CPMPPUG0.WS4 (Edited by Emmanuel ROCHE.)

Programmer's Utilities Guide
for the
CP/M-86
Family of
Operating Systems

Digital Research
P.O. Box 579
160 Central Avenue
Pacific Grove, CA 93950
(408) 649-3896
TWX 910 360 5001

COPYRIGHT

DISCLAIMER

Digital Research makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, Digital Research reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Digital Research to notify any person of such revision or changes.

TRADEMARKS

CP/M-86 is a registered trademark of Digital Research. Concurrent CP/M-86, MP/M-86, CB86, RASM-86, XREF-86, LINK-86, LIB-86, SID-86, and PL/I-86 are trademarks of Digital Research. Intel is a registered trademark of Intel

Corporation. MCS-86 is a trademark of Intel Corporation.

The <Programmer's Utilities Guide for the CP/M-86 Family of Operating System> was prepared using the Digital Research TEX Text Formatter and printed in the United States of America.

Second Edition: February 1983

Foreword

This manual describes several utility programs that aid programmers and system designers in the software development process. Collectively, these utilities allow you to assemble 8086 assembly language modules, link them together to form a program that runs, and generate a cross-reference map of the variables used in a program. You can also use these utilities to create and manage your own libraries of subroutines and program modules, as well as create large programs by breaking them into separate overlays.

The <Programmer's Utilities Guide> assumes that you are familiar with the CP/M-86, MP/M-86, or Concurrent CP/M-86 operating system environment. It also assumes that you are familiar with the basic elements of 8086 assembly language programming.

RASM-86 is an assembler that translates 8086 assembly language statements into a relocatable object file in the Intel format. RASM-86 facilities include assembly of Intel 8086 mnemonics, assembly-time expressions, conditional assembly, page formatting of listing files, and powerful code-macro capabilities.

Section 1 describes the overall operation of RASM-86 and its optional run-time parameters. Section 2 describes elements of RASM-86 assembly language, including the character set, delimiters, constants, identifiers, operators, expressions, and statements.

Section 3 describes the various RASM-86 directives that control the assembly process. Section 4 contains a brief description of the RASM-86 instructions for data transfer, mathematical operations, string manipulation, control transfer, and processor control. Section 5 describes the code-macro facilities of RASM-86.

Section 6 describes XREF-86, an assembly language cross-reference program used with RASM-86. Section 7 describes LINK-86, the linkage editor that combines relocatable object modules into an absolute file that runs under CP/M-86, MP/M-86, or Concurrent CP/M-86. Section 8 describes how to use LINK-86 to produce overlays. Section 9 explains how to use LIB-86, the software librarian that creates and manages libraries.

The appendixes contain a complete list of error messages output by each of the utility programs.

Table of Contents
---

1 Introduction to RASM-86

1.1 Assembler Operation
1.2 Invoking RASM-86
1.3 Optional Run-time Parameters
1.4 Halting RASM-86

2 Elements of RASM-86 Assembly Language

2.1 RASM-86 Character Set
2.2 Tokens and Separators
2.3 Delimiters
2.4 Constants

2.4.1 Numeric Constants
2.4.2 Character Strings

2.5 Identifiers

2.5.1 Keywords
2.5.2 Symbols and Their Attributes

2.6 Operators

2.6.1 Operator Examples
2.6.2 Operator Precedence

2.7 Expressions
2.8 Statements

3 Assembler Directives

3.1 Segments
3.2 The Segment Directive

3.2.1 <segment name>
3.2.2 <align type>
3.2.3 <combine type>
3.2.4 <class name>

3.3 The GROUP Directive
3.4 The ORG Directive
3.5 The END Directive
3.6 The NAME Directive
3.7 The PUBLIC Directive
3.8 The EXTRN Directive
3.9 The IF, ELSE, and ENDIF Directives

3.10 The EQU Directive

3.11 The DB Directive

3.12 The DW Directive
3.13 The DD Directive
3.14 The RS Directive
3.15 The RB Directive
3.16 The RW Directive
3.17 The RD Directive

3.18 The EJECT Directive
3.19 The NOIFLIST and IFLIST Directives
3.20 The NOLIST and LIST Directives
3.21 The PAGESIZE Directive
3.22 The PAGEWIDTH Directive
3.23 The SIMFORM Directive
3.24 The TITLE Directive
3.25 The INCLUDE Directive

4 The RASM-86 instruction Set

4.1 Introduction
4.2 Data Transfer Instructions
4.3 Arithmetic, Logical, and Shift Instructions
4.4 String Instructions
4.5 Control Transfer Instructions
4.6 Processor Control Instructions

5 Code-macro Facilities

5.1 Introduction to Code-macros
5.2 Specifiers
5.3 Modifiers
5.4 Range Specifiers
5.5 Code-macro Directives

5.5.1 SEGFIX
5.5.2 NOSEGFIX
5.5.3 MODRM
5.5.4 RELB and RELW
5.5.5 DB, DW and DD
5.5.6 DBIT

6 XREF-86

6.1 Introduction
6.2 Invoking XREF-86

7 LINK-86

7.1 Introduction
7.2 Invoking LINK-86
7.3 Halting LINK-86
7.4 Definitions
7.5 The Link Process

7.5.1 Phase 1 - Collection

7.5.2 Phase 2 - Positioning

7.6 LINK-86 Command Options
7.7 CMD File Options

7.7.1 GROUP, CLASS, SEGMENT
7.7.2 ABSOLUTE, ADDITIONAL, MAXIMUM
7.7.3 ORIGIN
7.7.4 FILL/NOFILL

7.8 SYM File Options

7.8.1 LOCALS/NOLOCALS
7.8.2 LIBSYMS/NOLIBSYMS

7.9 MAP File Options
7.10 L86 File Options
7.11 Command Input File Options
7.12 I/0 Options

7.12.1 $Cd - Command
7.12.2 $Ld - Library
7.12.3 $Md - Map
7.12.4 $Od - Object
7.12.5 $Sd - Symbol

7.13 Command Line Errors

8 Overlays

8.1 Introduction
8.2 Writing Programs that Use Overlays

8.2.1 Overlay Method 1
8.2.2 Overlay Method 2
8.2.3 General Overlay Constraints

8.3 Command Line Syntax

9 LIB-86

9.1 LIB-86 Operation
9.2 Halting LIB-86
9.3 LIB-86 Command Options
9.4 Creating and Updating Libraries

9.4.1 Creating a New Library
9.4.2 Adding to a Library
9.4.3 Replacing a Module
9.4.4 Deleting a Module
9.4.5 Selecting a Module

9.5 Displaying Library Information

9.5.1 Cross-reference File
9.5.2 Library Module Map
9.5.3 Partial Library Maps

9.6 LIB-86 Commands on Disk

9.7 Redirecting I/O

Appendixes

A Mnemonic Differences from the Intel Assembler
B Reserved words
C RASM-86 Instruction Summary
D Code-macro Definition Syntax
E Sample Program
F RASM-86 Error messages
G LINK-86 Error Messages
H LIB-86 Error Messages
I XREF-86 Error Messages

Tables

1-1. RASM-86 Run-time Parameters
1-2. RASM-86 Command Line Examples

2-1. Separators and Delimiters
2-2. Radix Indicators for Constants
2-3. String Constant Examples
2-4. Register Keywords
2-5. RASM-86 Operators
2-6. Precedence of Operations in RASM-86

3-1. Default Segment Names
3-2. Default Align Types
3-3. Default Class name for Segments

4-1. Operand Type Symbols
4-2. Flag Register Symbols
4-3. Data Transfer Instructions
4-4. Effects of Arithmetic Instructions on Flags
4-5. Arithmetic Instructions
4-6. Logical and Shift Instructions
4-7. String Instructions
4-8. Prefix Instructions
4-9. Control Transfer Instructions
4-10. Processor Control Instructions

5-1. Code-macro Operand Specifiers
5-2. Code-macro Operand Modifiers

7-1. LINK-86 Usage of Class Names
7-2. LINK-86 Command Options
7-3. CMD File Option Parameters
7-4. Default Values for CMD File Options

9-1. LIB-86 Filetypes
9-2. LIB-86 Command Line Options

A-1. Mnemonic Differences
B-1. Reserved Words
C-1. RASM-86 Instruction Summary
F-1. RASM-86 Non-recoverable Errors
F-2. RASM-86 Diagnostic Error Messages
G-1. LINK-86 Error Messages
H-1. LIB-86 Error Messages
I-1. XREF-86 Error Messages

Figures

1-1. RASM-86 Source and object Files

6-1. XREF-86 Operation

7-1. LINK-86 Operation
7-2. Combining Segments with the Public Combine Type
7-3. Combining Segments with the Common Combine Type
7-4. Combining Segments with Stack Combination
7-5. Combining Segments using the Align Type
7-6. Paragraph Alignment
7-7. The Effect of Grouping Segments
7-7a. Segments without Group
7-7b. Segments within a Group

8-1. Using Overlays in a Large Program
8-1a. Without Overlays
8-1b. Separate Overlays
8-2. Tree Structure of Overlays

9-1. LIB-86 Operation

Listing

E-1. Sample Program APPE.A86

EOF

Section 1

Introduction to RASM-86

1.1 Assembler Operation

RASM-86 processes an 8086 assembly language source file in three passes and produces an 8086 machine language object file. RASM-86 can optionally produce three output files from one source file as shown in Figure 1-1.

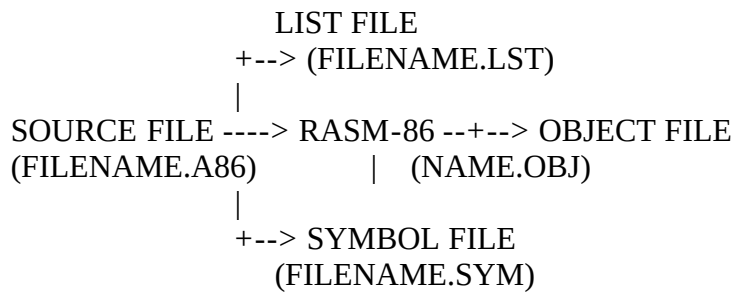


Figure 1-1. RASM-86 Source and Object Files

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

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.

1.2 Invoking RASM-86

Invoke RASM-86 with a command in the form:

RASM86 source file ($ optional parameters)

The filespec has the form:

[d:]filename[.typ]

where

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

filename is a valid CP/M-86 filename of 1 to 8 characters.

typ is a valid filetype of 1 to 3 characters, usually A86.

RASM-86 accepts a source file with any filetype. If you omit the filetype from the command line, RASM-86 searches the directory for the specified filename with the filetype A86.

