l0
START
  4C23:0000 MOV     DX,005C .FCB
  4C23:0003 CALL    0041 .OPEN
LOOP:
  4C23:0006 CALL    0019 .GETCHR
  4C23:0009 CMP     AL,1A .EOF
  4C23:000B JZ      0012 .DONE
  4C23:000D CALL    0059 .CONOUT
  4C23:0010 JMPS    0006 .LOOP
DONE:
  4C23:0012 MOV     DL,00 .VARIABLES
  4C23:0014 MOV     CL,00 .VARIABLES
  4C23:0016 JMP     0061 .BDOS
GETCHR:
  4C23:0019 CMP     BYTE [0188],80 .BPTR
  4C23:001E JB      0023 .GET1
```

Disassemble the OPEN routine.

```
#l41
OPEN:
  4C23:0041 MOV     CL,0F .OPENC
  4C23:0043 CALL    0061 .BDOS
  4C23:0046 CMP     AL,FF
  4C23:0048 JNZ     0064 .ERR
```

```
4C23:004A RET
```

Assemble patch instruction.

```
#a48

4C23:0048 jz 0064
4C23:004A .
```

Write the patched file back to disk. The start and end addresses need not be included in the W command, since the overall length of the file did not change.

```
#wtyp.cmd
```

Reload the patched file and symbols. This is needed since the R command doesn't set up registers.

```
#etyp.cmd typ.sym
      START       END
CS 4BDD:0000 4BDD:006F
DS 4BE4:0000 4BE4:018F
SYMBOLS

#ityp.a86
```

Execute the program with a break point at DONE..

```
#g,.done
ERROR
*4BDD:0012  .DONE
#^c
```

Still not correct. Invoke SID-86 again, leaving off the file types, since SID-86 uses the appropriate defaults.

A>sid86 typ typ

```
      START       END
CS 4BDD:0000 4BDD:006F
DS 4BE4:0000 4BE4:018F
SYMBOLS
```

Set up default file control block.

```
#ityp.a86
```

Disassemble start of code segment.

```
#l
START
  4BDD:0000 MOV     DX,005C .FCB
  4BDD:0003 CALL    0041 .OPEN
LOOP:
  4BDD:0006 CALL    0019 .GETCHR
  4BDD:0009 CMP     AL,1A .EOF
  4BDD:000B JZ      0012 .DONE
  4BDD:000D CALL    0059 .CONOUT
  4BDD:0010 JMPS    0006 .LOOP
DONE:
  4BDD:0012 MOV     DL,00 .VARIABLES
  4BDD:0014 MOV     CL,00 .VARIABLES
  4BDD:0016 JMP     0061 .BDOS
GETCHR:
  4BDD:0019 CMP     BYTE [0188],80 .BPTR
  4BDD:001E JB      0023 .GET1
```

Trace without call, so SID-86 doesn't trace the OPEN routine, which should be fixed.

```
#tw2
            AX   BX   CX   DX   SP   BP   SI   DI   IP
--I------ 0000 0000 0000 0000 005C 0000 0000 0000 0000 MOV     DX,005C .FCB
--I------ 0000 0000 0000 005C 005C 0000 0000 0000 0003 CALL    0041 .OPEN
*4BDD:0006 .LOOP
```

Disassemble next few instructions.

```
#l
LOOP:
  4BDD:0006 CALL    0019 .GETCHR
  4BDD:0009 CMP     AL,1A .EOF
  4BDD:000B JZ      0012 .DONE
  4BDD:000D CALL    0059 .CONOUT
  4BDD:0010 JMPS    0006 .LOOP
DONE:
  4BDD:0012 MOV     DL,00 .VARIABLES
  4BDD:0014 MOV     CL,00 .VARIABLES
  4BDD:0016 JMP     0061 .BDOS
GETCHR:
  4BDD:0019 CMP     BYTE [0188],80 .BPTR
  4BDD:001E JB      0023 .GET1
  4BDD:0020 CALL    002F .FILBUF
GET1:
  4BDD:0023 MOV     BX,0108 .BUFFER
```

Trace without call next three instructions, to see if this sequence is working.

```
#tw3
            AX   BX   CX   DX   SP   BP   SI   DI   IP
--I---A-C 0000 0000 0000 0000 005C 0000 0000 0000 0006 CALL   0019 .GETCHR
--I------ 00ED 0108 0000 0000 005C 0000 0000 0000 0009 CMP    AL,1A .EOF
--I-S-A-C 0000 0108 0000 0000 005C 0000 0000 0000 000B JZ     0012 .DONE
*4BDD:000D
```

GETCHR is returning a 000H - something must be wrong there. Use the E command to bring in a fresh copy of the program.

```
#etyp typ
     START       END
CS 4BDD:0000 4BDD:006F
DS 4BE4:0000 4BE4:018F
SYMBOLS
```

Disassemble code segment.

```
#l
START
  4BDD:0000 MOV     DX,005C .FCB
  4BDD:0003 CALL    0041 .OPEN
LOOP:
  4BDD:0006 CALL    0019 .GETCHR
  4BDD:0009 CMP     AL,1A .EOF
  4BDD:000B JZ      0012 .DONE
  4BDD:000D CALL    0059 .CONOUT
  4BDD:0010 JMPS    0006 .LOOP
DONE:
  4BDD:0012 MOV     DL,00 .VARIABLES
  4BDD:0014 MOV     CL,00 .VARIABLES
  4BDD:0016 JMP     0061 .BDOS
GETCHR:
  4BDD:0019 CMP     BYTE [0188],80 .BPTR
  4BDD:001E JB      0023 .GET1
```

Disassemble GETCHR routine.

