FlexOS™ 286 Programmer's Utilities Guide

1073-2043-001

Foreword

The FlexOS™ 286 Programmer's Utilities Guide (cited as the Programmer's Utilities Guide) assumes that you are familiar with the CP/M-86 or FlexOS operating system environment. It also assumes that you are familiar with the basic elements of 8086/286 assembly language programming.

Part I of the Programmer's Utilities Guide describes several utility programs that aid programmers and system designers in the software development process. Collectively, these utilities allow you to assemble 8086 and 80286 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 tools described in this guide can be run either on the 8086 microprocessor under CP/M-86 or on the 80286 microprocessor under FlexOS 286.

RASM-86™ is an assembler that translates 8086, 80186, or 80286 assembly language statements into a relocatable object file in the Intel™ format. RASM-86 facilities include assembly of Intel mnemonics, assembly-time expressions, conditional assembly, page formatting of listing files, and powerful code-macro capabilities.

The Programmer's Utilities Guide is divided into the following sections:

Section 1	The overall operation of RASM-86 and its optional run-time parameters.
Section 2	Elements of RASM-86 assembly language, including the character set, delimiters, constants, identifiers, operators, expressions, and statements.
Section 3	The various RASM-86 directives that control the assembly process.

Section 4	A brief description of the RASM-86 instructions for data transfer, mathematical operations, string manipulation, control transfer, and processor control.
Section 5	The code-macro facilities of RASM-86.
Section 6	XREF-86™, an assembly language cross-reference program used with RASM-86.
Section 7	LINK 86™, the linkage editor that combines relocatable object modules into an absolute file that runs under CP/M-86 or FlexOS.
Section 8	How to use LINK 86 to produce overlays.
Section 9	How to use LIB-86™, the software librarian that creates and manages libraries.

Sections 10 through 16 explain how to use SID-286 , the Symbolic Instruction Debugger.

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

TYPOGRAPHICAL CONVENTIONS USED IN THIS GUIDE

This guide uses the following notation to describe commands:

```
    command  parameter  [option]
```

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

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

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

```
A>rasm86 test
```

illustrates a specific usage of the RASM86 command.

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

Contents

Tables

The RASM-86 Assembler

1.1 Introduction

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

1.2 RASM-86 Operation

The RASM-86 assembler converts source files containing 8086, 8087, 80186, 80286, and 80287 instructions into machine language object files. RASM-86 processes an assembly language source file in three passes and can produce three output files from one source file as shown in Figure 1-1.

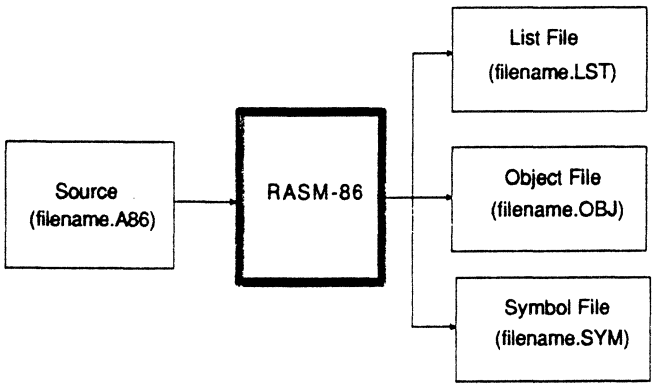


Figure 1-1. RASM-86 Source and Object Files

The LST list file contains the assembly language listing with any error messages. The OBJ object file contains the object code in Intel 8086 and 80286 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.

1.3 RASM-86 Command Syntax

Invoke RASM-86 with the following command form:

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

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

The **d:** is an optional drive specification denoting the source file's location. The drive specification is not needed if the source is on the current drive. The **typ** is the optional filetype, which can be any valid filetype of I to 3 characters. If no filetype is specified, filetype A86 is assumed. The **run-time parameters** are described below in Section 1.3.1.

1.3.1 RASM-86 Run-Time Parameters

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

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

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

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

All run-time parameters are optional, and you can enter them in the command line in any order. Enter the dollar sign only once at the beginning of the parameter string. Spaces can separate parameters, but are not required. However, no space is permitted between a parameter and its device name.

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

```
SYNTAX ERROR
```

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

A, O, P, and S Parameters

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

```
$ AA
```

specifies the source file on drive A.

The A, O, P, and S parameters specify the following:

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

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

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

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

IFILENAME Parameter

If a CP/M filename is preceded by an I (upper case i), the contents of the file is included at the beginning of the module being assembled. If no filename extension is specified, RASM-86 assumes an extension of A86. The file specified must be a CP/M file. This parameter will not work with PC DOS files.

L Parameter

The L parameter directs RASM-86 to include local symbols in the object file so that they appear in the SYM file created by LINK-86. Otherwise, only public symbols appear in the SYM file. You can use the SYM file with the Symbolic Instruction Debugger, SID-286.., to simplify program debugging.

NC Parameter

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

186 and 286 Parameters

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

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

1.3.2 RASM-86 Command Line Examples

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

Command Line	Result
`A>rasm86 io`	Assembles file IO.A86 and produces IO.OBJ, IO.LST, and IO.SYM, all on the default drive.
`A>rasm86 io.asm $ ad sz`	Assembles file IO.ASM on drive D and produces IO.LST and IO.OBJ. Suppresses the symbol file.
`A>rasm86 io $ py sx`	Assembles file IO.A86, produces IO.OBJ, and sends listing directly to printer. Also outputs symbols on console.
`A>rasm86 io $ Ifirst`	Assembles file IO.A86 with the contents of the file, first.a86, appearing at the beginning of the module. Then produces IO.OBJ, IO.LST, and IO.SYM, all on the default drive.
`A>rasm86 io $ lo`	Includes local symbols in IO.OBJ.

After the command to invoke RASM-86 is given, the following sign-on message is displayed:

```
-----------------------------------------------------
RASM-86 (87)    12/14/85                  Version 1.3
Serial No. xxxx-0000-654321   All Rights Reserved
Copyright (C) 1982,1985    Digital Research, Inc.
-----------------------------------------------------
```

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

```
NO FILE
```

and stops processing.

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

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

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

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

1.4 Stopping RASM-86

To stop the assembly at any time, press any key on the console keyboard. When you press a key, RASM-86 responds

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

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

End of Section 1

Elements of RASM-86 Assembly Language

2.1 Introduction

This section describes the elements of the RASM-86 assembly language. RASM-86 elements are described in the following order:

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

2.2 RASM-86 Character Set

RASM-86 recognizes a subset of the ASCII character set. Valid RASM-86 characters are the following alphanumeric characters, special characters, and non-printing characters.