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

```
A>rasm86 b:bios88

A>rasm86 bios88.a86 $ aa ob pb sb

A>rasm86 d:test
```

Once invoked, RASM-86 responds with the message:

```
-------------------------------------------------
RASM-86 Assembler  04-Feb-87  CP/M-86 Version 1.4
Serial No. XXXX-0000-654321    All Rights Reserved
Copyright (C) 1982-86       Digital Research, EDC.
-------------------------------------------------
```

where 1.4 is the RASM-86 version number. RASM-86 then attempts to open the source file. If the file does not exist on the designated drive or does not have the correct filetype, RASM-86 displays the message:

```
NO FILE
```

and stops processing.

By default, RASM-86 creates the output files on the currently logged-in 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%
```

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.

1.3 Optional Run-time Parameters

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

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

Parameter	Specifies	Valid Arguments

A	Source file device	A, B, C, ..., P
L	Local symbols in object file	O (upper-case "o", not zero)
O	object file device	A ..., P, Z
P	List file device	A ..., P, X, Y, Z
S	Symbol file device	A ..., P, X, Y, Z

All the 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 the message:

SYNTAX ERROR

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

A device name must follow the parameters A, O, P, and S. The devices are labeled as follows:

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

Device names A through P specify disk drives A through P, respectively. X specifies the user console, CON:, Y specifies the list device, LST:, and Z suppresses output, NUL:.

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

The LO 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.

Table 1-2. RASM-86 Command Line Examples

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

1.4 Halting RASM-86

You can halt the assembly at any time by pressing any key on the console keyboard. When you press a key, RASM-86 responds with the question:

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

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

EOF

Section 2

Elements of RASM-86 Assembly Language

2.1 RASM-86 Character Set

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

A B C D E F G H I J K L M N 0 P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9

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

space, tab, carriage return, and line-feed

RASM-86 treats lower-case letters as upper-case except within strings. Only alphanumerics, special characters, and spaces can appear in a string.

2.2 Tokens and Separators

A token is the smallest meaningful unit of 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.

2.3 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; these are explained in greater detail in Section 2.6.

Table 2-1. Separators and Delimiters

Character	Name	Use
20H	space	separator

09H	tab	legal in source files, expanded in list files
CR	carriage-return	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; also used in segment override specification
.	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	legal in identifiers
_	underscore	legal in identifiers
!	exclamation point	logically terminates a statement, allowing multiple statements on a single source line
'	apostrophe	delimits string constants

2.4 Constants

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

2.4.1 Numeric Constants

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

Table 2-2. Radix Indicators for Constants

Indicator	Constant Type	Base

B or b	binary	2
O or o	octal	8
Q or q	octal	8
D or d	decimal	10
H or h	hexadecimal	16

RASM-86 assumes that any numeric constant that does not terminate with a radix indicator is a decimal constant. Radix indicators can be upper- or lower-case.

A constant is thus a sequence of digits followed by an optional radix indicator where the digits are in the range for the radix. Binary constants must be composed of 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 that 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

2.4.2 Character Strings

A character string constant is a string of ASCII characters delimited by apostrophes. All RASM-86 instructions that allow 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 upper- and lower-case 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 3.8). The string cannot exceed 255 bytes. If you want to include an apostrophe in the string, you must enter it twice. RASM-86 interprets the two keystrokes " 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
```

2.5 Identifiers

The following rules apply to all identifiers:

1) Identifiers can be up to 80 characters long.

2) The first character must be alphabetic, one of the special characters ?, @, or the underscore character _.

3) Any subsequent characters can be either alphabetic, numeric, or the 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. The first type are keywords that have predefined meanings to RASM-86. The second type are symbols you define. The following are all valid identifiers:

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

2.5.1 Keywords

Keywords are reserved for use by RASM-86; you cannot define an identifier identical to a keyword. Appendix B contains the complete list of keywords.

RASM-86 recognizes five types of keywords:

1) instructions
2) directives
3) operators
4) registers
5) predefined numbers

Section 4 defines the 8086 instruction mnemonic keywords and the actions they initiate. Section 3 discusses RASM-86 directives. Section 2.6 defines operators. Table 2-4 lists the RASM-86 keywords that identify 8086 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	100B	Accumulator High Byte
BH	1	111B	Base Register High Byte
CH	1	101B	Count Register High Byte
DH	1	110B	Data Register High Byte
AL	1	000B	Accumulator Low Byte
BL	1	011B	Base Register Low Byte
CL	1	001B	Count Register Low Byte
DL	1	010B	Data Register Low Byte
AX	2	000B	Accumulator (full word)
BX	2	011B	Base Register (full word)
CX	2	001B	Count Register (full word)
DX	2	010B	Data Register (full word)
BP	2	101B	Base Pointer
SP	2	100B	Stack Pointer
SI	2	110B	Source Index
DI	2	111B	Destination Index
CS	2	01B	Code Segment Register
DS	2	11B	Data Segment Register
SS	2	10B	Stack Segment Register
ES	2	00B	Extra Segment Register

2.5.2 Symbols and Their Attributes

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

1) variables
2) labels
3) numbers

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

1) Segment tells which segment was being assembled when the variable was defined.

2) Offset tells how many bytes there are between the beginning of the segment and the location of this variable.

3) Type tells how many bytes of data are manipulated when this variable is referenced.

A segment can be a code segment, a data segment, a stack segment, or an extra segment, depending on its contents and the register that contains its starting address (see Section 3.2). The segment's starting address is a number between 0 and 65,535D. This number indicates the paragraph in memory to which the current segment is assigned, either when the program is assembled, when it is linked, or when it is 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 or 65,535D.

A variable has one of the following Type attributes:

```
BYTE
WORD
DWORD
```

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

```
my_variable db 0
```

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

```
another-variable EQU my_variable
```

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

Label segment and offset attributes are essentially the same as variable segment and offset attributes. 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
```

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:

```
MOV AL,5
```

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

2.6 Operators

RASM-86 operators fall into the following categories:

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

Table 2-5 defines the RASM-86 operators. In this table, a and b represent two elements of the expression. The validity column defines the type of operands the operator can manipulate. The vertical bar character "|" separates alternatives.

In this table, "number" refers to an absolute number which is a number whose value is known at assembly-time, such as a numeric constant. A relocatable number 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.

Table 2-5. RASM-86 Operators

Syntax	Result	Validity
Logical Operators		
a XOR b	bit-by-bit logical EXCLUSIVE OR of a and b	a, b = number
a OR b	bit-by-bit logical OR of a and b	a, b = number
a AND b	bit-by-bit logical AND of a and b	a, b = number
NOT a	logical inverse of a: all 0s become 1s, 1s become 0s	a = 16-bit number
Relational Operators		
a EQ b	returns 0FFFFH if a = b, otherwise 0	a, b = unsigned numbers; or labels, variables, or relocatable numbers defined in the same segment
a LT b	returns 0FFFFH if a < b, otherwise 0	a, b = unsigned numbers; or labels, variables, or relocatable numbers defined in the same segment

a LE b	returns 0FFFFH if a <= b, otherwise 0	a, b = unsigned numbers; or labels, variables, or relocatable numbers defined in the same segment
a GT b	returns 0FFFFH if a > b, otherwise 0	a, b = unsigned numbers; or labels, variables, or relocatable numbers defined in the same segment
a GE b	returns 0FFFFH if a >= b, otherwise 0	a, b = unsigned numbers; or labels, variables, or relocatable numbers defined in the same segment
a NE b	returns 0FFFFH if a <> b, otherwise 0	a, b = unsigned numbers; or labels, variables, or relocatable numbers defined in the same segment

Arithmetic Operators

a + b	arithmetic sum of a and b	a = number, variable, label, relocatable number, or external b = number
a - b	arithmetic difference of a and b	a = number, variable, label, relocatable number or external b = number; or variable, label, or relocatable number in the same segment as "a"
a * b	does unsigned multiplication of a and b	a, b = number
a / b	does unsigned division of a and b	a, b = number
a MOD b	returns remainder of a / b	a, b = number
a SHL b	returns the value that results from shifting a to left by an amount b	a, b = number
a SHR b	returns the value that results from shifting a to the right by an amount b	a, b = number
+ a	gives a	a = number
- a	gives 0-a	a = number

Segment Override

<seg reg>:	overrides assembler's choice	<seg reg> = CS, DS, SS, or ES

<addr exp> of segment register

Variable Manipulators, Creators

SEG a	creates a number whose value is the segment value of the variable or label a	a = label | variable
OFFSET a	creates a number whose value is the offset value of the variable or label a	a = label | variable
TYPE a	creates a number. If the variable a is of type BYTE, WORD or DWORD, the value of the number is 1, 2, or 4, respectively	a = label | variable
LENGTH a	creates a number whose value is the length attribute of the variable a. The length attribute is the number of bytes associated with the variable	a = variable
LAST a	if LENGTH a > 0, then LAST a = LENGTH a - 1; if LENGTH a = 0, then LAST a = 0	a = variable
a PTR b	creates virtual variable or label with type of a and attributes of b	a = BYTE | WORD | DWORD b = <addr exp>
	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	no argument

2.6.1 Operator Examples

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

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

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

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

Addition and subtraction operators compute the arithmetic sum and difference of two operands. The first operand can be a variable, label, or number. For addition, the second operand 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. For example,

```
0002          count   equ     2
0005          disp1   equ     5
000A FF         flag    db      offh
000B 2EA00B00           mov     al,flag+1
000F 2E8A0E0F00         mov     cl,flag+disp1
0014 B303               mov     bl,disp1-count
```

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

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

Unary operators accept both signed and unsigned operators, as shown below.

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

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

```
segment register:address expression
```

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

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

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. For 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
```

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 as shown below.

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

The period operator "." creates a variable in the current Data segment. The new variable has a segment attribute equal to the current Data segment and an offset attribute equal to its operand.