```
#l
GETCHR:
  4BDD:0019 CMP     BYTE [0188],80 .BPTR
  4BDD:001E JB      0023 .GET1
  4BDD:0020 CALL    002F .FILBUF
GET1:
  4BDD:0023 MOV     BX,0108 .BUFFER
  4BDD:0026 MOV     AL,0108[BX] .BUFFER
  4BDD:002A INC     BYTE [0188] .BPTR
  4BDD:002E RET
```

```
FILBUF:
  4BDD:002F MOV     DX,0108 .BUFFER
  4BDD:0032 CALL    004B .SETDMA
  4BDD:0035 MOV     DX,005C .FCB
  4BDD:0038 CALL    004F .READ
  4BDD:003B MOV     BYTE [0188],00 .BPTR
```

Trace first few instructions. Note that SID-86 shows the contents of BPTR as is it being compared.

```
#t2
          AX   BX   CX   DX   SP   BP   SI   DI   IP
--I---A-C 0000 0000 0000 0000 005A 0000 0000 0000 0019 CMP     BYTE [0188],80 .BPTR =80
--I--Z-P- 0000 0000 0000 0000 005A 0000 0000 0000 001E JB      0023 .GET1
*4BDD:0020
```

The compare worked ok; keep going.

```
#t
          AX   BX   CX   DX   SP   BP   SI   DI   IP
--I--Z-P- 0000 0000 0000 0000 005A 0000 0000 0000 0020 CALL    002F .FILBUF
*4BDD:002F .FILBUF
```

See what FILBUF looks like.

```
#l
FILBUF:
  4BDD:002F MOV     DX,0108 .BUFFER
  4BDD:0032 CALL    004B .SETDMA
  4BDD:0035 MOV     DX,005C .FCB
  4BDD:0038 CALL    004F .READ
  4BDD:003B MOV     BYTE [0188],80 .BPTR
  4BDD:0040 RET
OPEN:
  4BDD:0041 MOV     CL,0F .OPENC
  4BDD:0043 CALL    0061 .BDOS
  4BDD:0046 CMP     AL,FF
  4BDD:0048 JZ      0064 .ERR
  4BDD:004A RET
SETDMA:
  4BDD:004B MOV     CL,1A .EOF
```

See what's in the buffer before the disk read takes place

```
#d.buffer
4BE4:0108 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
4BE4:0118 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
4BE4:0128 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
4BE4:0138 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

```
4BE4:0148 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00 ................
4BE4:0158 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00 ................
4BE4:0168 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ........ ........
4BE4:0178 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
4BE4:0188 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
4BE4:0198 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
4BE4:01A8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
4BE4:01B8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ........ ........
```

Trace the FILBUF routine

```
#twe
            AX   BX   CX   DX   SP   BP   SI   DI   IP
--I--Z-P- 0000 0000 0000 0000 0058 0000 0000 0000 002F MOV    DX,0108 .BUFFER
--I--Z-P- 0000 0000 0000 0108 0058 0000 0000 0000 0032 CALL   004B .SETDMA
-----Z-P- 0000 0000 0000 0108 0058 0000 0000 0000 0035 MOV    DX,005C .FCB
-----Z-P- 0000 0000 0000 005C 0058 0000 0000 0000 0038 CALL   004F .READ
-----Z-P- 0000 0000 0000 0000 0058 0000 0000 0000 003B MOV    BYTE [0188],00 .BPTR =0
-----Z-P- 0000 0000 0000 0000 0058 0000 0000 0000 0040 RET
*4BDD:0023 .GET1
```

See what's in the buffer after the read

```
#d.buffer
4BE4:0108 0D 0A 3B 20 09 52 41 53 4D 20 38 36 20 73 61 6D ..; .RASM-86 sam
4BE4:0118 70 6C 65 20 66 6F 72 20 43 6F 6E 63 75 72 72 65 ple for Concurre
4BE4:0128 6E 74 20 44 4F 53 20 38 36 0D 0A 3B 20 20 20 20 nt DOS 86..;
4BE4:0138 20 20 20 20 64 69 73 70 6C 61 79 20 74 68 65 20     display the
4BE4:0148 6F 6E 74 65 6E 74 73 20 6F 66 20 61 6E 20 41 53 ontents of an AS
4BE4:0158 43 49 49 20 66 69 6C 65 20 61 74 20 74 68 65 20 CII file at the
4BE4:0168 63 6F 6E 73 6F 6C 65 0D 0A 0D 0A 66 63 62 09 65 console....fcb.e
4BE4:0178 71 75 09 35 63 68 0D 0A 65 6F 66 09 65 71 75 09 qu.5ch..eof.equ.
4BE4:0188 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
4BE4:0198 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
4BE4:01A8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
4BE4:01B8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

Looks like good data in the buffer. See what's next.

```
#u
GET1:
  4BDD:0023 MOV     BX,0108 .BUFFER
  4BDD:0026 MOV     AL,0108[BX] .BUFFER
  4BDD:002A INC     BYTE [0188] .BPTR
  4BDD:002E RET
FILBUF:
  4BDD:002F MOV     DX,0108 .BUFFER
  4BDD:0032 CALL    004B .SETDMA
  4BDD:0035 MOV     DX,005C .FCB
```

```
  4BDD:0038 CALL    004F .READ
  4BDD:003B MOV     BYTE [0188],00 .BPTR
OPEN:
  4BDD:0041 MOV     CL,0F .OPNEC
  4BDD:0043 CALL    0061 .BDOS
```

Trace the code getting the next character from the buffer.