Valid Alphanumeric Characters:

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

Valid Special Characters:

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

Valid Nonprinting Characters:

space, tab, carriage return, and line-feed

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

2.3 Tokens and Separators

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

2.4 Delimiters

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

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

Table 2-1. Separators and Delimiters

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

Table 2-1. (continued)

Character	Name	Use
*	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

2.5 Constants

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

2.5.1 Numeric Constants

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

Table 2-2. Radix Indicators for Constants

Indicator	Constant Type	Base
B	binary	2
O	octal	8
Q	octal	8
D	decimal	10
H	hexadecimal	16

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

A constant is thus a sequence of digits followed by an optional radix indicator, where the digits are in the range for the radix. Binary constants must be composed of zeros and ones. Octal digits range from 0 to 7; decimal digits range from 0 to 9. Hexadecimal constants contain decimal digits and the hexadecimal digits A (10D), B (11D), C (12D), D (13D), E (14D), and F (15D). The leading character of a hexadecimal constant must be a decimal digit so RASM-86 cannot confuse a hex constant with an identifier. The following are valid numeric constants:

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

2.5.2 Character String Constants

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

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

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

Table 2-3. String Constant Examples

String in source text	As processed by RASM-86
'a'	a
'Ab''Cd'	Ab'Cd
'I like CP/M'	I like CP/M
''''	'
'ONLY UPPERCASE'	ONLY UPPERCASE
'only lowercase'	only lowercase

2.6 Identifiers

The following rules apply to all identifiers:

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

There are two types of identifiers:

Keywords

Symbols

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

2.6.1 Keyword Identifiers

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

RASM-86 recognizes five types of keywords:

- instructions
- directives
- operators
- registers
- predefined numbers

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

Three keywords, BYTE, WORD, and DWORD, are predefined numbers. The values of these numbers are 1, 2, and 4, respectively. RASM-86 also associates a Type attribute with each of these numbers. The keyword's Type attribute is equal to the keyword's numeric value.

Table 2-4. Register Keywords

Register Symbol	Size (bytes)	Numeric Value	Meaning
AH	1	100 B	Accumulator High Byte
BH	1	111 B	Base Register High Byte
CH	1	101 B	Count Register High Byte
DH	1	110 B	Data Register High Byte
AL	1	000 B	Accumulator Low Byte
BL	1	011 B	Base Register Low Byte
CL	1	001 B	Count Register Low Byte
DL	1	010 B	Data Register Low Byte
AX	2	000 B	Accumulator (full word)
BX	2	011 B	Base Register (full word)
CX	2	001 B	Count Register (full word)
DX	2	010 B	Data Register (full word)
BP	2	101 B	Base Pointer
SP	2	100 B	Stack Pointer
SI	2	110 B	Source Index
DI	2	111 B	Destination Index
CS	2	01 B	Code Segment Register
DS	2	11 B	Data Segment Register
SS	2	10 B	Stack Segment Register
ES	2	00 B	Extra Segment Register
ST	10	000 B	8087 stack top register
ST0	10	000 B	8087 stack top register
ST1	10	001 B	8087 stack top - 1 register
ST2	10	111 B	8087 stack top - 7 register

2.6.2 Symbol Identifiers

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

- variables
- labels
- numbers

Variables

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

Segment	tells which segment was being assembled when the variable was defined.
Offset	tells how many bytes are between the beginning of the segment and the location of this variable.
Type	tells how many bytes of data are manipulated when this variable is referenced.

A segment can be a code segment, a data segment, a stack segment, or an extra segment, depending on its contents and the register containing its starting address (see "Segment Control Directives" in Section 3.3). The segment's starting address is a number between 0 and 0FFFFH (65,535D). This number indicates the paragraph in memory to which the current segment is assigned, either when the program is assembled, linked, or loaded.

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

A variable has one of the following type attributes:

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

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

```
my_variable db 0
```

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

```
another_variable equ my_variable
```

Labels

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

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

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

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

```
another_label   equu   my_label
```

Numbers

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

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

is equivalent to the following:

```
mov   al,5
```

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

2.6.3 Example Identifiers

The following are valid identifiers:

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

2.7 Operators

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

RASM-86 operators fall into the following categories:

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

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

Table 2-5. RASM-86 Operator Summary

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

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

2.7.1 Arithmetic Operators

Addition and Subtraction

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

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

Syntax:

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

Example:

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

Multiplication and Division

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

Syntax:

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

Example:

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

Unary

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

Syntax:

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

Example:

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

2.7.2 Logical Operators

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

Syntax:

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

Example:

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

2.7.3 Relational Operators

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

Syntax:

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

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

Example:

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

2.7.4 Segment Override Operator

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

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

Syntax:

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

Example:

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

2.7.5 Variable Manipulation Operators

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

Syntax:

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

Example:

```
002D 000000000000 wordbuffer     dw     0,0,0
0033 0102030405   buffer         db     1,2,3,4,5
                                 •
                                 •
                                 •
0038 B80500                mov   ax,length buffer
003B B80400                mov   ax,last buffer
003E B80100                mov   ax,type buffer
0041 B80200                mov   ax,type wordbuffer
```

2.7.6 Variable Creation Operators

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

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

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

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

Syntax:

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

Examples:

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


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


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

2.7.7 Operator Precedence

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

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

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

```
15/3 + 18/9 = 5 + 2 = 7
15/(3 + 18/9) = 15/(3 + 2) = 15/5 = 3
(20*4) + ((27/9 - 4/2)) = (20*4) + (3 - 2) = 80 + 1 = 8
```

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

Table 2-6. Precedence of Operations in RASM-86

Order	Operator Type	Operators
1	Logical	XOR, OR
2	Logical	AND
3	Logical	NOT
4	Relational	EQ, LT, LE, GT, GE, NE
5	Addition/subtraction	+, -
6	Multiplication/division	*, /, MOD, SHL, SHR

Table 2-6. (continued)

Order	Operator Type	Operators
7	Unary	+, -
8	Segment override	segment_override:
9	Variable manipulators/creators LENGTH, LAST	SEG, OFFSET, PTR, TYPE,
10	Parentheses/brackets	( ), [ ]
11	Period and Dollar	., $

2.8 Expressions

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

- segment value
- offset value
- type

Both variables and labels are address expressions. An address expression is not a number, but its components are numbers. You can combine numbers with operators such as PTR to make an address expression.

A numeric expression evaluates to a number. It contains no variables or labels, only numbers and operands.

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

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

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

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

2.9 Statements

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

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

2.9.1 Instruction Statements

The following is the syntax for an instruction statement:

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

The fields are defined as follows:

label	A symbol followed by a colon defines a label at the current value of the location counter in the current segment. This field is optional.
prefix	Certain machine instructions such as LOCK and REP can prefix other instructions. This field is optional.
mnemonic	A symbol defined as a machine instruction, either by RASM-86 or by an EQU directive. This field is optional unless preceded by a prefix instruction. If you omit this field, no operands can be present, although the other fields can appear. Section 4 describes the RASM-86 mnemonics.
operand(s)	An instruction mnemonic can require other symbols to represent operands to the instruction. Instructions can have zero, one, or two operands.
comment	Any semicolon appearing outside a character string begins a comment. A comment ends with a carriage return. This field is optional, but you should use comments to facilitate program maintenance and debugging.