Its operand must be a number. For example,

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

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

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

2.6.2 Operator Precedence

Expressions combine variables, labels, or numbers with operators. RASM-86 allows several kinds of expressions (see Section 2.7). This section defines the order in which RASM-86 performs operations if more than one operator appears 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 RASK-86

Order	Operator Type	Operators
1	Logical	XOR, OR
2	Logical	AND
3	Logical	NOT
4	Relational	EQ, LT, LE, GT, GE, NE
5	Addition/subtraction	+, -
6	multiplication/division	*, /, MOD, SHL, SHR
7	Unary	+, -
8	Segment override	<segment override>:
9	Variable manipulators, creators	SEG, OFFSET, PTR, TYPE, LENGTH, LAST
10	Parentheses/brackets	(), []
11	Period and Dollar	. $

2.7 Expressions

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

1) a segment value
2) an offset value
3) a 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]
```

2.8 Statements

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

You must terminate each assembly language statement with a carriage-return (CR) and line-feed (LF), or with an 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.

Section 3 describes the RASM-86 instruction set in detail. The syntax for an instruction statement is

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

where the fields are defined as:

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 may 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. The syntax for a directive statement is

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

where the fields are defined as:

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.

EOF

Section 3

Assembler Directives

Assembler 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.

Section 2.8 shows the general syntax for directive statements. The following sections give the specific syntax and explanation for each directive statement. In the general syntax line for each statement, square brackets [] enclose optional arguments, and user-supplied arguments are shown in angle brackets <>.

Assembler directives are grouped into the following categories:

segment control:
    CSEG DSEG ESEG SSEG GROUP

linkage control:
    NAME PUBLIC EXTRN END

conditional assembly:
    IF ELSE ENDIF

symbol definition:
    EQU

data definition and memory allocation:
    DB DW DD RS RB RW RD

output listing control:
    EJECT IFLIST/NOIFLIST LIST/NOLIST PAGESIZE
    PAGEWIDTH SIMFORM TITLE

miscellaneous:
    INCLUDE ORG

3.1 Segments

The 8086 CPU can address one megabyte (1,048,576 bytes) of memory. This entire address space can be subdivided into an arbitrary number of smaller units called segments. These can be up to 64K bytes in length. Each segment is comprised of contiguous memory locations that make up a logically independent and separately addressable unit. Each segment must have a base address that specifies its starting location in the memory space. Each segment base address must begin on a boundary divisible by 16, but there are no other restrictions on segment boundaries.

Every location in the memory space has a physical address and a logical address. A physical address is a 20-bit value that specifies a unique byte location within the memory space. A logical address is the combination of a 16-bit segment base value and a 16-bit 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 20-bit physical address. Note that 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 that usually contains 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 also typically contains data.

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

The size and type of segments you use in a program determine the type of memory execution model used by the operating system. 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. With RASM-86, you can also create an arbitrary number of Code, Data, Stack, and Extra segments to more fully use the address space of the 8086 processor. You can therefore have more than 64K of code or data by using several segments and managing the segments with the assembler directives.

3.2 The Segment Directive

Every instruction and variable in a program must be contained in a segment. Instruction statements must be assigned to the Code Segment, but directive statements can be assigned to any segment. Create a segment and name it by using the following Segment directive.

```
[<segment name>]<segment>[<align type>][<combine type>][<'class name'>]
```

where is one of the following:

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

For example,

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

3.2.1 <segment 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. Table 3-1 shows the default names.

Table 3-1. Default Segment Names

Segment Directive	Default 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.

3.2.2 <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 to combine segments when it produces an executable file. You can specify one of four different 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. Table 3-2 shows the default values.

Table 3-2. Default Align Types

Segment Directives	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.

3.2.3 <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 types:

1) PUBLIC
2) COMMON
3) STACK
4) LOCAL
5) nnnn (absolute segment)

If you specify a <combine type>, it must be with the first definition of the segment. You can omit the <combine type> on subsequent Segment directives that name the same segment, but you cannot change the original type. If you do not specify a <combine type>, RASM-86 supplies the default type, PUBLIC.

PUBLIC means that the linkage editor can combine the segment with other segments that have 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 Section 3.3).

The STACK <combine type> is similar to PUBLIC, in that the storage allocated for STACK segments is the sum of the STACK segments from each module. But, 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.

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

For an absolute segment, RASM-86 determines 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.

3.2.3 <class name>

The <class name> can be any valid RASM-86 identifier. The <class name> is a means of identifying segments that are to be placed in the same section of the CMD file created by LINK-86. Unless overriden 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

3.3 The GROUP Directive

```
<group name> GROUP <segment name 1>,<segment name 2> ....
```

The GROUP directive instructs RASM-86 to combine the named segments into a collection called a group whose length can be up to 64K. Offsets within any of the segments of a group are relative to the beginning of the group rather than the beginning of the segment.

The order of the <segment names> in the directive is the order in which the linkage editor arranges the segments in the CMD file.

3.4 The ORG Directive

```
ORG <numeric expression>
```

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

The <numeric expression> is relative to 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 <numeric expression> indicates the actual offset within the segment.

However, 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 part of the segment from the program being assembled.

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.

3.5 The 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. The <start label> also 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.

3.6 The 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. 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 object module names to identify object modules.

3.7 The PUBLIC Directive

PUBLIC <name>[,<name>,...]

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

3.8 The 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 symbol can be a variable, label, or number.

Type is one of the following:

Variables:

BYTE
WORD
DWORD

Labels:

NEAR
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 segment in which the symbol is defined in some other module.

3.9 The 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 the alternative source lines.

3.10 The EQU Directive

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

The EQU (equate) directive assigns values and attributes to user-defined symbols. 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 or address expressions before using the EQU directive.

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

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

3.11 The DB Directive

```
[symbol] DB <numeric expression>[,<numeric expression>,...]
[symbol] DB <string constant>[,<string constant>,...]
```

The DB directive defines initialized storage areas in byte format. 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 2.4.2. Note that RASM-86 does not perform translation from lower- to upper-case 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 as long as 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: 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 statements show DB directives with symbols:

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

3.12 The 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 statements:

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

3.13 The DD Directive

[symbol] DD <address expression>[,<address expression>,...]

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

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

3.14 The RS Directive

[symbol] RS <numeric expression>

The RS directive allocates storage in memory but does not initialize it. The numeric expression gives the number of bytes to reserve. Note that the RS directive does not give a type Byte attribute to the optional symbol. For example,

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

3.15 The 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 type Byte attribute to the optional symbol.

3.16 The RW Directive

[symbol] RW <numeric expression>

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

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

3.17 The 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
```

3.18 The 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.

3.19 The NOIFLIST and 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.

3.20 The NOLIST and LIST Directives

NOLIST
LIST

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

3.21 The PAGESIZE Directive

PAGESIZE <numeric expression>

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

3.22 The 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.

3.23 The 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 that is unable to interpret the form-feed character.

3.24 The 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'
```

3.25 The INCLUDE Directive

```
INCLUDE <filename>
```

The INCLUDE directive includes another RASM-86 source file in the source text. For example,

```
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; that is, 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 assumes the drive containing the source file.

EOF

Section 4

The RASM-86 Instruction Set

4.1 Introduction

The RASM-86 instruction set includes all 8086 machine instructions. Section 2.8 gives the general syntax for instruction statements. The following sections define the specific syntax and required operand types for each instruction, without reference to labels or comments. The instruction definitions are presented in tables for easy reference.

For a more detailed description of each instruction, see the Intel MCS-86 Assembly Language Reference Manual. For descriptions of the instruction bit patterns and operations, see the Intel MCS-86 User's Manual.

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.

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

Table 4-1. Operand Type Symbols

Symbol	Operand Type
numb	any numeric expression
numb8	any numeric expression that evaluates to an 8-bit number
acc	accumulator register, AX or AL
reg	any general purpose register that is not a segment register
reg16	a 16-bit general purpose register that is not a segment register
segreg	any segment register: CS, DS, SS, or ES
mem	any address expression with or without base- and/or index-addressing modes, such as: variable variable+3 variable[bx]

variable[SI]
variable[BX+SI]
[BX]
[BP+DI]

simpmem any address expression without base- and index-addressing modes, such as:
variable
variable+4

mem|reg any expression symbolized by reg or mem

mem|regl6 any expression symbolized by mem|reg, but must be 16 bits

label any address expression that evaluates to a label

lab8 any label that is within +/- 128 bytes distance from the instruction

The 8086 CPU has nine single-bit Flag registers that reflect the state of the 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 4-2 to represent the nine Flag registers.

Table 4-2. Flag Register Symbols

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

4.2 Data Transfer Instructions

There are four classes of data transfer operations:

1) general purpose
2) accumulator specific
3) address-object
4) flag

Only SAHF and POPF affect flag settings. Note in Table 4-3 that if acc = AL, a byte is transferred, but if acc = AX, a word is transferred.

Table 4-3. Data Transfer Instructions

Syntax	Result
IN acc,numb8	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
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
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 Arithmetic, Logical, and Shift Instructions

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

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

Table 4-4. Effects of Arithmetic Instructions on Flags

Flag Bit	Result
CF	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 4-5. Arithmetic Instructions

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

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 4-6. Logical and shift instructions

Syntax

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
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
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

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

4.4 String Instructions

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

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

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

Table 4-7. String instructions

Syntax	Result
CMPS mem|reg,mem|reg	subtract source from destination, affect flags, but do not return result
CMPSB	an alternate mnemonic for CMPS 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
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 mem1reg 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 4-8 defines prefixes for string instructions. A prefix repeats its string instruction the number of times contained in the CX register, which is decremented by 1 for each iteration. Prefix mnemonics precede the string instruction mnemonic in the statement line.

Table 4-8. Prefix Instructions

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

4.5 Control Transfer Instructions

There are four classes of control transfer instructions:

1) calls, jumps, and returns
2) conditional jumps
3) iteration control
4) interrupts

All control transfer instructions cause program execution to continue at some new location in memory, possibly in a new code segment. The transfer may be absolute, or it can depend upon a certain condition. Table 4-9 defines control transfer instructions. In the definitions of conditional jumps, "above" and "below" refer to the relationship between unsigned values. "Greater than" and "less than" refer to the relationship between signed values.

Table 4-9. Control Transfer Instructions

Syntax	Result
CALL label	push the offset address of the next instruction on the stack, jump to the target label
CALL mem|regl6	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 interupt-vector elements - uses three levels of stack
INTO	if OF (the overflow flag) is set, push the flag registers (as in PUSHF), clear TF and IF flags, 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 lab8	jump if "below or equal" or "not above" ((CF or ZF)=1)
JC lab8	same as JB
JCXZ lab8	jump to target label if CX register is zero
JE lab8	jump if "equal" or "zero"(ZF=1)
JG lab8	jump if "not less or equal" or "greater" (SF xor OF) or ((ZF)=0)
JGE lab8	jump if "not less" or "greater or equal" ((SF xor OF)=0)
JL lab8	jump if "less" or "not greater or equal" ((SF xor OF)=1)
JLE lab8	jump if "less or equal" or "not greater" (SF xor OF) or ZF)=1)
JMP label	jump to the target label

JMP mem|regl6 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=O)
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 labB jump if "overflow" (OF=1)
JP lab8 jump if "parity" or "parity even" (PF=1)
JPE lab8 same as JP
JPO lab8 same as JNP
JS lab8 jump if "sign" (SF=1)
JZ lab8 same as JE
LOOP lab8 decrement CX register by one, jump to target label if CX is not zero
LOOPE lab8 decrement CX register by one, jump to target label if CX is not zero and the ZF flag is set - "loop while zero" or "loop while equal"
LOOPNE lab8 decrement CX register by one, jump to target label if CX is not zero and ZF flag is cleared - "loop while not zero" or "loop while not equal"
LOOPNZ lab8 same as LOOPNE
LOOPZ lab8 same as LOOPE
RET return to the 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

4.6 Processor Control Instructions

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

Table 4-10. Processor Control Instructions

Syntax	Results
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
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.