```
#t4
             AX   BX   CX   DX   SP   BP   SI   DI   IP
--I--Z-P- 0000 0000 0000 0000 005A 0000 0000 0000 0023 MOV    BX,0108 .BUFFER
--I--Z-P- 0000 0108 0000 0000 005A 0000 0000 0000 0026 MOV    AL,0108[BX] .BUFFER =ED
--I--Z-P- 0000 0108 0000 0000 005A 0000 0000 0000 002A INC    BYTE [0188] .BPTR =00
--I------ 0000 0108 0000 0000 005A 0000 0000 0000 002E RET
*4BDD:0009
```

It's getting the wrong data, because BX should have the contents of BPTR in it, rather than the address of the buffer. Leave SID-86 and edit the file (be sure to include the fix that was patched earlier). After editing, reassemble and relink

^C

Try it again.

A>typ typ.a86 ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

Still no good.

A>sid86 typ typ

```
CS 4BDD:0000 4BDD:006F
DS 4BE4:0000 4BE4:018F
SYMBOLS

#itvp.a86
#l
START
  4BDD:0000 MOV     DX,005C .FCB
  4BDD:0003 CALL    0044 .OPEN
LOOP:
  4BDD:0006 CALL    0019 .GETCHR
  4BDD:0009 CMP     AL,1A .EOF
  4BDD:000B JZ      0012 .DONE
  4BDD:000D CALL    005C .CONOUT
  4BDD:0010 JMPS    0006 .LOOP
```

```
DONE:
  4BDD:0012 MOV     DL,00 .VARIABLES
  4BDD:0014 MOV     CL,00 .VARIABLES
  4BDD:0016 JMP     0064 .BDOS
GETCHR:
  4BDD:0019 CMP     BYTE [0188],80 .BPTR
  4BDD:001E JB      0023 .GETL
```

Trace at the top level.

```
#t#5
             AX   BX   CX   DX   SP   BP   SI   DI   IP
--I------ 0000 0000 0000 0000 005C 0000 0000 0000 0000 MOV    DX,005C .FCB
--I------ 0000 0000 0000 005C 005C 0000 0000 0000 0003 CALL   0044 .OPEN
LOOP:
--I---A-C 0000 0000 0000 0000 005C 0000 0000 0000 0006 CALL   0019 .GETCHR
--I------ 0000 0000 0000 0000 005C 0000 0000 0000 0009 CMP    AL,1A .EOF
--I-S--PC 0000 0000 0000 0000 005C 0000 0000 0000 000B JZ     0012 .DONE
*4BDD:000D
```

Got the right data from GETCHR.

```
#l
  4BDD:000D CALL    005C .CONOUT
  4BDD:0010 JMPS    0006 .LOOP
DONE:
  4BDD:0012 MOV     DL,00 .VARIABLES
  4BDD:0014 MOV     CL,00 .VARIABLES
  4BDD:0016 JMP     0064 .BDOS
GETCHR:
  4BDD:0019 CMP     BYTE [0188],80 .BPTR
  4BDD:001E JB      0023 .GETL
  4BDD:0020 CALL    0032 .FILBUF
GETL:
  4BDD:0023 MOV     BL,[0188] .BPTR
  4BDD:0027 MOV     BH,00 .VARIABLES
  4BDD:0029 MOV     AL,0108[BX] .BUFFER
  4BDD:002D INC     BYTE [0188] .BPTR

#t
             AX   BX   CX   DX   SP   BP   SI   DI   IP
--I-S--PC 0000 0000 0000 0000 005C 0000 0000 0000 000D CALL   005C .CONOUT
*4BDD:005C .CONOUT

#l
CONOUT:
  4BDD:005C MOV     CL,02 .CONOUTC
  4BDD:005E JMPS    0064 .BDOS
```

```
PRINTM:
  4BDD:0060 MOV     CL,09
  4BDD:0062 JMPS    0064 .BDOS
BDOS:
  4BDD:0064 INT     E0 .BDOSI
  4BDD:0066 RET
ERR:
  4BDD:0067 MOV     DX,0100 .ERRORM
  4BDD:006A CALL    0060 .PRINTM
  4BDD:006D JMP     0012 .DONE
  4BDD:0070 ??=     6F
  4BDD:0071 ADD     [BX+SI],AL
  4BDD:0073 ESC     10,[8F00]
```

At this point, the data to output to the console should be in DL, but it's not. Edit the file again, then reassemble and relink.

 # ^C

Now try one more time.

A>typ typ.a86

```
;   RASM-86 sample for Concurrent DOS 86
;   display the contents of an ASCII file at the console

fcb     equ     5ch
eof     equ     1ah
bdosi   equ     224
conoutc equ     2
openc   equ     15
readc   equ     20
setdmac equ     26

start:  mov     dx,fcb
        call    open

loop:   call    getchr
        cmp     al,eof
        jz      done
        mov     dl,al           ;prepare for conout
        call    conout
        jmps    loop

done:   mov     dl,0            ;reset function
        mov     cl,0
        jmp     bdos
```