Section 3 describes the RASM-86 directives.

2.9.2 Directive Statements

The following is the syntax for a directive statement:

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

The fields are defined as follows:

name	Names are legal for CSEG, DSEG, ESEG, SSEG, GROUP, DB, DW, DD, RB, RW, RD, RS, and EQU directives. The name is required for the EQU and GROUP directives, but it is optional for the other directives. Unlike the label field of an instruction, the name field of a directive is never terminated with a colon.
directive	One of the directive keywords defined in Section 3.
operand(s)	Analogous to the operands for instruction mnemonics. Some directives, such as DB and DW allow any operand; others have special requirements.
comment	Exactly as defined for instruction statements in Section 2.9.1.

End of Section 2

RASM-86 Directives

3.1 Introduction

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

RASM-86 directives are grouped into the following categories:

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

3.2 Assembler Directive Syntax

The following is the general syntax for a directive statement:

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

The fields are defined as follows:

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

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

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

3.3 Segment Control Directives

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

CSEG
DSEG
ESEG
SSEG
GROUP

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

3.3.1 The 8086/80286 Segmented Architecture

The address space of an 8086 or an 80286 processor can be subdivided into an arbitrary number of units called segments. Each segment is comprised of contiguous memory locations, up to 64K bytes in length, making up logically independent and separately addressable units. Each segment must have a base address specifying its starting location in the memory space. Each segment base address must begin on a paragraph boundary, a boundary divisible by 16.

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

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

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

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

3.3.2 CSEG, DSEG, ESEG, and SSEG Directives

Every instruction and variable in a program must be contained in a segment. The segment directives described in this section allow you to specify the attributes of a segment or a group of segments of the same type. Create a segment and name it by using the segment directive syntax:

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

where seg_directive is one of the following:

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

The optional parameters are described below. Examples illustrating how segment directives are used are provided at the end of this section.

seg_name

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

Table 3-1. Default Segment Names

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

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

align_type

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

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

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

Table 3-2. Default Align Types

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

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

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

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

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

combine_type

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

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

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

The RASM-86 combine types are defined as follows:

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

COMMON	means that the segment shares identical memory locations with other segments of the same name. Offsets inside a COMMON segment are absolute unless the segment is contained in a GROUP (see "Group Directive" in this section).
STACK	is similar to PUBLIC, in that the storage allocated for STACK segments is the sum of the STACK segments from each module. However, instead of concatenating segments with the same name, the linkage editor overlays STACK segments against high memory, because stacks grow downward from high addresses to low addresses when the program runs.
LOCAL	means that the segment is local to the program being assembled, and the linkage editor will not combine it with any other segments.
ABSOLUTE SEGMENT	causes RASM-86 to determine the load-time position of the segment during assembly, rather than allowing its position to be determined by the linkage editor, or at load time.

class_name

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

Table 3-3. Default Class Name for Segments

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

Examples:

The following are examples of segment directives:

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

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

3.3.3 GROUP Directive

group_name GROUP segment_name1, segment_name2, ...

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

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

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

See Section 7.15 for more information on the grouping and other link processes.

3.4 Linkage Control Directives

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

```
END
NAME
PUBLIC
EXTRN
```

3.4.1 END Directive

```
END [start_label]
```

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

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

3.4.2 NAME Directive

```
NAME  module_name
```

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

3.4.3 PUBLIC Directive

```
PUBLIC name [, name, ...]
```

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

3.4.4 EXTRN Directive

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

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

The external id uses the form:

```
symbol:type
```

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

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

For example,

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

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

3.5 Conditional Assembly Directives

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

IF
ELSE
ENDIF

3.5.1 IF, ELSE, and ENDIF Directives

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

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

    ENDIF
```

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

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

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

3.6 Symbol Definition Directive

The available symbol definition directive is:

```
    EQU
```

3.6.1 EQU Directive

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

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

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

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

3.7 Data and Memory Directives

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

DB
DW
DD
RS
RB
RW
RD

3.7.1 DB Directive

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

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

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

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

- segment
- offset
- type
- length

The segment and offset attributes determine the symbol's memory reference; the type attribute specifies single bytes; and the length attribute tells the number of bytes reserved.

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

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

3.7.2 DW Directive

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

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

The following are examples of DW directives:

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

3.7.3 DD Directive

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

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

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

3.7.4 RS Directive

[symbol] RS numeric_expression

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

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

3.7.5 RB Directive

[symbol] RB numeric_expression

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

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

3.7.6 RW Directive

[symbol] RW numeric_expression

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

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

3.7.7 RD Directive

```
[symbol] RD numeric_expression
```

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

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

3.8 Output Listing Control Directives

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

EJECT
IFLIST
NOIFLIST
LIST
NOLIST
PAGESIZE
PAGEWIDTH
SIMFORM
TITLE

3.8.1 EJECT Directive

```
EJECT
```

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

3.8.2 NOIFLIST/IFLIST Directives

```
NOIFLIST
IFLIST
```

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

3.8.3 NOLIST and LIST Directives

```
NOLIST
LIST
```

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

3.8.4 PAGESIZE Directive

```
PAGESIZE   numeric_expression
```

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

3.8.5 PAGEWIDTH Directive

```
PAGEWIDTH  numeric_expression
```

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

3.8.6 SIMFORM Directive

```
SIMFORM
```

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

3.8.7 TITLE Directive

```
TITLE   string_constant
```

RASM-86 prints the string constant defined by a TITLE directive statement at the top of each printout page in the listing file. The title character string can be up to 30 characters in length. For example,

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

3.9 8087 Control Directives

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

```
HARD8087
AUTO8087
```

3.9.1 HARD8087 Directive

```
HARD8087
```

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

3.9.2 AUTO8087 Directive

```
AUTO8087
```

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

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

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

3.10 Miscellaneous Directives

Additional RASM-86 directives are:

INCLUDE
ORG

3.10.1 INCLUDE Directive

INCLUDE filename

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

```
INCLUDE  EQUALS.A86
```

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

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

3.10.2 ORG Directive

```
ORG numeric_expression
```

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

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

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

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