EOF

Section 5

Code-macro Facilities

5.1 Introduction to Code-macros

RASM-86 does not support traditional assembly language macros, but it does allow you to define your own instructions using the Code-macro directive. RASM-86 assembles code-macros wherever they appear in assembly language code, but there the similarity to traditional macros ends.

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; 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.

RASM-86 treats a code-macro as an instruction, so that 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.

```
XCHG    BX,WORD3
MYCODE  PARM1,PARM2
MUL     AX,WORD4
```

Note that MYCODE accepts two operands that are its formal parameters. When you define MYCODE, RASM-86 classifies these two operands as to type, size, etc. The names of formal parameters are not fixed, so RASM-86 replaces them with the names or values supplied as operands when you invoke the code-macro. The formal parameters are placeholders that indicate where and how the operands are to be used.

A code-macro definition takes the general form:

```
Code-Macro <name> [<formal parameter list>]
code-macro body
EndM
```

where the optional <formal parameter list> is defined:

```
<formal name>:<specifier letter>[<modifier letter>][<range>]
```

If you specify a formal parameter list, the specifier letter is required but the modifier letter is optional. Possible specifiers are A, C, D, E, M, R, S, and X. Possible modifier letters are b, d, w, and sb. RASM-86 ignores case except within strings, but (for clarity) this section shows specifiers in upper-case, and modifiers in lower-case. Following subsections describe

specifiers, modifiers, and the optional range in greater detail.

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

```
SEGFIX
NOSEGFIX
MODRM
RELB
RELW
DB
DW
DD
DBIT
```

These directives are unique to code-macros. The code-macro directives DB, DW, and DD that appear to duplicate RASM-86 directives DB, DW, and DD have different meanings in code-macro context. These directives are discussed in detail in Section 5.5.5.

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. The following examples are typical code-macro definitions.

```
CodeMacro AAA
      DB    37H
EndM

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

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

5.2 Specifiers

Every formal parameter must have a specifier letter that indicates 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 to be used as an immediate value.
E Effective address, either an M (memory address) or an R (register).
M Memory address. This can be either a variable or a bracketed register expression.
R A general register only.
S Segment register only.
X A direct memory reference.

5.3 Modifiers

The optional modifier letter is a further requirement on the operand. The meaning of the modifier letter depends on the type of the operand. For variables, the modifier requires the operand to be of type b for byte, w for word, d for double-word, and sb for signed byte. For numbers, the modifiers require the number to be of a certain size: b for -256 to 255 and w for other numbers. Table 5-2 summarizes code-macro modifiers.

Table 5-2. Code-macro Operand Modifiers

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

5.4 Range Specifiers

The optional range is specified 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. The following example specifies that the input port must be identified by the DX register:

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

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

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

The last example specifies that the opcode is to be immediate data, and can range from 0 to 63 inclusive:

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

5.5 Code-macro Directives

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

5.5.1 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 form:

```
SEGFIX <formal name>
```

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

5.5.2 NOSEGFIX

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

```
NOSEGFIX segreg,<formname>
```

where segreg is one of the segment registers ES, CS, SS, or DS, and <formname> 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 use:

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

5.5.3 MODRM

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

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

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

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

For further information about the 8086's instructions and their bit patterns, see the Intel 8086 Assembly Language Programming Manual and the Intel 8086 Family User's Manual. MODRM has the forms:

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

where NUMBER7 is a value 0 to 7 inclusive, and <form name> is the name of a formal parameter. The following examples show how to use MODRM:

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

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

5.5.4 RELB and RELW

These directives, used in IP-relative branch instructions, instruct RASM-86 to generate a displacement between the end of the instruction and the label that is supplied as an operand. RELB generates one byte and RELW two bytes of displacement. The directives have the following forms:

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

where <form name> is the name of a formal parameter with a C (code) specifier.

For example,

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

5.5.5 DB, DW, and DD

These directives differ from those that occur outside code-macros. The directives have the following forms:

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

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

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

5.5.6 DBIT

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

```
DBIT <field description>[,<field description>]
```

where a <field description> has two forms:

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

where <number> ranges from 1 to 16, and specifies the number of bits to be set. <combination> specifies the desired bit combination. The total of all the <numbers> listed in the field descriptions must not exceed 16.

The second form shown above contains <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:

```
0 = 0000      AL     ES     AX  (= AL + AH)
1 = 0001      CL     CS     CX  (= CL + CH)
2 = 0010      DL     SS     DX  (= DL + DH)
3 = 0011      BL     DS     BX  (= BL + BH)
```

```
4 = 0100      AH     SP
5 = 0101      CH     BP
6 = 0110      DH     SI
7 = 0111      BH     DI
```

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

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

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

```
DEC DX
```

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

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

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

EOF

Section 6

XREF-86

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 the 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. Figure 6-1 illustrates XREF-86 operation.

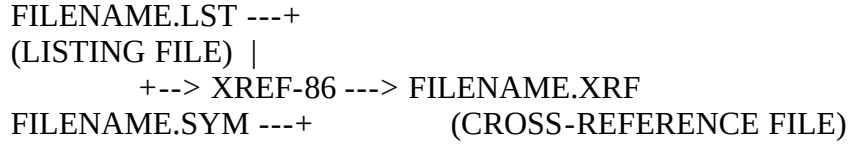


Figure 6-1. XREF-86 Operation

6.2 invoking XREF-86

Invoke XREF-86 with the command form:

```
XREF86 <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 that exist in the file <filename>.SYM.

After completing this copy operation, XREF-86 appends to <filename>.XRF a cross-reference report that lists 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>, it searches for the input files and creates the output file on the specified drive. Otherwise, XREF-86 associates the files with the default drive.

XREF-86 also allows you to direct the output file to the default list device instead of to <filename>.XRF. To redirect the output to the printer, add the

string "$p" to the command line. For example,

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

EOF

Section 7

LINK-86

7.1 Introduction

LINK-86 is the Digital Research linkage editor that combines relocatable object files into a command file that runs under any of the Digital Research family of 8086-based operating systems. The object files can be produced by Digital Research's 8086 language translators such as RASM-86, PL/I-86 and CB86, or by other translators that produce object files using a compatible subset of the Intel 8086 object module format.

LINK-86 accepts two types of object files. The first type is an object file containing a single object module. This type generally has the filetype OBJ, and is produced by a language translator. The second type is a library file which is an indexed library of object modules. A library file has a filetype L86, and is generated by the library manager, LIB-86, in the Intel 8086 object module format. LINK-86 can search such a library file and select only those modules needed by the other programs being linked.

LINK-86 produces three files:

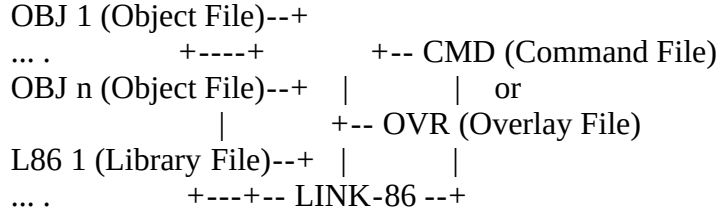
1) A Command (CMD) File
2) A Symbol Table (SYM) File
3) A Map (MAP) File

The CMD file contains a memory image of the program that runs directly under CP/M-86, MP/M-86, and Concurrent CP/M-86. The SYM file contains a list of symbols from the object files, and their offsets, and is suitable for use with SID-86, the Digital Research Symbolic Instruction Debugger. The MAP file contains information about the layout of the CMD file.

LINK-86 displays any unresolved symbols at the console. Unresolved symbols are those that have been referenced but not defined in the files being linked. These symbols must be resolved before the program will run properly, unless you are linking overlays (see Section 8).

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

Figure 7-1 illustrates LINK-86 operation.



```
L86 n (Library File)--+   |          +-- SYM (Symbol Table File)
                      |   |
INP (Input Command File)--+          +-- MAP (Module Map File)
```

Figure 7-1. LINK-86 Operation

7.2 Invoking LINK-86

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

```
LINK86 {file} = file1{,file2,...,filen}
```

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, the command

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

creates MYFILE.CMD and MYFILE.SYM.

If you omit the new 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.

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 the disk (see Section 7.11).

7.3 Halting LINK-86

You can halt LINK-86 during processing by pressing any console key. LINK-86 displays the message:

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

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

7.4 Definitions

This section uses the following terms to describe how LINK-86 processes object files and creates the CMD file.

Term	Description

Segment A Segment is a collection of code or data bytes whose length is less than 64K. A segment is the smallest unit that LINK-86 manipulates when creating the CMD file.

Segment name* A Segment name can be any valid RASM-86 identifier. LINK-86 combines all segments with the same segment name from separate object files.

Class name* A Class name can be 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>* The <Align type> indicates on what type of boundary the segment is to begin. The Align types are byte, word, paragraph, and page. LINK-86 uses the align type in two ways: first, when it combines parts of segments from separate files, and second, when it combines segments into groups or sections of the CMD file.

<Combine type>* The <Combine type> determines how LINK-86 can combine parts of segments with the same name from different files. The Combine types are: public, common, stack, absolute, and local.

Section A section is one of up to eight parts of an CMD file, any one of which can be up to one megabyte in length. Note: In the documentation for Digital Research's 8086-based operating systems, each part of a CMD file is described by a Group Descriptor. The term "section" is used here, instead of Group Descriptor, to avoid confusion with the term "group", defined below.

Group* A Group is a collection of segments whose total length is less than 64K, and thus is addressable from a single segment register. Groups allow you to break up a program into segments, while still allowing the segments to be addressed without changing the contents of a segment register. This technique results in shorter and faster code than addressing segments with 32-bit pointers.

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

7.5 The Link Process

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

7.5.1 Phase 1 - Collection

In Phase 1, LINK-86 first collects segments having the same Segment name and Class name from the separate files being linked, and then combines the segments according to the align and combine attributes.

For example, suppose there are three object files, FILEA.OBJ, FILEB.OBJ, and FILEC.OBJ, and each file defines a segment named Dataseg with the statement:

```
dataseg dseg
```

Figure 7-2 illustrates how LINK-86 combines this segment using the Public Combine type.

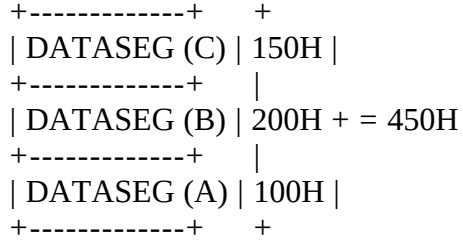


Figure 7-2. Combining Segments with the Public Combine Type

LINK-86 combines these segments by concatenating the parts of the segments found in the separate object files with the appropriate space between the parts 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, produces it as a default.