```
getchr: cmp     bptr,bsize      ;see if we need a
                                ;new buffer full
        jc      getl
        call    filbuf          ;refill buffer from file

getl:   mov     bl,bptr
        mov     bh,0
        mov     al,buffer[bx]   ;get next character from buffer
        inc     bptr            ;increment buffer pointer
        ret

filbuf: mov     dx,offset buffer
        call    setdma
        mov     dx,5ch
        call    read
        mov     bptr,0
        ret

open:   mov     cl,openc
        call    bdos
        cmp     al,0ffh
        jz      err
        ret

setdma: mov     cl,setdmac
        jmps    bdos

read:   mov     cl,readc
        call    bdos
        cmp     al,0
        jnz     err
        ret

conout: mov     cl,conoutc
        jmps    bdos

printm: mov     cl,9
        jmps    bdos

bdos:   int     bdosi
        ret

err:    mov     dx,offset errorm
        call    printm
        jmp     done

        dseg
```

```
errorm  db      'ERROR',0dh,0ah,'$'
bsize   equ     80h
buffer  rs      bsize
bptr    db      bsize

        end
```

Seems to work now. Just out of curiosity, let's find out how many characters are in the file by setting a pass point with a high count at CONOUT.

A>sid86 typ typ

```
CS 4BDD:0000 4BDD:007F
DS 4BE5:0000 4BE5:018F
#ityp.a86
```

Hopefully there are fewer than 65535 characters!

```
#p.conout,ffff
```

Begin execution, with a break point at DONE.

```
#g,.done
FFFF PASS 4BDD:005E .CONOUT
--I-S--PC 000D 0000 0000 000D 005A 0000 0000 0000 005E MOV    CL,02 .CONOUTC
FFFE PASS 4BDD:005E .CONOUT
--I-S--PC 100A 0001 0000 000A 005A 0000 0000 0000 005E MOV    CL,02 .CONOUTC

FFFD PASS 4BDD:005E .CONOUT
--I----P- 103B 0002 0000 003B 005A 0000 0000 0000 005E MOV    CL,02 .CONOUTC;
FFFC PASS 4BDD:005E .CONOUT
--I---AP- 1020 0003 0000 0020 005A 0000 0000 0000 005E MOV    CL,02 .CONOUTC
FFFB PASS 4BDD:005E .CONOUT
--I-S-A-C 1009 0004 0000 0009 005A 0000 0000 0000 005E MOV    CL,02 .CONOUTC
FFFA PASS 4BDD:005E .CONOUT
--I---A-- 1052 0005 0000 0052 005A 0000 0000 0000 005E MOV    CL,02 .CONOUTCR
FFF9 PASS 4BDD:005E .CONOUT
--I---AP- 1041 0006 0000 0041 005A 0000 0000 0000 005E MOV    CL,02 .CONOUTCA
FFF8 PASS 4BDD:005E .CONOUT
--I---AP- 1053 0007 0000 0053 005A 0000 0000 0000 005E MOV    CL,02 .CONOUTCS
FFF7 PASS 4BDD:005E .CONOUT
--I----P- 104D 0008 0000 004D 005A 0000 0000 0000 005E MOV    CL,02 .CONOUTCM
FFF6 PASS 4BDD:005E .CONOUT
--I------ 102D 0009 0000 002D 005A 0000 0000 0000 005E MOV    CL,02 .CONOUTC-
FFF5 PASS 4BDD:005E .CONOUT
--I---AP- 1038 000A 0000 0038 005A 0000 0000 0000 005E MOV    CL,02 .CONOUT8
Stop command? (Y/N) y
```

This is too messy. Using the -G command suppresses the display of the CPU registers until the pass count is 1.

```
#-g,.done

        display the contents of an ASCII file at the console

fcb     equ     5ch
eof     equ     1ah
bdosi   equ     224
conoutc equ     2
openc   equ     15
readc   equ     20
setdmac equ     26

start:  mov     dx,fcb
        call    open

loop:   call    getchr
        cmp     al,eof
        jz      done
        mov     dl,al           ;prepare for conout
        call    conout
        jmps    loop

done:   mov     dl,0            ;reset function
        mov     cl,0
        jmp     bdos

getchr: cmp     bptr,bsize      ;see if we need a
                                ;new buffer full
        jc      getl
        call    filbuf          ;refill buffer from file

getl:   mov     bl,bptr
        mov     bh,0
        mov     al,buffer[bx]   ;get next character from buffer
        inc     bptr            ;increment buffer pointer
        ret

filbuf: mov     dx,offset buffer
        call    setdma
        mov     dx,5ch
        call    read
        mov     bptr,0
        ret
```

```
open:   mov     cl,openc
        call    bdos
        cmp     al,0ffh
        jz      err
        ret

setdma: mov     cl,setdmac
        jmps    bdos

read:   mov     cl,readc
        call    bdos
        cmp     al,0
        jnz     err
        ret

conout: mov     cl,conoutc
        jmps    bdos

printm: mov     cl,9
        jmps    bdos

bdos:   int     bdosi
        ret

err:    mov     dx,offset errorm
        call    printm
        jmp     done

        dseg

errorm  db      'ERROR',0dh,0ah,'$'
bsize   equ     80h
buffer  rs      bsize
bptr    db      bsize

        end
```

Reached the break point at DONE

```
*4BDD:0014 .DONE
```

Display the currently active pass points.

```
#p
FB67 Pass 4BDD:005E .CONOUT
```

The pass count went from FFFF to FB67, so the difference is the number of times CONOUT was called, or the number of characters in

the file. The H command performs the subtraction, and displays the result in decimal.

```
#hffff-fb67
4BDD:0498 #1167
#
```

Thus, there are 1176 characters in the file.