```
ORG 0
```

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

End of Section 3

RASM-86 Instruction Set

4.1 Introduction

The RASM-86 instruction set includes all 8086, 8087, 80186, and 80286 machine instructions. The general syntax for instruction statements is described in Section 2.9. This section defines the specific syntax and required operand types for each instruction without reference to labels or comments. The instruction definitions are presented in tables for easy reference.

For a more detailed description of each instruction, see the Intel assembly language reference manual for the processor you are using. For descriptions of the instruction bit patterns and operations, see the Intel user's manual for the processor you are using.

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

4.2 RASM-86 Instruction Set Summary

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

Table 4-1. RASM-86 Instruction Summary

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

Table 4-1. (continued)

Mnemonic	Description	Section
FADD64	Add Real, 64-bit	4.3.8
FADDP	Add Real and Pop	4.3.8
FBLD	Packed Decimal Load	4.3.8
FBSTP	Packed Decimal Store and Pop	4.3.8
FCHS	Change Sign	4.3.8
FCLEX/FNCLEX	Clear Exceptions	4.3.8
FCOM	Compare Real	4.3.8
FCOM32	Compare Real, 32-bit	4.3.8
FCOM64	Compare Real, 64-bit	4.3.8
FCOMP	Compare Real and Pop	4.3.8
FCOM32P	Compare Real and Pop, 32-bit	4.3.8
FCOM64P	Compare Real and Pop, 64-bit	4.3.8
FCOMPP	Compare Real and Pop Twice	4.3.8
FDECSTP	Decrement Stack Pointer	4.3.8
FDISI/FNDISI	Disable Interrupts	4.3.8
FDIV	Divide Real	4.3.8
FDIV32	Divide Real, 32-bit	4.3.8
FDIV64	Divide Real, 64-bit	4.3.8
FDIVR	Divide Real Reversed	4.3.8
FDIVR32	Divide Real Reversed, 32-bit	4.3.8
FDIVR64	Divide Real Reversed, 64-bit	4.3.8
FDIVRP	Divide Real Reversed and Pop	4.3.8
FDUP	Duplicate Top of Stack	4.3.8
FENI/FNENI	Enable Interrupts	4.3.8
FFREE	Free Register	4.3.8
FIADD16	Integer Add, 16-bit	4.3.8
FIADD32	Integer Add, 32-bit	4.3.8
FICOM16	Integer Compare, 16-bit	4.3.8
FICOM32	Integer Compare, 32-bit	4.3.8
FICOM16P	Integer Compare and Pop, 16-bit	4.3.8
FICOM32P	Integer Compare and Pop, 32-bit	4.3.8
FIDIV16	Integer Divide, 16-bit	4.3.8
FIDIV32	Integer Divide, 32-bit	4.3.8

Table 4-1. (continued)

Mnemonic	Description	Section
FIDIVR16	Integer Divide Reversed, 16-bit	4.3.8
FIDIVR32	Integer Divide Reversed, 32-bit	4.3.8
FILD16	Integer Load, 16-bit	4.3.8
FILD32	Integer Load, 32-bit	4.3.8
FILD64	Integer Load, 64-bit	4.3.8
FIMUL16	Integer Multiply, 16-bit	4.3.8
FIMUL32	Integer Multiply, 32-bit	4.3.8
FINCSTP	Increment Stack Pointer	4.3.8
FINIT/FNINIT	Initialize Processor	4.3.8
FIST16	Integer Store, 16-bit	4.3.8
FIST32	Integer Store, 32-bit	4.3.8
FIST16P	Integer Store and Pop, 16-bit	4.3.8
FIST32P	Integer Store and Pop, 32-bit	4.3.8
FIST64P	Integer Store and Pop, 64-bit	4.3.8
FISUB16	Integer Subtract, 16-bit	4.3.8
FISUB32	Integer Subtract, 32-bit	4.3.8
FISUBR16	Integer Subtract Reversed, 16-bit	4.3.8
FISUBR32	Integer Subtract Reversed, 32-bit	4.3.8
FLD	Load Real	4.3.8
FLD32	Load Real, 32-bit	4.3.8
FLD64	Load Real, 64-bit	4.3.8
FLD80	Load Real, 80-bit	4.3.8
FLDCW	Load Control Word	4.3.8
FLDENV	Load Environment	4.3.8
FLDZ	Load + 0.0	4.3.8
FLD1	Load + 1.0	4.3.8
FLDPI	Load 80-bit value for pi.	4.3.8
FLDL2T	Load $\log_2 10$	4.3.8
FLDL2E	Load $\log_2 e$	4.3.8
FLDLG2	Load $\log_{10} 2$	4.3.8
FLDLN2	Load $\log_e 2$	4.3.8
FMUL	Multiply Real	4.3.8
FMUL32	Multiply Real, 32-bit	4.3.8

Table 4-1. (continued)

Mnemonic	Description	Section
FMUL64	Multiply Real, 64-bit	4.3.8
FMULP	Multiply Real and Pop	4.3.8
FNOP	No Operation	4.3.8
FPATAN	Partial Arctangent	4.3.8
FPOP	same as FSTP ST0	4.3.8
FPREM	Partial Remainder	4.3.8
FPTAN	Partial Tangent	4.3.8
FRNDINT	Round to Integer	4.3.8
FRSTOR	Restore State	4.3.8
FSAVE/FNSAVE	Save State	4.3.8
FSCALE	Scale	4.3.8
FST	Store Real	4.3.8
FST32	Store Real, 32-bit	4.3.8
FST64	Store Real, 64-bit	4.3.8
FSTP	Store Real and Pop	4.3.8
FST32P	Store Real and Pop, 32-bit	4.3.8
FST64P	Store Real and Pop, 64-bit	4.3.8
FSTENV/FNSTENV	Store Environment	4.3.8
FSTCW/FNSTCW	Store Control Word	4.3.8
FSTSW/FNSTSW	Store Status Word	4.3.8
FSQRT	Square Root	4.3.8
FSUB	Subtract Real	4.3.8
FSUB32	Subtract Real, 32-bit	4.3.8
FSUB64	Subtract Real, 64-bit	4.3.8
FSUBP	Subtract Real and Pop	4.3.8
FSUBR	Subtract Real Reversed	4.3.8
FSUBR32	Subtract Real Reversed, 32-bit	4.3.8
FSUBR64	Subtract Real Reversed, 64-bit	4.3.8
FSUBRP	Subtract Real Reversed and Pop	4.3.8
FTST	Test	4.3.8
FWAIT	CPU Wait	4.3.8
FXAM	Examine	4.3.8
FXCH	Exchange Registers	4.3.8
FXCHG	same as FXCH ST1	4.3.8

Table 4-1. (continued)

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

Table 4-1. (continued)

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

Table 4-1. (continued)

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

Table 4-1. (continued)

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

4.3 Instruction-definition Tables

4.3.1 Symbol Conventions

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

Table 4-2. Operand Type Symbols

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

Table 4-2. (continued)

Symbol	Operand Type
mem	any address expression with or without base- and/or index-addressing modes, such as the following: variable variable+3 variable[bx] variable[SI] variable[BX+SI] [BX] [BP+DI]
simpmem	any address expression without base- and index-addressing modes, such as the following: variable variable+4
mem|reg	any expression symbolized by reg or mem
mem|reg16	any expression symbolized by mem|reg, but must be 16 bits
label	any address expression that evaluates to a label
lab8	any label within +/- 128 bytes distance from the instruction

4.3.2 Flag Registers

The 8086 and 80286 CPUs have nine single-bit Flag registers that can be displayed to reflect the state of the processor. You cannot access these registers directly, but you can test them to determine the effects of an executed instruction upon an operand or register. The effects of instructions on Flag registers are also described in the instruction-definition tables, using the symbols shown in Table 4-3 to represent the nine Flag registers.

Table 4-3. Flag Register Symbols

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

4.3.3 8086 Data Transfer Instructions

There are four classes of data transfer operations:

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

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

Table 4-4. 8086 Data Transfer Instructions

Operation	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

Table 4-4. (continued)

Operation	Syntax	Result
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

Table 4-4. (continued)

Operation	Syntax	Result
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

Table 4-4. (continued)

Operation	Syntax	Result
XCHG	reg,mem|reg	exchange register and memory or register
XCHG	mem|reg,reg	exchange memory or register and register
XLAT	mem|reg	perform table lookup translation, table given by mem|reg, which is always BX. Replaces AL with AL offset from BX

4.3.4 8086 Arithmetic, Logical, and Shift Instructions

The 8086 and 80286 CPUs perform addition, subtraction, multiplication, and division in several ways. Both CPUs support 8- and 16-bit operations and also signed and unsigned arithmetic.

Six of the nine flag bits are set or cleared by most arithmetic operations to reflect the result of the operation. Table 4-5 summarizes the effects of arithmetic instructions on flag bits. Table 4-6 defines arithmetic instructions. Table 4-7 defines logical and shift instructions.

Table 4-5. Effects of Arithmetic Instructions on Flags

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

Table 4-6. 8086 Arithmetic Instructions

Instruction	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

Table 4-6. (continued)

Instruction	Syntax	Result
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

Table 4-2. (continued)

Instruction	Syntax	Result
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

Table 4-6. (continued)

Instruction	Syntax	Result
SBB	mem|reg,numb	subtract (with borrow) immediate data from memory or register
SUB	reg,mem|reg	subtract memory or register from register
SUB	mem|reg,reg	subtract register from memory or register
SUB	mem|reg,numb	subtract data constant from memory or register