Figure 7-3 illustrates the Common Combine type. Suppose the three files FILEA.OBJ, FILEB.OBJ, and FILEC.OBJ each contain a segment named Dataseg defined with the statement:

```
dataseg dseg common
```

LINK-86 combines these segments so that all parts of the segments from the separate files being linked have the same low address in memory. Note that this corresponds to a common block in high-level languages.

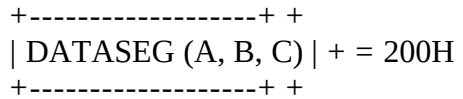


Figure 7-3. Combining Segments with the Common Combine Type

Figure 7-4 illustrates the Stack Combine type. LINK-86 combines these segments so that the total length of the segment is the sum of the parts from the separate files being linked, including any intersegment gaps due to the Align type. However, all the parts share the same high address since stacks grow downward from high memory.

For example, suppose the three files FILEA.OBJ, FILEB.OBJ, and FILEC.OBJ each contain a segment named Stkseg. Figure 7-4 illustrates how they are combined

by LINK-86.

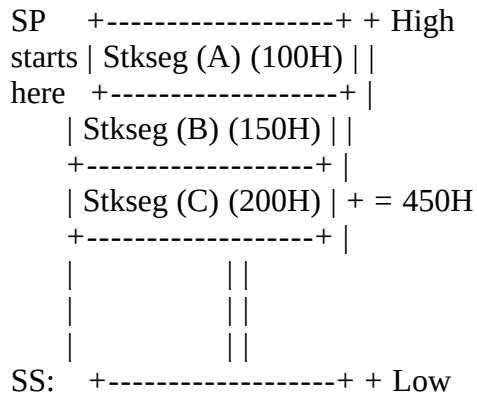


Figure 7-4. Combining Segments with Stack Combination

The Align type indicates on what type of boundary the segment begins, and thus determines the amount of space LINK-86 leaves between parts of segments of the same name. For example, suppose the three files FILEA.OBJ, FILEB.OBJ, and FILEC.OBJ each contain a segment named Dataseg. Figure 7-4 illustrates how LINK-86 uses the Align type to combine these segments.

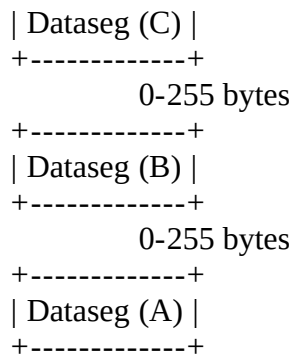


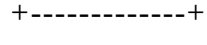
Figure 7-5. Combining Segments using the Align Type

In Figure 7-5, the gap between the segments is determined by the Align type, and can be up to 255 bytes in length. For example, there is no gap if the Align type is byte. This produces the most compact code.

If the Align type is word, 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 processor performs faster memory accesses for word-aligned data.

The gap required for paragraph-aligned segments can be up to 15 bytes, while page-aligned segments can require up to 255 bytes.

Figure 7-6 illustrates a specific example. Suppose the segment Dataseg has the paragraph Align type. Suppose also that Dataseg 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.



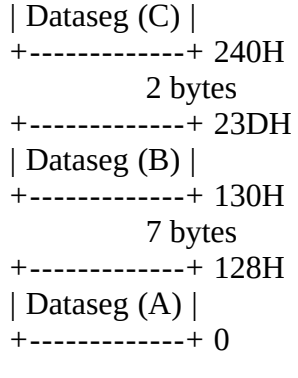


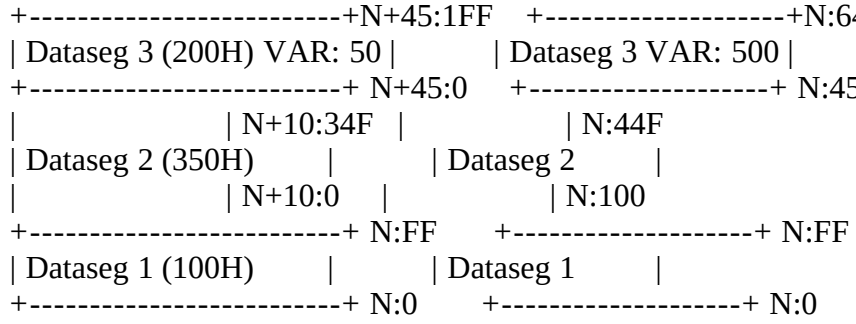
Figure 7-6. Paragraph Alignment

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

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

7.5.2 Phase 2 - Positioning

In Phase 2, LINK-86 combines segments that are members of groups, again using the Align type to determine intersegment gaps. Figure 7-7 illustrates how LINK-86 combines segments into groups.



7-7a. Segments Without Groups 7-7b. Segments Within A Group

Figure 7-7. The Effect of Grouping Segments

In Figure 7-7, N:0 is the base address where the segments are loaded at run-time (paragraph N, offset 0). Figure 7-7a shows that each segment not contained in a group begins at offset zero, and thus can be up to 64K in length. 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 the variable, VAR, at run-time, the program must load a segment register with the base of the segment Dataseg3 because LINK-86 assigns VAR an offset of 50H.

In Figure 7-7b, the same Segments are combined in a group. The offsets of the segments are now cumulative, and thus cannot extend past 0FFFFH. The offset of VAR is 500H relative to the base of the group. At run-time, the program does not need to change a segment register to point to Dataseg3, but can access VAR

directly using the segment register that points to the base of the group.

After combining segments into groups, LINK-86 assigns each segment to a section of the CMD file as follows:

1) Segments belonging to the group CGROUP are placed in the CODE section of the CMD file.

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

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

4) Segments that have not been processed by any of the above means are omitted from the CMD file, because LINK-86 does not have sufficient information to position them.

You can override the way LINK-86 positions segments by using command line options, as described in Section 7.7.

Table 7-1. LINK-86 Usage of Class Names

Class Name	CMD File Section	Segment Directive (RASM-86)
CODE	CODE	CSEG
DATA	DATA	DSEG
EXTRA	EXTRA	ESEG
STACK	STACK	SSEG
X1 *	X1	
X2 *	X2	
X3 *	X3	
X4 *	X4	

* There is no segment directive that produces this class name as a default; you must supply it explicitly.

7.6 LINK-86 Command options

When you invoke LINK-86, you can specify command line options that control the link operation. Each command option falls into one of several categories, depending on the type of file it affects.

The first category of command options affects the contents of the CMD file, and therefore applies to the entire link operation. The second category of options affects the SYM and MAP files. These options act as toggles that turn on and off as LINK-86 processes the command line from left to right. The third category of options affects the Library and Input files, and therefore applies

only to one file in the command line.

Table 7-2 shows the LINK-86 command options including the abbreviation for each.

Table 7-2. LINK-86 Comand Options

Option	Abbr.	Meaning
CODE	C	controls contents of CODE section of CMD file
DATA	D	controls contents of DATA section of CMD file
EXTRA	E	controls contents of EXTRA section of CMD file
STACK	ST	controls contents of STACK section of CMD file
X1	X1	controls contents of X1 section of CMD file
X2	X2	controls contents of X2 section of CMD file
X3	X3	controls contents of X3 section of CMD file
X4	X4	controls contents of X4 section of CMD file
FILL	F	zero fill and include uninitialized data in CMD file
NOFILL	NOF	do not include uninitialized data in CMD file
INPUT	I	read command line from disk file
MAP	M	create a MAP file
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
SEARCH	S	search library and only link modules referenced

You enclose command options in square brackets immediately following a filename. For example,

```
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]
```

The following subsections describe the function and syntax for each of the command options.

The following subsections describe the function and syntax for each of the command options.

7.7 CMD File Options

The following command options affect the contents of the CMD file that LINK-86 creates:

```
CODE        DATA        STACK       EXTRA
X1          X2          X3          X4
FILL        NOFILL
```

Note that these options can appear after any filename in the command line.

The first eight options control the way LINK-86 places segments in the CMD file, and the contents of the CMD file header. The FILL and NOFILL options tell LINK-86 what to do with uninitialized data that can occur at the end of a section of the CMD file.

A CMD file consists of a 128-byte header record followed by up to eight sections, any one of which can be up to 1 megabyte in length. These sections, called CODE, DATA, STACK, EXTRA, X1, X2, X3, and X4, correspond to the LINK-86 command options. The header contains information such as the length of each section of the CMD file, its minimum and maximum memory requirements, and its load address. This information is used by the operating system to properly load the file (see the <CP/M-86 Operating System System Guide>).

When you link object modules created by a Digital Research compiler such as CB86, or PL/I-86, the linkage editor generates some prefix code at the beginning of the CMD file. If you are linking RASM-86 modules only, you can use the NOPREFIX option and cause LINK-86 to suppress generation of this prefix code.

Each of the options that affect the CMD file sections must be followed by one or more parameters enclosed in square brackets. Table 7-3 shows the parameters, their abbreviations, and meanings.

Table 7-3. CMD File Option Parameters

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

7.7.1 GROUP, CLASS, SEGMENT

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

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

instructs LINK-86 to place the segments CODE1, CODE2, and all the segments in group XYZ into the CODE section of the file TEXT.CMD.

7.7.2 ABSOLUTE, ADDITIONAL, MAXIMUM

The ADDITIONAL and MAXIMUM parameters tell LINK-86 the values to place in the CMD file header. These parameters override the default values that LINK-86 usually uses. Each parameter is a hexadecimal number enclosed in square brackets. Table 7-4 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 CMD file at run-time. The ADDITIONAL parameter indicates the amount of additional memory, in paragraphs, required by the indicated section of the CMD file. The program could use this memory for Symbol Table space or I/O buffers that are needed at run-time, but are not included in the source program, and thus are not in the OBJ file. The MAXIMUM parameter indicates the maximum amount of memory needed by the indicated section of the CMD 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:

1) The DATA section requires at least 100H paragraphs in addition to the data in the CMD file.

2) The DATA section can use up to 1000H paragraphs of memory.

3) The CODE section must load at absolute paragraph address 40H.

7.7.3 Origin

The ORIGIN parameter is a hexadecimal value that indicates the byte offset where the indicated section of the CMD 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 (see the <CP/M-86 Operating System System Guide>).

Table 7-4 summarizes the default values for each of the command options and parameters.

Table 7-4. Default Values for CMD File Options and Parameters

OPTION	GROUP	CLASS	SEGMENT	ABSOLUTE	ADDITIONAL	MAXIMUM	ORIGIN
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 0.