```
# ^C
```

End of Section 14

RASM-86 Example Source File

This example RASM-86 source files is provided to illustrate some of the characteristics unique to RASM-86.

RASM-86 Sample (for Concurrent DOS 86

```
;       display the contents of an ASCII file at the console

fcb     equ     5ch
eof     equ     1ah
bdosi   equ     224
conoutc equ     2
openc   equ     15
readc   equ     20
setdmac equ     26
bsize   equ     80h

cseg

start:  mov     dx,fcb
        call    open

loop:   call    getchr
        cmp     dl,eof
        jz      done
        call    conout
        jmps    loop

done:   mov     dl,0            ;reset function
        mov     cl,0
        jmp     bdos

getchr: cmp     bptr,bsize      ;see if we need a
                                ;new buffer full

        jc      getl
        call    filbuf          ;refill buffer from file
```

```
getl:   mov     bx,offset buffer
        mov     si,bptr         ;get next character from buffer
        mov     dl,byte ptr[bx+si]
        inc     bptr
        ret

filbuf: mov     dx,offset buffer
        call    setdma
        mov     dx,fcb
        call    read
        mov     bptr,0
        ret

open:   mov     cl,openc
        call    bdos
        cmp     al,0ffh
        jz      err
        ret

setdma: mov     cl,setdmac
        jmps    bdos

read:   mov     cl,readc
        call    bdos
        cmp     al,0
        jnz     err
        ret