Table 4-7. 8086 Logical and Shift Instructions

Instruction	Syntax	Result
AND	reg,mem|reg	perform bitwise logical AND of a register and memory or register
AND	mem|reg,reg	perform bitwise logical AND of memory or register and register
AND	mem|reg,numb	perform bitwise logical AND of memory or register and data constant

Table 4-7. (continued)

Instruction	Syntax	Result
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

Table 4-7. (continued)

Instruction	Syntax	Result
RCR	mem|reg,CL	rotate memory or register right through carry flag, number of bits given by CL register
ROL	mem|reg,1	rotate memory or register 1 bit left
ROL	mem|reg,CL	rotate memory or register left, number of bits given by CL register
ROR	mem|reg,1	rotate memory or register 1 bit right
ROR	mem|reg,CL	rotate memory or register right, number of bits given by CL register
SAL	mem|reg,1	shift memory or register 1 bit left, shift in low-order zero bit
SAL	mem|reg,CL	shift memory or register left, number of bits given by CL register, shift in low-order zero bits

Table 4-7. (continued)

Instruction	Syntax	Result
SAR	mem|reg,1	shift memory or register 1 bit right, shift in high-order bit equal to the original high-order bit
SAR	mem|reg,CL	shift memory or register right, number of bits given by CL register, shift in high-order bits equal to the original high-order bit
SHL	mem|reg,1	shift memory or register 1 bit left, shift in low-order zero bit. Note that SHL is a different mnemonic for SAL.
SHL	mem|reg,CL	shift memory or register left, number of bits given by CL register, shift in low-order zero bits. Note that SHL is a different mnemonic for SAL.
SHR	mem|reg,1	shift memory or register 1 bit right, shift in high-order zero bit

Table 4-7. (continued)

Instruction	Syntax	Result
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

Table 4-7. (continued)

Instruction	Syntax	Result
XOR	mem|reg,reg	perform bitwise logical exclusive OR of memory or register and register
XOR	mem|reg,numb	perform bitwise logical exclusive OR of memory or register and data constant

4.3.5 8086 String Instructions

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

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

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

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

Table 4-8. 8086 String Instructions

Instruction	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

Table 4-8. (continued)

Instruction	Syntax	Result
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	mem|reg	transfer a byte or word from accumulator to the destination operand

Table 4-8. (continued)

Instruction	Syntax	Result
STOSB		an alternate mnemonic for STOS that assumes a byte operand
STOSW		an alternate mnemonic for STOS that assumes a word operand

Table 4-9 defines prefixes for string instructions. A prefix repeats its string instruction the number of times contained in the CX register, which is decremented by 1 for each iteration. Prefix mnemonics precede the string instruction mnemonic in the statement line.

Table 4-9. 8086 Prefix Instructions

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

4.3.6 8086 Control Transfer Instructions

There are four classes of control transfer instructions:

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

All control transfer instructions cause program execution to continue at some new location in memory, possibly in a new code segment. The transfer can be absolute, or can depend upon a certain condition. Table 4-10 defines control transfer instructions. In the definitions of conditional jumps, above and below refer to the relationship between unsigned values. Greater than and less than refer to the relationship between signed values.

Table 4-10. 8086 Control Transfer Instructions

Instruction	Syntax	Result
CALL	label	push the offset address of the next instruction on the stack, jump to the target label
CALL	mem|reg16	push the offset address of the next instruction on the stack, jump to location indicated by contents of specified memory or register

Table 4-10. (continued)

Instruction	Syntax	Result
CALLF	label	push CS segment register on the stack, push the offset address of the next instruction on the stack (after CS), jump to the target label
CALLF	mem	push CS register on the stack, push the offset address of the next instruction on the stack, jump to location indicated by contents of specified double word in memory
INT	numb8	push the flag registers (as in PUSHF), clear TF and IF flags, transfer control with an indirect call through any one of the 256 interrupt-vector elements - uses three levels of stack

Table 4-10. (continued)

Instruction	Syntax	Result
INTO		if OF (the overflow flag) is set, push the flag registers (as in PUSHF), clear TF and IF flags, transfer control with an indirect call through interrupt-vector element 4 (location 10H). If the OF flag is cleared, no operation takes place
IRET		transfer control to the return address saved by a previous interrupt operation, restore saved flag registers, as well as CS and IP. Pops three levels of stack
JA	lab8	jump if "not below or equal" or "above" ((CF or ZF)=0)
JAE	lab8	jump if "not below" or "above or equal" (CF=0)
JB	lab8	jump if "below" or "not above or equal" (CF=1)

Table 4-10. (continued)

Instruction	Syntax	Result
JBE	lab8T	jump if "below or equal" or "not above" ((CF or ZF)=1)
JC	lab8	same as JB
JCXZ	lab8	jump to target label if CX register is zero
JE	lab8	jump if "equal" or "zero" (ZF=1)
JG	lab8	jump if "not less or equal" or "greater" (((SF xor OF) or ZF)=0)
JGE	lab8	jump if "not less" or "greater or equal" ((SF xor OF)=0)
JL	lab8	jump if "less" or "not greater or equal" ((SF xor OF)=1)
JLE	lab8	jump if "less or equal" or "not greater" (((SF xor OF) or ZF)=1)
JMP	label	jump to the target label
JMP	mem|reg16	jump to location indicated by contents of specified memory or register

Table 4-10. (continued)

Instruction	Syntax	Result
JMPF	label	jump to the target label possibly in another code segment
JMPS	lab8	jump to the target label within +/- 128 bytes from instruction
JNA	lab8	same as JBE
JNAE	lab8	same as JB
JNB	lab8	same as JAE
JNBE	lab8	same as JA
JNC	lab8	same as JNB
JNE	lab8	jump if "not equal" or "not zero" (ZF=0)
JNG	lab8	same as JLE
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)

Table 4-10. (continued)

Instruction	Syntax	Result
JNZ	lab8	same as JNE
JO	lab8	jump if "overflow" (OF=1)
JP	lab8	jump if "parity" or "parity even" (PF=1)
JPE	lab8	same as JP
JPO	lab8	same as JNP
JS	lab8	jump if "sign" (SF=1)
JZ	lab8	same as JE
LOOP	lab8	decrement CX register by one, jump to target label if CX is not zero
LOOPE	lab8	decrement CX register by one, jump to target label if CX is not zero and the ZF flag is set - "loop while zero" or "loop while equal"
LOOPNE	lab8	decrement CX register by one, jump to target label if CX is not zero and ZF flag is cleared - "loop while not zero" or "loop while not equal"
LOOPNZ	lab8	same as LOOPNE

Table 4-10. (continued)

Instruction	Syntax	Result
LOOPZ	lab8	same as LOOPE
RET		return to the address pushed by a previous CALL instruction, increment stack pointer by 2
RET	numb	return to the address pushed by a previous CALL, increment stack pointer by 2+numb
RETF		return to the address pushed by a previous CALLF instruction, increment stack pointer by 4
RETF	numb	return to the address pushed by a previous CALLF instruction, increment stack pointer by 4+numb

4.3.7 8086 Processor Control Instructions

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

Table 4-11. 8086 Processor Control Instructions

Instruction	Syntax	Result
CLC		clear CF flag
CLD		clear DF flag, causing string instructions to auto-increment the operand registers
CLI		clear IF flag, disabling maskable external interrupts
CMC		complement CF flag
ESC	numb8,mem|reg	do no operation other than compute the effective address and place it on the address bus (ESC is used by the 8087 numeric coprocessor) numb8 must be in the range 0 - 63
HLT		cause 8086 processor to enter halt state until an interrupt is recognized

Table 4-11. (continued)

Instruction	Syntax	Result
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

4.3.8 8087 Instruction Set

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

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