7.7.4 FILL/NOFILL

The FILL and NOFILL options tell LINK-86 what to do with uninitialized data that can occur at the end of a section of the CMD file. The FILL option directs LINK-86 to include this uninitialized data in the CMD file, and fill it with zeros. The NOFILL option directs LINK-86 to omit the uninitialized data from the CMD file. The FILL option usually results in a larger CMD file, but the LINK-86 operation is usually faster when FILL is enabled. The default is FILL. Note that these options apply only to uninitialized data at the end of a section of the CMD file. Uninitialized data which is not at the end of a section is always zero filled and included in the CMD file.

7.8 SYM File Options

The following command options affect the contents of the SYM file that LINK-86 creates:

- LOCALS
- LIBSYMS
- NOLOCALS
- NOLIBSYMS

These options must appear in the command line after the specific file or files to which they apply. These options remain in effect until you change them, as LINK-86 processes the command line from left to right.

7.8.1 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 option 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.

7.8.2 LIBSYMS/NOLIBSYNS

The LIBSYMS option directs LINK-86 to include in the SYM file any symbols that come from a library that is 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 Run-time Subroutine Library of a high-level language such as PL/I-86. Because the symbols in such a library are usually of no interest to the programmer, the default is NOLIBSYMS.

7.9 MAP File Options

The MAP option directs LINK-86 to create a MAP file that contains information about the segments in the CMD file. The amount of information that LINK-86 puts into the MAP file is controlled by the optional parameters:

```
OBJMAP NOOBJMAP
L86MAP NOL86MAP
ALL
```

that 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.

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,

```
A>LINK86 FINANCE [MAP[ALL]],SCREEN,GRAPH.L86[S,MAP[NOL86MAP]]
```

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

7.10 L86 File Options

The SEARCH option directs LINK-86 to search the preceding file and include in the CMD file only those modules which satisfy 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 CMD file. For example, the command

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

creates the file TEST1.CMD by combining the object files TEST1.OBJ, TEST2.OBJ, and any modules from MATH.L86 that are referenced in 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.

7.11 Input File Options

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. LINK-86 stops scanning the command line, entered from the console, when it encounters this option. 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. For example, the file TEST.INP might include the lines:

```
MEMTEST=TEST1,TEST2,TEST3,
IOLIB.L86[S],MATH.L86[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.

7.12 I/O Options

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

```
$td
```

where t is a type and d is a drive specifier.

LINK-86 recognizes five types:

```
C - Command File (CMD or OVR)
L - Library File (L86)
M - Map File (MAP)
O - Object File (OBJ or L86)
```

S - Symbol File (SYM)

The drive specifier can be a letter in the range A thru P corresponding to one of sixteen logical drives, or one of the following special characters:

X - Console
Y - Printer
Z - Byte bucket

When you use the $ option, you cannot separate the td character pair with commas. However, 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 it is changed as LINK-86 processes the command line from left to right. This is useful when linking overlays (see Section 8). For example, the command

```
A>link86 root (ov1[$sz])(ov2)(ov3)(ov4[$sa])
```

suppresses the SYM file generated when OV1, OV2 and OV3 are linked. When LINK-86 links OV4, it places the SYM file on drive A.

7.12.1 $Cd - Command

LINK-86 normally generates the CMD file on the same drive as the first object file in the command line. The $C option instructs LINK-86 to place the CMD file on the drive specified by the character following the $C, or to suppress the generation of a command file if you specify $CZ. This option also applies to OVR files if you are using LINK-86 to create overlays (see Section 8).

7.12.2 $Ld - Library

LINK-86 normally searches on the default drive for Run-time Subroutine Libraries that are linked automatically. The $L option directs LINK-86 to search the specified drive for these library files.

7.12.3 $Md - Map

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

7.12.4 $Od - Object

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 files that are searched automatically (see Section 7.12.2).

7.12.5 $Sd - Symbol

LINK-86 normally generates a Symbol file on the same drive as the CMD file. The $S option directs LINK-86 to place the Symbol file on the drive specified by the character following the $S, or to suppress the generation of a symbol file if you specify $SZ.

7.13 Command Line Errors

If LINK-86 detects any kind of command line error, it prints the message

```
SYNTAX ERROR
```

echoes the command tail up to the point where the error occurs, and follows it with a question mark.

For example,

```
A>link86 a, b, c; d
SYNTAX ERROR
A, B, C;?

A>link86 longfilename
SYNTAX ERROR
LONGFILEN?
```

EOF

Section 8

Overlays

8.1 Introduction

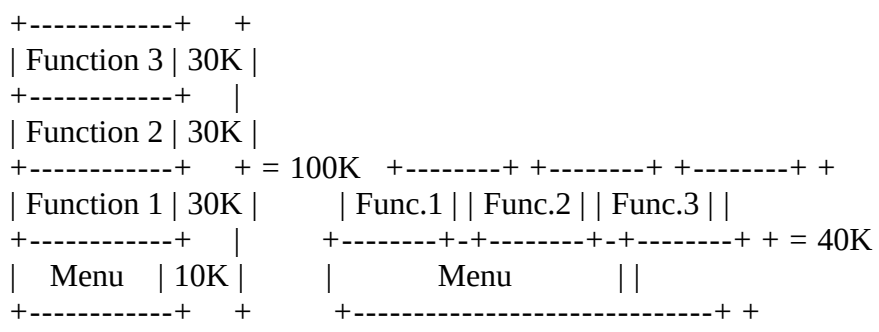
This section describes how to use LINK-86 to create programs comprised of separate files called overlays. The advantage of overlays is that they share the same memory locations, so you can write large programs that run in a limited memory environment.

Overlays are also important if you are programming in a high-level language because most compilers generate OBJ files that assume the Small memory model (see Section 7.5.2). The Small model means that when you link the OBJ files with the Run-time Subroutine Library (RSL), the size of the code or data in the CMD file must be 64K or less.

You can have multiple OBJ files, each of which has less than 64K code or data, but you cannot link them together with the RSL to create a CMD file with more than 64K of code or data. LINK-86 outputs an error message if you attempt to do so (see Appendix G). Thus, the compiler determines the upper limit on the size of any program, but the size limit is not encountered until link-time.

By using a modular design, you can write a large program so that it need not reside in memory all at once. For example, many application programs are menu-driven, with the user selecting one of a number of functions to perform. Because the functions are separate and invoked sequentially, there is no reason for them to reside in memory simultaneously. When one of the functions is complete, control returns to the menu portion of the program, from which the user selects the next function. Using overlays, you can divide such a program into separate subprograms, which can be stored on disk and loaded only when required.

Figure 8-1 illustrates the concept of overlays. Suppose a menu-driven application program consists of three separate user-selectable functions. If each function requires 30K of memory, and the menu portion requires 10K, then the total memory required for the program is 100K as shown in Figure 8-1a. However, if the three functions are designed as overlays as shown in Figure 8-1b, the program requires only 40K because all three functions share the same memory locations.



8-1a. Without Overlays 8-1b. Separate Overlays

Figure 8-1. Using Overlays in a Large Program

You can also create nested overlays in the form of a tree structure, where each overlay can call other overlays up to a maximum nesting level determined by the Overlay Manager. Section 8.3 describes the command line syntax for creating nested overlays.

Figure 8-2 illustrates such an overlay structure. The top of the highest overlay determines the total amount of memory required. In Figure 8-2, the highest overlay is SUB4. Note that this is substantially less memory than would be required if all the functions and subfunctions had to reside in memory simultaneously.

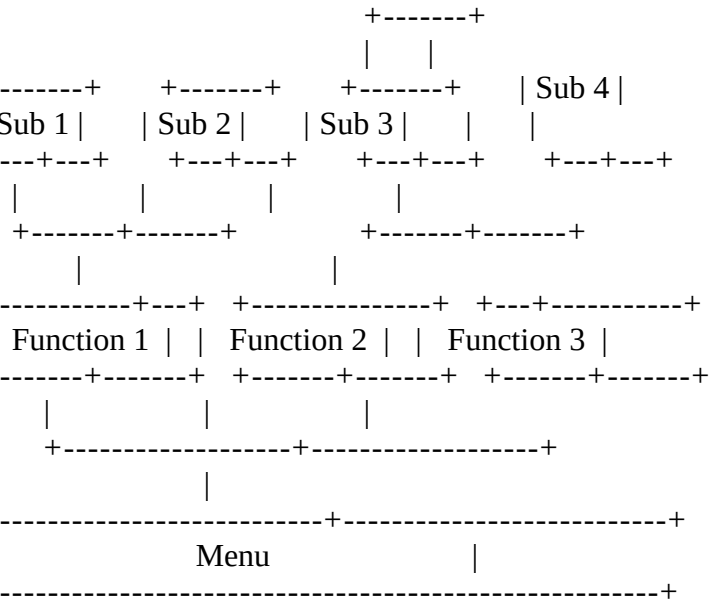


Figure 8-2. Tree Structure of Overlays

8.2 Writing Programs that Use Overlays

There are two ways to write programs that use overlays. The first method involves no special coding, but has two restrictions. The first restriction is that all overlays must be on the default drive. The second restriction is that the overlay names are determined at translation-time and cannot be changed at run-time.

The second method requires a more involved calling sequence, but does not have either of the restrictions of the first method.

8.2.1 Overlay Method 1

To use the first method, you declare an overlay as an external label in the

module where it is referenced. The overlay itself is simply a program that ends with a RET instruction.

For example, the following RASM-86 program is a root module having one overlay:

```
; ROOT.A86
; --------
;
        CSEG
        EXTRN   overlay1:near
root:   mov     bx,offset root_message
        mov     cl,9            ; Print string function number
        int     224             ; Ask BDOS to do it
        mov     bx,offset overlay_message
        call    overlay1        ; Call the overlay
        retf                    ; Return to the Operating System
;
        DSEG
root_message    DB      'root',0DH,0AH,'$'
overlay_message DB      'overlay 1',0DH,0AH,'$'
;
        END
```

with the overlay OVERLAY1.A86 defined as follows:

```
; OVERLAY1.A86
; ------------
;
        CSEG
overlay1:mov    cl,9            ; Print string function number
        int     224             ;   passed as parameter.
        ret                     ; Return to root module
```

Note that when you pass parameters to an overlay, you must ensure that the number and type of the parameters agree between the calling program and the overlay itself.

When the program runs, ROOT.CMD first displays the message 'root' at the console. The CALL statement then transfer control to the Overlay Manager. The Overlay Manager loads the file OVERLAY1.OVR from the default drive and transfers control to it.

When the overlay receives control, it displays the message 'overlay 1' at the console. OVERLAY1 then returns control directly to the statement following the CALL statement in ROOT.CMD. The program then continues from that point.

If the requested overlay is already in memory, the Overlay Manager does not reload it before transferring control.

The following constraints apply to Overlay Method 1:

1) The label used in the CALL statement is the actual name of the OVR file loaded by the Overlay Manager, so the two names must agree.

2) The name of the entry point to an overlay need not agree with the name used in the calling sequence. You should use the same name to avoid confusion.

3) The Overlay Manager loads overlays only from the drive that was the default drive when the root module began execution. The Overlay Manager disregards any changes in the default drive that occur after the root module begins execution.

4) The names of the overlays are fixed. To change the names of the overlays, you must edit, reassemble, and relink the program.

5) No non-standard statements are needed. Thus, you can postpone the decision on whether or not to create overlays until link-time.

8.2.2 Overlay Method 2

In some applications, it is useful to have greater flexibility with overlays, such as the ability to load overlays from different drives, or the ability to determine the name of an overlay from the console or a disk file at run-time.

To do this, a program must declare an explicit entry point into the Overlay Manager as follows:

```
EXTRN   ?ovlay:near
```

This entry point requires two parameters. The first is the offset of a 10-character string specifying the name of the overlay to load with an optional drive code in the standard format: "d:filename".

The second parameter is the Load Flag. If the Load Flag is 1, the Overlay Manager loads the specified overlay whether or not it is already in memory. If the Load Flag is 0, then the Overlay Manager loads the overlay only if it is not already in memory.

Note that the parameters are not passed in registers or on the stack, but as shown in the code sequence below, they follow the statement:

```
CALL    ?ovlay
```

in the Code Segment.

Using this method, the example illustrating Method 1 appears as follows:

```
; ROOT2.A86
; ---------
;
        CSEG
        EXTRN   ?ovlay:near     ; Entry point of Overlay Manager
root:   mov     bx,offset root_message
        mov     cl,9            ; Print string function number
```

```
        int     224             ; Ask BDOS to do it
        mov     bx,offset overlay_message
        call    ?ovlay          ; Call the Overlay Manager
        DW      overlay_name    ; Offset of overlay name
        DB      0               ; Load Flag
        ret                     ; Return to the Operating System
;
        DSEG
root_message    DB      'root',0DH,0AH,'$'
overlay_message DB      'overlay 1',0DH,0AH,'$'
overlay_name    DB      'OVERLAY1  '    ; Name of overlay to load (10 chars)
;
        END
```

The file OVERLAY1.A86 is the same as the previous example.

At run-time, the statement:

```
        CALL    ?ovlay
```

directs the Overlay Manager to load OVERLAY1.OVR from the default drive, because that is the current value of the variable "overlay_name", and then transfers control to it. When OVERLAY1.OVR finishes processing, control returns to the statement following the invocation.

In this example, the variable "overlay_name" is assigned the value "OVERLAY1". However, you could also supply the overlay name as a character string from some other source, such as the console.

The following constraints apply to Overlay Method 2:

1) You can specify a drive code, so the Overlay Manager can load overlays from drives other than the default drive. If you do not specify a drive code, the Overlay Manager uses the default drive as described in Method 1.

2) If you pass any parameters to the overlay, they must agree in number and type with the parameters expected by the overlay.

8.2.3 General Overlay Constraints

The following general constraints apply when you use LINK-86 to create overlays:

1) Each overlay has only one entry point. The Overlay Manager assumes that this entry point is at the load address of the overlay.

2) You cannot make a forward (upward) reference from a module to entry points in overlays higher on the tree. The only exception is a reference to the main entry point of the overlay as described above. You can make backward (downward) references to entry points in overlays lower on the tree or in the root module.

3) Common segments that are declared in one module cannot be initialized by a module higher in the tree. LINK-86 ignores any attempts to do so.

4) You can nest overlays to a depth of 5 levels.

5) The Overlay Manager uses the default buffer located at 80H in the Data Segment, so user programs should not depend on data stored in this buffer.

8.3 Command Line Syntax

You specify overlays in the LINK-86 command line by enclosing each overlay specification in parentheses. You must explicitly include the Overlay Manager in the command line, unless you are writing in a high-level language such as PL/I-86 or CB86. In that case, the Overlay Manager is automatically included from the Run-time Subroutine Library of the language you are using.

You can specify an overlay in one of the following forms:

```
A>link86 root,ovlmgr(overlay1)
A>link86 root,ovlmgr(overlay1,part2,part3)
A>link86 root,ovlmgr(overlay1=part1,part2,part3)
```

The first form produces the file OVERLAY1.OVR from the file OVERLAY1.OBJ. The second form produces the file OVERLAY1.OVR from OVERLAY1.OBJ, PART2.OBJ and PART3.OBJ. The third form produces the file OVERLAY1.OVR from PART1.OBJ, PART2.OBJ and PART3.OBJ.

In the command line, a left parenthesis indicates the start of a new overlay specification, and also indicates the end of the group preceding it. All files to be included at any overlay must appear together, without any intervening overlay specifications. You can use spaces to improve readability, and commas to separate parts of a single overlay. However, do not use commas to set off the overlay specifications from the root module or from each other.

For example, the following command line is invalid:

```
A>link86 root(overlay1),moreroot,ovlmgr
```

The correct command is:

```
A>link86 root,moreroot,ovlmgr(overlay1)
```

To nest overlays, you must specify them in the command line with nested parentheses. For example, the following command line creates the overlay system shown in Figure 8-2:

```
A>link86 menu,ovlmgr(func1(sub1)(sub2))(func2)(func3(sub3)(sub4))
```

EOF

Section 9

LIB-86

LIB-86 is a utility program for creating and maintaining library files that contain 8086 object modules. These modules can be produced by Digital Research's 8086 language translators such as RASM-86, PL/I-86, and CB86, or by any other translators that produce modules in Intel's 8086 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.

9.1 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, which is a decimal number indicating the percent of the available memory that LIB-86 used during processing. Figure 9-1 shows the operation of LIB-86.

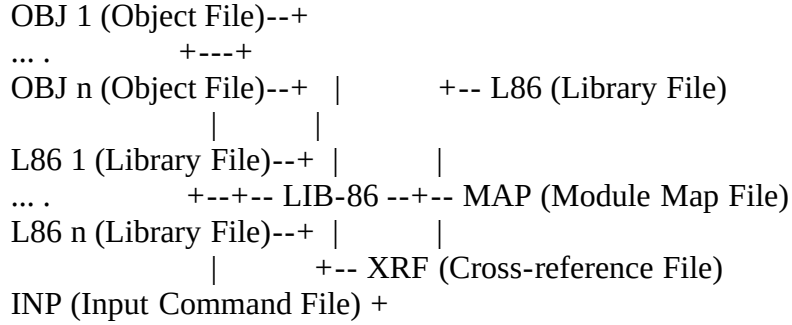


Figure 9-1. LIB-86 Operation

Table 9-1 shows the filetypes that LIB-86 recognizes.

Table 9-1. LIB-86 Filetypes

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

9.2 Halting LIB-86

You can halt LIB-86 during processing by pressing any console key. LIB-86 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.

9.3 LIB-86 Command options

When you invoke LIB-86, you can specify optional parameters in the command line that control the operation. Table 9-2 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 9-2. LIB-86 Command Line Options

Option	Purpose	Abbr.
DELETE	Delete a Module from a Library file	D
EXTERNALS	Show EXTERNALS in a Library file	E
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

9-4 Creating and Updating Libraries

You can create or update libraries using a command line of the general form:

LIB86 <library file> = <file 1> {[switches]}{,<file 2>, ..., <file n>}

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

LIB-86 reads the files specified by <file 1> through <file n> and produces the library file. If <file 1> through <file n> 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; that is, they can be OBJ or L86 files, or a combination of the two.

Modules in a library need not be 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 that are declared in earlier modules in the library) so that 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.

9.4.1 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.

9.4.2 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
```

9.4.3 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 <new library> = <old library> [REPLACE [<replace list>]]
```

where <new library> is the name of the file that LIB86 creates; <old library> is the name of the file (that can be the same as <new library>) containing the module you want to replace; and <replace list> contains one or more module names of the form:

<module name> = <file name>

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 that replaces 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).

9.4.4 Deleting a Module

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

LIB86 <new library> = <old library> [DELETE [<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 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.

9.4.5 Selecting a Module

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

LIB86 <new library> = <old library> [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.

9.5 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 9.7.

9.5.1 Cross-reference File

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

LIB86 <library name> [XREF]

LIB-86 produces the file <library name>.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 that were processed.

9.5.2 Library Module Map

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

LIB86 <library name> [MAP]

LIB-86 produces the file <library name>.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 that were 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]

9.5.3 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 <library name> [MODULES]
LIB86 <library name> [SEGMENTS]
LIB86 <library name> [PUBLICS]
LIB86 <library name> [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]]

9.6 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 <file name> [INPUT]

If <file name> 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]
```

9.7 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 that is 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:

```
$Md - MAP file destination drive
$Od - OBJ or L86 source file location
$Xd - XRF file destination drive
```