conout: mov     cl,conoutc
        jmps    bdos

printm: mov     cl,9
        jmps    bdos

bdos:   int     bdosi
        ret

err:    mov     dx,offset errorm
        call    printm
        jmp     done
```

```
        dseg

bptr    dw      bsize
errorm  db      'ERROR',0dh,0ah,'$'
buffer  rb      bsize

        end
```

End of Appendix A

Mnemonic Differences from the Intel Assembler

RASM-86 uses the same instruction mnemonics as the Intel 8086 assembler except for explicitly specifying far and short jumps, calls, and returns. Table B-1 shows the four differences:

Table B-1. RASM-86/Intel Mnemonic Differences

Mnemonic Function	RASM-86	Intel
Intra-segment short jump:	JMPS	JMP SHORT
Inter-segment jump:	JMPF	JMP
Inter-segment return:	RETF	RET
Inter-segment call:	CALLF	CALL

RASM-86 also uses a different method than Intel for specifying the size of memory operands for 8087 instructions. Intel associates the size with the operand, RASM-86 places it in the instruction. Table B-2 shows the differences:

Table B-2. Memory Operands for 8087 Instruction

RASM-86	Intel	
FLD32	FLD	SYM 32
FLD64	FLD	SYM 64
FLD80	FLD	SYM 80
FST32	FST	
FST64	FST	
FST80	FST	
FILD16	FILD	
FILD32	FILD	
FILD64	FILD	
FIST16	FIST	
FIST32	FIST	
FIST64	FIST	

End of Appendix B

Reserved Words

Table C-1. Reserved Words

Predefined Numbers				
BYTE	WORD	DWORD		
Operators				
AND	LAST	MOD	OR	SHR
EQ	LE	NE	PTR	TYPE
GE	LENGTH	NOT	SEG	XOR
GT	LT	OFFSET	SHL	
Assembler Directives				
AUTO8087	CODEMACRO	ELSE	GROUP	NOIFLIST
RD	ENDIF	END	HARD8087	NOLIST
RS	DB	ENDIF	IF	ORG
RW	DD	ENDM	IFLIST	PAGESIZE
SIMFORM	DSEG	EQU	INCLUDE	PAGEWIDTH
DW	ESEG	LIST	PUBLIC	SSEG
EJECT	EXTRN	NAME	RB	TITLE
Code-macro Directives				
DB	DD	MODRM	RELB	SEGFIX
DBIT	DW	NOSEGFIX	RELW	
8086 Registers				
AH	BL	CL	DI	ES
AL	BP	CS	DL	SI
AX	BX	CX	DS	SP
BH	CH	DH	DX	SS

Table C-1. (Continued)

8087 Registers

ST	ST0	ST1	ST2	ST3
	ST4	ST5	ST6	ST7

Default Segment Names

CODE	DATA	EXTRA	STACK

Segment Descriptors

BYTE	LOCAL	PARA	STACK
COMMON	PAGE	PUBLIC	WORD

External Descriptors

ABS	DWORD	NEAR
BYTE	FAR	WORD

Instruction Mnemonics - See Section 13

End of Appendix C

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 | NUMBER8

<double$range> ::= NUMBER8,NUMBER8 | NUMBER8,REGISTER |
                   REGISTER,NUMBER8 | REGISTER,REGISTER

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

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

<db> ::= DB NUMBER8 | DB <form$name>

<dw> ::= DW NUMBER16 | 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 D

RASM-86 Error Messages

RASM-86 displays two kinds of error messages:

- nonrecoverable errors
- diagnostics

Nonrecoverable errors occur when RASM-86 is unable to continue assembling. Table E-1 lists the non-recoverable errors RASM-86 can encounter during assembly.

Table E-1. RASM-86 Non-recoverable Errors

Error Message	Cause
NO FILE	RASM-86 cannot find the indicated source or INCLUDE file on the indicated drive.
DISK FULL	There is not enough disk space for the output files. You should either erase some unnecessary files or get another disk with more room and run RASM-86 again.
DIRECTORY FULL	There is not enough directory space for the output files. You should either erase some unnecessary files or get another disk with more directory space and run RASM-86 again.

Table E-1. (Continued)

Error Message	Cause
DISK READ ERROR	RASM-86 cannot properly read a source or INCLUDE file. This is usually the result of an unexpected end-of-file. Correct the problem in your source file.
CANNOT CLOSE	RASM-86 cannot close an output file. You should take appropriate action after checking to see if the correct disk is in the drive and the disk is not write-protected.
SYMBOL TABLE OVERFLOW	There is not enough memory for the Symbol Table. Either reduce the length or number of symbols, or reassemble on a system with more memory.
SYNTAX ERROR	A parameter in the command tail of the RASM-86 command was specified incorrectly.

Diagnostic messages report problems with the syntax and semantics of the program being assembled. When RASM-86 detects an error in the source file, it places a numbered ASCII error message in the listing file in front of the line containing the error. If there is more than one error in the line, only the first one is reported. Table E-2 shows the RASM-86 diagnostic error messages by number and gives a brief explanation of the error.

Table E-2. RASM-86 Diagnostic Error Messages

Error Message	Cause
ERROR NO: 0	ILLEGAL FIRST ITEM The first item on a source line is not a valid identifier, directive, or mnemonic. For example, `1234H`
ERROR NO: 1	MISSING PSEUDO INSTRUCTION The first item on a source line is a valid identifier, and the second item is not a valid directive that can be preceded by an identifier. For example, `THIS IS A MISTAKE`
ERROR NO: 2	ILLEGAL PSEUDO INSTRUCTION Either a required identifier in front of a pseudo instruction is missing, or an identifier appears before a pseudo instruction that does not allow an identifier.

ERROR NO: 3	DOUBLE DEFINED VARIABLE
	An identifier used as the name of a variable is used elsewhere in the program as the name of a variable or label. For example,
	`X DB 5`
	`. . .`
	`X DB 123H`

Table E-2. (Continued)

Error Message	Cause

ERROR NO: 4	**DOUBLE DEFINED LABEL**
	An identifier used as a label is used elsewhere in the program as a label or variable name. For example,
	`LAB3: MOV BX,5`
	`. . .`
	`LAB3: CALL MOVE`

ERROR NO: 5	**UNDEFINED INSTRUCTION** The item following a label on a source line is not a valid instruction. For example, `DONE: BAD INSTR`

ERROR NO: 6	**GARBAGE AT END OF LINE - IGNORED**
	Additional items were encountered on a line when RASM-86 was expecting an end of line. For example,
	`NOLIST 4`
	`MOV AX,4 RET`

ERROR NO: 7	**OPERAND(S) MISMATCH INSTRUCTION**
	Either an instruction has the wrong number of operands, or the types of the operands do not match. For example,
	`MOV CX,1,2`
	`X DB 0`
	`MOV AX,X`

Table E-2. (Continued)

Error Message	Cause

ERROR NO: 8	**ILLEGAL INSTRUCTION OPERANDS**
	An instruction operand is improperly formed. For example,
	`MOV [BP+SP],1234`
	`CALL BX+1`

ERROR NO: 9	**MISSING INSTRUCTION** A prefix on a source line is not followed by an instruction. For example, `REPNZ`

ERROR NO: 10	**UNDEFINED ELEMENT OF EXPRESSION**
	An identifier used as an operand is not defined or has been illegally forward referenced. For example,
	`JMP X`
	`A EQU B`
	`B EQU 5`
	`MOV AL,B`

ERROR NO: 11	**ILLEGAL PSEUDO OPERAND**
	The operand in a directive is invalid. For example,
	`X EQU 0AGH`
	`TITLE UNQUOTED STRING`

ERROR NO: 12	**NESTED IF ILLEGAL - IF IGNORED** The maximum nesting level for IF statements has been exceeded.

Table E-2. (Continued)

Error Message.	Cause