```
    fadd64  byte ptr  my_var
```

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

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

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

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

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

Table 4-12. 8087 Data Transfer Instructions

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

Table 4-12. (continued)

Syntax	Result
FXCHG	same as FXCH ST1
Integer Transfers	
FILD16	Integer Load (16-bit operands)
FILD32	Integer Load (32-bit operands)
FILD64	Integer Load (64-bit operands)
FIST16	Integer Store (16-bit operands)
FIST32	Integer Store (32-bit operands)
FIST16P	Integer Store and Pop (16-bit operands)
FIST32P	Integer Store and Pop (32-bit operands)
FIST64P	Integer Store and Pop (64-bit operands)
Packed Decimal Transfers	
FBLD	Packed Decimal (BCD) Load

Table 4-13. 8087 Arithmetic Instructions

Syntax	Operands	Result
	Additon	
FBSTP		Packed Decimal (BCD) Store 10 bytes and Pop
FADD		Add Real ST0 to ST1, store result in ST1 and Pop
FADD	STi,ST0	Add Real ST0 to STi, store result in STi
FADD32	mem	Add Real mem to ST0, store result in ST0 (32-bit operands)
FADD64	mem	Add Real mem to ST0, store result in ST0 (64-bit operands)
FADDP	STi,ST0	Add Real ST0 to STi, store result in STi and Pop
FIADD16	mem	Integer Add mem to ST0, store result in ST0 (16 bit-operands)
FIADD32	mem	Integer Add mem to ST0, store result in ST0 (32 bit-operands)

Table 4-13. (continued)

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

Table 4-13. (continued)

Syntax	Operands	Result
FSUBR		Subtract Real ST1 from ST0, store result in ST1 and Pop
FSUBR	STi,ST0	Subtract Real STi from ST0, store result in STi
FSUBR	ST0,STi	Subtract Real ST0 from STi, store result in ST0
FSUBR32	mem	Subtract Real mem from ST0, store result in ST0 (32-bit operands)
FSUBR64	mem	Subtract Real mem from ST0, store result in ST0 (64-bit operands)
FSUBRP	STi,ST0	Subtract Real STi from ST0, store result in STi and Pop
FISUBR16	mem	Integer Subtract ST0 from mem, store result in ST0 (16-bit operands)
FISUBR32	mem	Integer Subtract ST0 from mem, store result in ST0 (32-bit operands)

Table 4-13. (continued)

Syntax	Operands	Result
Multiplication		
FMUL		Multiply Real ST1 by ST0, store result in ST1 and Pop
FMUL	STi,ST0	Multiply Real STi by ST0, store result in STi
FMUL	ST0,STi	Multiply Real ST0 by STi, store result in ST0
FMUL32	mem	Multiply Real ST0 by mem, store result in ST0 (32-bit operands)
FMUL64	mem	Multiply Real ST0 by mem, store result in ST0 (64-bit operands)
FMULP	STi,ST0	Multiply Real STi by ST0, store result in STi and Pop
FIMUL16	mem	Integer Multiply ST0 by mem, store result in ST0 (16-bit operands)
FIMUL32	mem	Integer Multiply ST0 by mem, store result in ST0 (32-bit operands)

Table 4-13. (continued)

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

Table 4-13. (continued)

Syntax	Operands	Result
FDIVR		Divide Real ST0 by ST1, store result in ST1 and Pop
FDIVR	STi,ST0	Divide Real ST0 by STi, store result in STi
FDIVR	ST0,STi	Divide Real STi by ST0, store result in ST0
FDIVR32	mem	Divide Real mem by ST0, store result in ST0 (32-bit operands)
FDIVR64	mem	Divide Real mem by ST0, store result in ST0 (64-bit operands)
FDIVRP	STi,ST0	Divide Real STi by ST0, store result in STi and Pop
FIDIVR16	mem	Integer Divide mem by ST0, store result in ST0 (16-bit operands)
FIDIVR32	mem	Integer Divide mem by ST0, store result in ST0 (32-bit operands)

Table 4-13. (continued)

Syntax	Operands	Result
Other Operations		
FSQRT		Square Root
FSCALE		Interpret ST1 as an integer and add to exponent of ST0
FPREM		Partial Remainder
FRNDINT		Round to Integer
FXTRACT		Extract Exponent and Significand
FABS		Absolute Value
FCHS		Change Sign
FCOM32	mem	Compare Real mem and ST0 (32-bit operands)
FCOM64	mem	Compare Real mem and ST0 (64-bit operands)

Table 4-13. (continued)

Syntax	Operands	Result
FCOMP		Compare Real ST0 and ST1 and Pop
FCOM32P	mem	Compare Real mem and ST0 and Pop (32-bit operands)
FCOM64P	mem	Compare Real mem and ST0 and Pop (64-bit operands)
FCOMPP		Compare Real ST0 and ST1, then Pop ST0 and ST1
FICOM16	mem	Integer Compare mem and ST0 (16-bit operands)
FICOM32	mem	Integer Compare mem and ST0 (32-bit operands)
FICOM16P	mem	Integer Compare mem and ST0 and Pop (16-bit operands)

Table 4-13. (continued)

Syntax	Operands	Result
FICOM32P	mem	Integer Compare mem and ST0 and Pop (32-bit operands)
FTST		Test ST0 by comparing it to zero
FXAM		Report ST0 as either positive or negative

Table 4-14. 8087 Transcendental Instructions

Syntax	Result
FPTAN	Partial Tangent
FPATAN	Partial Arctangent
F2XM1	2^x-1
FYL2X	$Y * \log_2 X$
FYL2XP1	$Y * \log_2(X + 1)$
FLDZ	Load + 0.0
FLD1	Load + 1.0
FLDPI	Load 80-bit value for pi.
FLDL2T	Load $\log_2 10$

Table 4-15. 8087 Constant Instructions

Syntax	Result
FLDL2E	Load $\log_2 e$
FLDLG2	Load $\log_{10} 2$
FLDLN2	Load $\log_e 2$

Table 4-16. 8087 Processor Control Instructions

Syntax	Operands	Result
FINIT/FNINIT		Initialize Processor
FDISI/FNDISI		Disable Interrupts
FENI/FNENI		Enable Interrupts
FLDCW	mem	Load Control Word
FSTCW/FNSTCW	mem	Store Control Word
FSTSW/FNSTSW	mem	Store Status Word
FCLEX/FNCLEX		Clear Exceptions
FSTENV/FNSTENV	mem	Store Environment
FLDENV	mem	Load Environment
FSAVE/FNSAVE		Save State
FRSTOR		Restore State
FINCSTP		Increment Stack Pointer

Table 4-16. (continued)

Syntax	Operands	Result
FDECSTP		Decrement Stack Pointer
FFREE		Free Register
FNOP		No Operation
FWAIT		CPU Wait

4.3.9 Additional 186 and 286 Instructions

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

Table 4-17. Additional 186 and 286 Instructions

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

4.3.10 Additional 286 Instructions

The following instructions are specific to the 80286 processor.

Table 4-18. Additional 286 Instructions

Syntax	Result
CTS	Clear Task Switched Flag
ARPL	Adjust Priviledge level
LGDT	Load Global Descriptor Table Register
SGDT	Store Global Descriptor Table Register
LIDT	Load Interrupt Descriptor Table Register
SIDT	Store Interrupt Descriptor Table Register
LLDT	Load Local Descriptor Table Register from Register/Memory
SLDT	Store Local Descriptor Table Register to Register/Memory
LTR	Load Task Register from Register/Memory
STR	Store Task Register to Register/Memory
LMSW	Load Machine Status Word from Register/Memory
SMSW	Store Machine Status Word
LAR	Load Access Rights from Register/Memory
LSL	Load Segment Limit from Register/Memory

Table 4-18. (continued)

Syntax	Result
ARPL	Adjust Required Privilege Level from Register/Memory
VERR	Verify Read Access; Register/Memory
VERW	Verify Write Access

End of Section 4

RASM-86 Code-macro Facilities

5.1 Introduction

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

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

5.2 Invoking Code-macros

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