where d is a drive name (A-P). For the MAP and XRF files, d 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 processes 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
```

EOF

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. The following table shows the four differences:

Table A-1. Mnemonic Differences

Mnemonic Function	RASM-86	Intel
Intra-segment short jump:	JMPS	JMP
Inter-segment jump:	JMPF	JMP
Inter-segment return:	RETF	RET
Inter-segment call:	CALLF	CALL

EOF

Appendix B

Reserved Words

Table B-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

CODEMACRO ELSE GROUP NOLIST RS
CSEG END IF ORG RW
DB ENDIF IFLIST PAGESIZE SIMFORM
DD ENDM INCLUDE PAGEWIDTH SSEG
DSEG EQU LIST PUBLIC TITLE
DW ESEG NAME RB
EJECT EXTRN NOIFLIST RD

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

Default Segment Names

CODE DATA EXTRA STACK

Segments Descriptors

BYTE LOCAL PARA STACK
COMMON PAGE PUBLIC WORD

External Descriptors

ABS DWORD NEAR

BYTE FAR WORD

EOF

Appendix C

RASM-86 Instruction Summary

Table C-1. RASM-86 Instruction Sumary

Mnemonic	Description	Section
AAA	ASCII adjust for Addition	4.3
AAD	ASCII adjust for Division	4.3
AAM	ASCII adjust for Multiplication	4.3
AAS	ASCII adjust for Subtraction	4.3
ADC	Add with Carry	4.3
ADD	Add	4.3
AND	And	4.3
CALL	Call (intra segment)	4.5
CALLF	Call (inter Segment)	4.5
CBW	Convert Byte to Word	4.3
CLC	Clear Carry	4.6
CLD	Clear Direction	4.6
CLI	Clear Interrupt	4.6
CMC	Complement Carry	4.6
CMP	Compare	4.3
CMPS	Compare Byte or Word (of string)	4.4
CMPSB	Compare Byte (of string)	4.4
CMPSW	Compare Word (of string)	4.4
CWD	Convert Word to Double Word	4.3
DAA	Decimal Adjust for Addition	4.3
DAS	Decimal Adjust for Subtraction	4.3
DEC	Decrement	4.3
DIV	Divide	4.3
ESC	Escape	4.6
HLT	Halt	4.6
IDIV	Integer Divide	4.3
IMUL	Integer Multiply	4.3
IN	Input Byte or Word	4.2
INC	Increment	4.3
INT	Interrupt	4.5
INTO	Interrupt on Overflow	4.5
IRET	Interrupt Return	4.5
JA	Jump on Above	4.5
JAE	Jump on Above or Equal	4.5
JB	Jump on Below	4.5
JBE	Jump on Below or Equal	4.5
JC	Jump on Carry	4.5
JCXZ	Jump on CX Zero	4.5
JE	Jump on Equal	4.5
JG	Jump on Greater	4.5
JGE	Jump on Greater or Equal	4.5
JL	Jump on Less	4.5
JLE	Jump on Less or Equal	4.5

POP	Pop	4.2
POPF	Pop Flags	4.2
PUSH	Push	4.2
PUSHF	Push Flags	4.2
RCL	Rotate through Carry Left	4.3
RCR	Rotate through Carry Right	4.3
REP	Repeat	4.4
REPE	Repeat While Equal	4.4
REPNE	Repeat While Not Equal	4.4
REPNZ	Repeat While Not Zero	4.4
REPZ	Repeat While Zero	4.4
RET	Return (intra segment)	4.5
RETF	Return (inter segment)	4.5
ROL	Rotate Left	4.3
ROR	Rotate Right	4.3
SAHF	Store AH into Flags	4.2
SAL	Shift Arithmetic Left	4.3
SAR	Shift Arithmetic Right	4.3
SBB	Subtract with Borrow	4.3
SCAS	Scan Byte or Word (of string)	4.4
SCASB	Scan Byte (of string)	4.4
SCASW	Scan Word (of string)	4.4
SHL	Shift Left	4.3
SHR	Shift Right	4.3
STC	Set Carry	4.6
STD	Set Direction	4.6
STI	Set Interrupt	4.6
STOS	Store Byte or Word (of string)	4.4
STOSB	Store Byte (of string)	4.4
STOSW	Store Word (of string)	4.4
SUB	Subtract	4.3
TEST	Test	4.3
WAIT	Wait	4.6
XCHG	Exchange	4.2
XLAT	Translate	4.2
XOR	Exclusive Or	4.3

EOF

Appendix D

Code-macro Definition Syntax

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

<name> ::= IDENTIFIER

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

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

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

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

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

<single$range> ::= REGISTER | NUMBERB

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

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

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

<db> ::= DB NUMBERB | DB <form$name>
<dw> ::= DW NUMBERW | DW <form$name>
<dd> ::= DD <form$name>

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

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

<modrm> ::= MODRM NUMBER7,<form$name> |
         MODRM <form$name>,<form$name>
<relb> ::= RELB <form$name>
<relw> ::= RELW <form$name>
<dbit> ::= DBIT <field$descr>{,<field$descr>}

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

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

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

EOF

Appendix E

Sample Program

CP/M RASM-86 1.4 Source: APPE.A86 Terminal Input/Output Page 1

```
                        TITLE   'Terminal Input/Output'
                        PAGESIZE 59
                        PAGEWIDTH 78
                        SIMFORM
                    ;
                    ;---- Terminal I/O Subroutines ----
                    ;
                    ; The following subroutines
                    ; are included:
                    ;
                    ; CONSTAT -- console status
                    ; CONIN   -- console input
                    ; CONOUT  -- console output
                    ;
                    ; Each routine requires CONSOLE NUMBER
                    ; in the BL-register
                    ;
                    ; +------------+
                    ; | Jump Table |
                    ; +------------+
                    ;
                        CSEG
                    ;
                    jmp_tab:
0000 E90600             0009        jmp     constat
0003 E91C00             0022        jmp     conin
0006 E92F00             0038        jmp     conout
                    ;
                    ; +------------------+
                    ; | I/O port numbers |
                    ; +------------------+
                    ;
                    ; Terminal 1:
                    ;
0010                    instat1     EQU     10H     ; Input status port
0011                    indata1     EQU     11H     ; Input port
0011                    outdata1    EQU     11H     ; Output port
0001                    readyinmask1    EQU     01H     ; Input ready mask
0002                    readyoutmask1   EQU     02H     ; Output ready mask
                    ;
                    ; Terminal 2:
                    ;
0012                    instat2     EQU     12H     ; Input status port
0013                    indata2     EQU     13H     ; Input port
```

```
0013                outdata2        EQU     13H     ; Output port


CP/M RASM-86 1.4   Source: APPE.A86      Terminal Input/Output      Page   2


0004                readyinmask2    EQU     04H     ; Input ready mask
0008                readyoutmask2   EQU     08H     ; Output ready mask
                    ;
                    ; +---------+
                    ; | CONSTAT |
                    ; +---------+
                    ;
                    ; Entry: BL-reg = terminal number
                    ; Exit:  AL-reg = 0 if not ready
                    ;               0FFH if ready
                    ;
                    constat:
0009 53E84700       0054        push bx! call okterminal
                    constat1:
000D 52                     push    dx
000E B600                     mov     dh,0          ; Read status port
0010 8A970000     R             mov     dl,instatustab [BX]
0014 EC                     in      al,dx
0015 22870600     R             and     al,readyinmasktab [BX]
0019 7402           001D        jz      constatout
001B B0FF                     mov     al,0FFH
                    constatout:
001D 5A5B0AC0C3                 pop dx! pop bx! or al,al! ret
                    ;
                    ; +-------+
                    ; | CONIN |
                    ; +-------+
                    ;
                    ; Entry: BL-reg = terminal number
                    ; Exit:  AL-reg = read character
                    ;
0022 53E82E00       0054 conin:  push bx! call okterminal
0026 E8E4FF         000D conin1: call    constat1       ; Test status
0029 74FB           0026        jz      conin1
002B 52                     push    dx            ; Read character
002C B600                     mov     dh,0
002E 8A970200     R             mov     dl,indatatab [BX]
0032 EC                     in      al,dx
0033 247F                     and     al,7FH        ; Strip parity bit
0035 5A5BC3                     pop dx! pop bx! ret
                    ;
                    ; +--------+
                    ; | CONOUT |
                    ; +--------+
                    ;
```

```
                ; Entry: BL-reg = terminal number
                ;        AL-reg = character to print
                ;
0038 53E81800       0054 conout: push bx! call okterminal
003C 52                  push    dx
003D 50                  push    ax
003E B600                mov     dh,0          ; Test status
0040 8A970000     R      mov     dl,instatustab [BX]
                conout1:
0044 EC                  in      al,dx
0045 22870800     R      and     al,readyoutmasktab [BX]
0049 74F9         0044   jz      conout1
004B 58                  pop     ax            ; Write byte
004C 8A970400     R      mov     dl,outdatatab [BX]
0050 EE                  out     dx,al
0051 5A5BC3              pop dx! pop bx! ret
                ;
                ; +------------+
                ; | OKTERMINAL |
                ; +------------+
                ;
                ; Entry: BL-reg = terminal number
                ;
                okterminal:
0054 0ADB                or      bl,bl
0056 740A         0062   jz      error
0058 80FB03              cmp     bl,length instatustab + 1
005B 7305         0062   jae     error
005D FECB                dec     bl
005F B700                mov     bh,0
0061 C3                  ret
                ;
0062 5B5BC3       error: pop bx! pop bx! ret     ; Do nothing
                ;
                ;------ End of Code Segment -----
                ;
                ; +--------------+
                ; | Data Segment |
                ; +--------------+
                ;
                     DSEG
                ;
                ; +------------------------+
                ; | Data for each terminal |
                ; +------------------------+
```



```
                ;
```

```
0000 1012           instatustab     DB      instat1,instat2
0002 1113           indatatab       DB      indata1,indata2
0004 1113           outdatatab      DB      outdata1,outdata2
0006 0104           readyinmasktab  DB      readyinmask1,readyinmask2
0008 0208           readyoutmasktab DB      readyoutmask1,readyoutmask2
                ;
                ;------ End of File -------------
                ;
                    END


End of assembly.  Number of errors:   0.  Use factor:  1%


EOF
```

Appendix F

RASM-86 Error Messages

RASM-86 displays two kinds of error messages:

- non-recoverable errors
- diagnostics

Non-recoverable errors occur when RASM-86 is unable to continue assembling. Table F-1 lists the non-recoverable errors RASM-86 can encounter during assembly.

Table F-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.
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 that 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 available.
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 F-2 shows the RASM-86 diagnostic error messages by number, and gives a brief explanation of the error.

Table F-2. RASM-86 Diagnostic Error messages

Error Message	Meaning

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
```

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
```

ERROR NO: 8 ILLEGAL INSTRUCTION OPERANDS

An instruction operand is improperly formed. For example,

```
MOV [BP+SP],1234
CALL BX+l
```

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 OAGH
  TITLE UNQUOTED STRING
```

ERROR NO: 12 NESTED IF ILLEGAL - IF IGNORED

The maximum nesting level for IF statements has been exceeded.

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.

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
```

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.

EOF

Appendix G

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 described in Table G-1.

Table G-1. LINK-86 Error Messages

Error Message	Meaning
CANNOT CLOSE	LINK-86 cannot close an output file. You should take appropriate action after checking to see if the correct disk is in the drive, and that 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 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. Correct the problem in your source file.
DISK WRITE ERROR	A file cannot be written properly, probably due to a full disk.
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	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.

SEGMENT ATTRIBUTE ERROR

The <Align type> or <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 re-link.

SYMBOL TABLE OVERFLOW

LINK-86 ran out of Symbol Table space. Either reduce the number or length of symbols in the program, or re-link on a system with more memory available.

SYNTAX ERROR

LINK-86 detected a syntax error in the command line, probably due to 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.

TARGET OUT OF RANGE

The target of a fixup cannot be reached from the location of the fixup. (What is a "fixup"? It is not in the index...)

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 file system or later, due to its use of the random I/O functions.

EOF

Appendix H

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

Error 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 that 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.
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.

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 needs a version 2 file system or later, due to its use of the random I/O functions.

EOF

Appendix I

XREF-86 Error Messages

During the course of operation, XREF-86 can display error messages. Table I-1 shows the error messages and a brief explanation of their cause.

Table I-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 that 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.
SYMBOL FILE ERROR	XREF-86 issues this message when it reads an invalid SYM file. Specifically, a line in the SYM file that is 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.

EOF