ERROR NO: 13	**ILLEGAL IF OPERAND - IF IGNORED** Either the expression in an IF statement is not numeric, or it contains a forward reference.
ERROR NO: 14	**NO MATCHING IF FOR ENDIF** An ENDIF statement was encountered without a matching IF statement.
ERROR NO: 15	**SYMBOL ILLEGALLY FORWARD REFERENCED - NEGLECTED** The indicated symbol was illegally forward referenced in an ORG, RS, EQU or IF statement.
ERROR NO: 16	**DOUBLE DEFINED SYMBOL - TREATED AS UNDEFINED** The identifier used as the name of an EQU directive is used as a name elsewhere in the program.
ERROR NO: 17	**INSTRUCTION NOT IN CODE SEGMENT** An instruction appears in a segment other than a CSEG.
ERROR NO: 18	**FILE NAME SYNTAX ERROR** The filename in an INCLUDE directive is improperly formed. For example, `INCLUDE FILE.A86X`
ERROR NO: 19	**NESTED INCLUDE NOT ALLOWED** An INCLUDE directive was encountered within a file already being included.

Table E-2. (Continued)

Error Message	Cause
ERROR NO: 20	ILLEGAL EXPRESSION ELEMENT

	An expression is improperly formed. For example,
	`X DB 12X`
	`DW (4 * )`

ERROR NO: 21	MISSING TYPE INFORMATION IN OPERAND(S)
	Neither instruction operand contains sufficient type information. For example, `MOV [BX],10`
ERROR NO: 22	LABEL OUT OF RANGE
	The label referred to in a call, jump, or loop instruction is out of range. The label can be defined in a segment other than the segment containing the instruction. In the case of short instructions (JMPS, conditional jumps, and loops), the label is more than 128 bytes from the location of the following instruction.
ERROR NO: 23	MISSING SEGMENT INFORMATION IN OPERAND

	The operand in a CALLF or JMPF instruction (or an expression in a DD directive) does not contain segment information. The required segment information can be supplied by including a numeric field in the segment directive as shown:
	`CSEG 1000H`
	`X:`
	`. . .`
	`JMPF X`
	`DD X`

Table E-2. (Continued)

Error Message	Cause
ERROR NO: 24	**ERROR IN CODEMACRO BUILDING** Either a code-macro contains invalid statements, or a code-macro directive was encountered outside a code-macro.
ERROR NO: 25	**NO MATCHING IF FOR ELSE** An ELSE statement was encountered without a matching IF statement.
ERROR NO: 26	**NO MATCHING ENDIF FOR IF** An IF statement was encountered without a matching ENDIF statement.
ERROR NO: 27	**"HARD8087" USED AFTER FLOATING INSTRUCTION** The HARD8087 directive cannot be specified after a floating point instruction.
ERROR NO: 28	**ATTEMPT TO USE 186/286 INSTRUCTIONS WITHOUT SWITCH** 80186 or 80286 instructions were encountered and the corresponding RASM-86 run-time parameter (186 or 286) was not specified on the RASM-86 command line.
ERROR NO: 29	**Command included not used in source file** The command defined in the file included, via the INCLUDE command, in the RASM-86 source file is not used by the source file.

End of Appendix E

XREF-86 Error Messages

During the course of operation, XREF-86 can display error messages. Table F-1 shows the error messages and a brief explanation of their cause.

Table F-1. XREF-86 Error Messages

Error Message	Meaning
CANNOT CLOSE	XREF-86 cannot close an output file. You should take appropriate action after checking to see if the correct disk is in the drive and the disk is not write-protected.
DIRECTORY FULL	There is not enough directory space for the output files. You should either erase some unnecessary files or get another disk with more directory space and run XREF-86 again.
DISK FULL	There is not enough disk space for the output files. You should either erase some unnecessary files or get another disk with more room and run XREF-86 again.
NO FILE	XREF-86 cannot find the indicated file on the indicated drive.

Table F-1. (Continued)

Error Message	Meaning
SYMBOL FILE ERROR	XREF-86 issues this message when it reads an invalid SYM file. Specifically, a line in the SYM file not terminated with a carriage return line-feed causes this error message.
SYMBOL TABLE OVERFLOW	XREF-86 ran out of Symbol Table space. Either reduce the number or length of symbols in the program, or rerun on a system with more memory.
LATER CP/M OR MP/M VERSION REQUIRED	XREF-86 requires a more recent version of the operating system.

End of Appendix F

LINK-86 Error Messages

During the course of operation, LINK-86 can display error messages. The error messages and a brief explanation of their cause are listed below.

Table G-1. LINK-86 Error Messages

Message	Meaning
ALIGN TYPE NOT IMPLEMENTED	The object file contains a segment align type not implemented in LINK-86.
CANNOT CLOSE	LINK-86 cannot close an output file. Check to see if the correct disk is in the drive and the disk is not write-protected or full.
CLASS NOT FOUND	The class name specified in the command line does not appear in any of the files linked.
COMBINE TYPE NOT IMPLEMENTED	The object file contains a segment align type not implemented in LINK-86.
COMMAND TOO LONG	The total length of input to LINK-86, including the input file, cannot exceed 2048 characters.

Table G-1. (Continued)

Message	Meaning
DIRECTORY FULL	There is not enough directory space for the output files. You should either erase some unnecessary files or get another disk with more directory space and run LINK-86 again.
DISK READ ERROR	LINK-86 cannot properly read a source or object file. This is usually the result of an unexpected end-of-file character. Correct the problem in your source file.
DISK WRITE ERROR	A file cannot be written properly; the disk is probably full.
FIXUP TYPE NOT IMPLEMENTED	The object file uses a fixup type not implemented in LINK-86. Make sure the object file has not been corrupted.
GROUP NOT FOUND	The group name specified in the command line does not appear in any of the files linked.
GROUP OVER 64K	The group listed must be made smaller than 64k before relinking. Either delete segments from the group, split it up into 2 or more groups or do not use groups.
GROUP TYPE NOT IMPLEMENTED	LINK-86 only supports segments as elements of a group.

Table G-1. (Continued)

Message	Meaning
INVALID LIBRARY-REQUESTED SUFFIX	The command file suffix requested by a library is not supported. Verify that the correct library is being used.
LINK-86 ERROR 1	This error indicates an inconsistency in the LINK-86 internal tables, and should never be emitted.
MULTIPLE DEFINITION	The indicated symbol is defined as PUBLIC in more than one module. Correct the problem in the source file, and try again.
MORE THAN ONE MAIN PROGRAM	A program linked by LINK-86 may have at most one main program.
NO FILE	LINK-86 cannot find the indicated source or object file on the indicated drive.
OBJECT FILE ERROR	LINK-86 detected an error in the object file. This is caused by a translator error or by a bad disk file. Try regenerating the file.
RECORD TYPE NOT IMPLEMENTED	The object file contains a record type not implemented in LINK-86. Make sure the object file has not been corrupted by regenerating it and linking again.
SEGMENT OVER 64K	The segment listed after the error message has a total length greater than 64k bytes. Make the segment smaller, or do not combine it with other PUBLIC segments of the same name.

Table G-1. (Continued)

Message	Meaning
SEGMENT CLASS ERROR	The class of a segment must be CODE, DATA, STACK, EXTRA, X1*, X2*, X3*, or X4*.[1]
SEGMENT ATTRIBUTE ERROR	The Combine type of the indicated segment is not the same as the type of the segment in a previously linked file. Regenerate the object file after changing the segment attributes as needed.
SEGMENT COMBINATION ERROR	An attempt is made to combine segments that cannot be combined, such as LOCAL segments. Change the segment attributes and relink.
SEGMENT NOT FOUND	The segment name specified in the command line does not appear in any of the files linked.
SYMBOL TABLE OVERFLOW	LINK-86 ran out of Symbol Table space. Either reduce the number or length of symbols in the program, or relink on a system with more memory.
SYNTAX ERROR	LINK-86 detected a syntax error in the command line; the error is probably an improper filename or an invalid command option. LINK-86 echoes the command line up to the point where it found the error. Retype the command line or edit the INP file.

[1] * native-mode only

Table G-1. (continued)

Message	Meaning
TARGET OUT OF RANGE	The target of a fixup cannot be reached from the location of the fixup.
TOO MANY MODULES IN LIBRARY	The library contains more modules than LINK-86 can handle. Split the library up into 2 or more libraries and relink.
TOO MANY MODULES LINKED FROM LIBRARY	A library may supply a maximum of 256 modules during 1 execution of LINK-86. Split the library up into 2 or more smaller libraries.
UNDEFINED SYMBOLS	The symbols following this message are referenced but not defined in any of the modules being linked.
VERSION 2 REQUIRED	LINK-86 needs a version 2 or later file system because its uses random disk I/O functions.

End of Appendix G

LIB-86 Error Messages

LIB-86 can produce the following error messages during processing. With each message, LIB-86 displays additional information appropriate to the error, such as the filename or module name, to help isolate the location of the problem.

Table H-1. LIB-86 Error Messages

Message	Meaning
CANNOT CLOSE	LIB-86 cannot close an output file. You should take appropriate action after checking to see if the correct disk is in the drive and the disk is not write-protected.
DIRECTORY FULL	There is not enough directory space for the output files. You should either erase some unnecessary files or get another disk with more directory space and run LIB-86 again.
DISK FULL	There is not enough disk space for the output files. You should either erase some unnecessary files or get another disk with more room and run LIB-86 again.

Table H-1. (Continued)

Message	Meaning
DISK READ ERROR	LIB-86 cannot properly read a source or object file. This is usually the result of an unexpected end-of-file. Correct the problem in your source file.
INVALID COMMAND OPTION	LIB-86 encountered an unrecognized option in the command line. Retype the command line or edit the INP file.
MODULE NOT FOUND	The indicated module name, which appeared in a REPLACE, SELECT, or DELETE switch, cannot be found. Retype the command line or edit the INP file.
MULTIPLE DEFINITION	The indicated symbol is defined as PUBLIC in more than one module. Correct the problem in the source file, and try again.
NO FILE	LIB-86 cannot find the indicated file.
OBJECT FILE ERROR	LIB-86 detected an error in the object file. This is caused by a translator error or a bad disk file. Try regenerating the file.
RENAME ERROR	LIB-86 cannot rename a file. Check that the disk is not write-protected.

Table H-1. (Continued)

Message	Meaning
SYMBOL TABLE OVERFLOW	There is not enough memory for the Symbol Table. Reduce the number of options in the command line (MAP and XREF each use Symbol Table space), or use a system with more memory.
SYNTAX ERROR	LIB-86 detected a syntax error in the command line, probably due to an improper filename or an invalid command option. LIB-86 echoes the command line up to the point where it found the error. Retype the command line or edit the INP file.
VERSION 2 REQUIRED	LIB-86 requires a version 2 file system or later.

End of Appendix H

SID-86 Error Messages

Table I-1. SID-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".
BAD COMMAND OR PARAMETER; PRESS ? FOR HELP	The command, or parameters for a command were not entered correctly.
BAD FILE NAME	A filename in an E, R, or W command is incorrectly specified.
BAD HEX DIGIT	A SYM file being loaded with an E command has an invalid hexadecimal digit.
CANNOT CLOSE	The disk file written by a W command cannot be closed.
DISK READ ERROR	The disk file specified in an R command could not be read properly.

Table I-1. (Continued)

Error Message	Meaning
DISK WRITE ERROR	A disk write operation could not be successfully performed during a W command, probably due to a full disk.
EMPTY FILE	The file specified in an R command has length 0.
INSUFFICIENT MEMORY	There is not enough memory to load the file specified in an R or E command.
MACRO ALREADY EXISTS	An attempt was made to define a macro with a name already in use Verify defined macro names with the = command
MACRO NAME NOT FOUND	An attempt was made to use a macro that has not been defined. Verify defined macros with the = command.
MACRO OVERFLOW ERROR	The macro definition is too long.
MEMORY REQUEST DENIED	A request for memory during an R command could not be fulfilled either because the maximum number of memory locations has already been made, or the memory at the specified address is not available. Up to eight blocks of memory can be can be allocated at a given time under Concurrent.

Table I-1. (Continued)

Error Message	Meaning
NESTING MACROS NOT ALLOWED CONTINUE ENTERING COMMANDS	Macro definitions cannot include macros; definition ignored.
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.
PROGRAM TERMINATED NORMALLY (PC DOS ONLY)	The program running under SID-86 completed, or was terminated by a Ctrl-Break.
SYMBOL LENGTH ERROR	A symbol in a SYM file being loaded with an E command has more than thirty-one characters.
SYMBOL TABLE FULL	There is no more space in SID-86's symbol table.

Table I-1. (Continued)

Error Message	Meaning
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 non-existent memory at the indicated location.

End of Appendix I

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z