```
    MYCODE  PARM1,PARM2
```

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

5.3 Defining Code-macros

A code-macro definition takes the general form:

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

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

5.3.1 Formal Parameter List

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

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

formal_name

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

specifier_letter

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

Table 5-1. Code-macro Operand Specifiers

Letter	Operand Type
A	Accumulator register, AX or AL.
C	Code, a label expression only.
D	Data, a number used as an immediate value.
E	Effective address, either an M (memory address) or an R (register).
M	Memory address. This can be either a variable or a bracketed register expression.
R	General register only.
S	Segment register only.
X	Direct memory reference.

modifier_letter

The optional modifier_letter in a code-macro definition is a further requirement on the operand. The meaning of the modifier letter depends on the type of the operand. For variables, the modifier requires the operand be a certain type:

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

For numbers, the modifiers require the number be a certain size: b for -256 to 255 and w for other numbers. Table 5-2 summarizes code-macro modifiers.

Table 5-2. **Code-macro Operand Modifiers**

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

range

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

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

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

5.3.2 Code-macro Directives

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

The following are legal code-macro directives:

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

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

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 E.

SEGFIX

SEGFIX instructs RASM-86 to determine whether a segment-override prefix byte is needed to access a given memory location. If so, it is output as the first byte of the instruction. If not, RASM-86 takes no action. SEGFIX has the following form:

```
SEGFIX formal_name
```

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

NOSEGFIX

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

```
NOSEGFIX   segreg, form_name
```

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

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

MODRM

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

The MODRM byte carries the information in three fields:

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

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

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

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

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

MODRM has the forms:

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

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

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

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

RELB and RELW

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

```
RELB form_name
RELW form_name
```

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

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

DB, DW and DD

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

The directives have the following forms:

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

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

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

DBIT

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

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

The field_description has two forms:

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

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

The second form shown contains form_name, a formal parameter name that instructs the assembler to put a certain number in the specified position. This number normally refers to the register specified in the first line of the code-macro. The numbers used in this special case for each register are the following:

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

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

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

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

```
    DEC    DX
```

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

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

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

IF, ELSE, and ENDIF

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

5.3.3 Example Code-Macro Definitions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

End of Section 5

XREF-86 Cross-Reference Utility

6.1 Introduction

XREF-86 is an assembly language cross-reference utility program that creates a cross-reference file showing the use of symbols throughout the program. XREF-86 accepts two input files created by RASM-86. XREF-86 assumes these input files have filetypes of LST and SYM respectively, and they both reside on the same disk drive. XREF-86 creates one output file with the filetype XRF. Figure 6-1 illustrates XREF-86 operation.

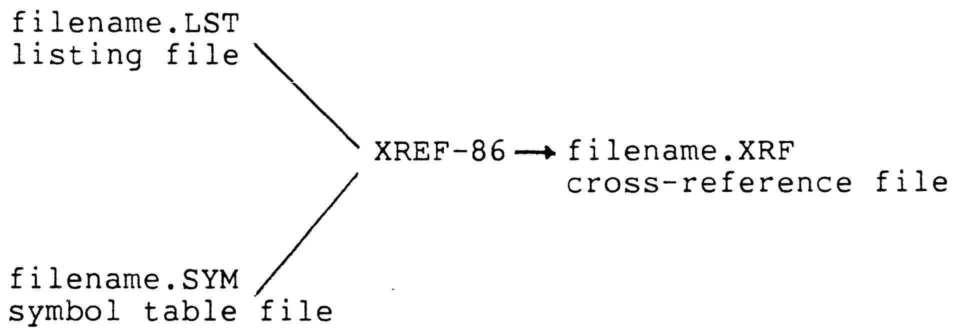


Figure 6-1. XREF-86 Operation

6.2 XREF-86 Command Syntax

XREF-86 is invoked using the command form:

XREF86 [drive:] filename

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

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

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

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

```
xref86 c:bios
```

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

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

End of Section 6

LINK 86 Linkage Editor

7.1 Introduction

LINK 86, Version 1.5, is the Digital Research linkage editor that combines relocatable object files into a command file that runs under any Digital Research 8086-based operating system. As used here, "8086-based" refers to Intel 8088, 8086, 80186, and 80286 microprocessors. The object files can be produced by Digital Research's 8086 and 80286 language translators or other translators producing object files using a compatible subset of the Intel 8086/80286 object module format.

7.2 LINK 86 Operation

LINK 86 accepts three types of files.

Object (OBJ) File	This is a language source file processed by the language translator into the relocatable object code used by the microprocessor. This type of file contains one or more object modules.
Library (L86) File	The library file is an indexed library of commonly used object modules. A library file is generated by the library manager, LIB-86, in the processor's relocatable object format.
Input (INP) File	The input file consists of filenames and options like a command line entered from the console. For a detailed explanation of the input file, see Section 7.12.

LINK 86 produces the following types of files:

Command (CMD or 286) File	Contains executable code that the operating system can load.[1] The filetype LINK 86 gives to the command file is either CMD or 286, depending on the filetype specified in the LIBATTR module of your compiler's runtime library. The default filetype for LINK 86 is 286. RASM-86 does not use a runtime library. Therefore, when linking RASM code, you must specify on the LINK 86 command line which filetype you want your command file to have. To keep the discussions in this guide simple, all of the command files have been given a CMD filetype.
Symbol Table (SYM) File	Contains a list of symbols from the object files and their offsets. This file is suitable for use with Digital Research's SID-286 symbolic instruction debuggers.
Line Number (LIN) File	Contains line number symbols, which can be used by SID-286 for debugging. This file is created only if the compiler puts line number information into the object files being linked.
Map (MAP) File	Contains segment information about the layout of the command file.

During processing, LINK 86 displays any unresolved symbols at the console. Unresolved symbols are symbols referenced but not defined in the files being linked. Unless you are linking overlays, such symbols must be resolved before the program can run properly. Overlays are described in detail in Section 8.

[1]A command file with type 286 runs in the native FlexOS environment.

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

Figure 7-1 illustrates LINK 86 operation.

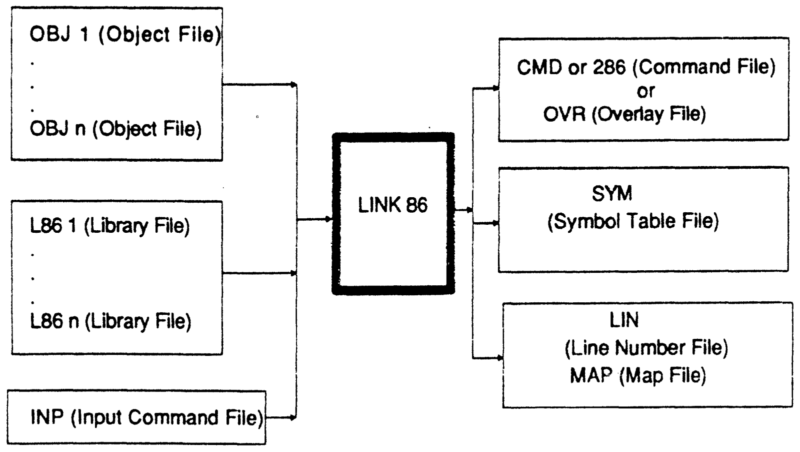


Figure 7-1. LINK 86 Operation

See Section 7.15 for a complete explanation of the link process.

7.3 LINK 86 Command Syntax

You invoke LINK 86 with a command of the form:

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

where **filespec** is a CP/M-type file specification, consisting of an optional drive specification and a filename with optional filetype. Each **filespec** can be followed by one or more of the command options described in Section 7.6. If you enter a filename to the left of the equal sign, LINK 86 creates the output files with that name and the appropriate filetypes. For example, if the files PARTA, PARTB, and PARTC are written in 8086 or 80286 assembly code, the command

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

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

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

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

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

You can also instruct LINK 86 to read its command line from a file, thus making it possible to store long or commonly used link commands on disk (see Section 7.12).

The following are examples of LINK 86 commands:

```
A>link86 myfile = parta,partb

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

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

The available LINK 86 command options are described in Section 7.6.

7.4 Stopping LINK 86

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

7.5 Shareable Runtime Libraries

LINK 86, version 1.5, supports shareable runtime libraries. Shareable runtime libraries, which are referred to as SRTLs in the remainder of this guide, allow multiple users to share a single copy of library code at runtime. This makes it unnecessary for users to store library code in their command files. When libraries are shared, only references to the library code are linked with the user's object files.

No extra steps are required when linking object files with shareable runtime library (SRTL) files. There are two methods for linking with a SRTL:

Method 1: Compiler-requested Libraries

If the SRTL is a compiler-requested library, you do not have to specify the library name in the link command. In the following example, assume the compiler used to compile the program HELLO requests the library XYZ.

```
A>link86 hello
```

If the library XYZ is a SRTL, LINK 86 treats it as such automatically.

Method 2: Explicitly Requested Libraries

You can specify a shareable library file in the command in the same manner you would specify a normal library file. For example, if the library SUPRUTIL.L86 is a SRTL, the following LINK 86 command links it with the object files: SUPRPROG, INIT, and TERM.

```
A>link86 suprprog,init,term,suprutil.l86
```

If you are using a large model SRTL, you must also specify a transfer vector file in your command line. Transfer vectors are described in Appendix B. For example, if SUPRUTRIL.L86 is a large model SRTL using the transfer vector, TRANSV.OBJ, enter

```
A>link86 suprprog,init,term,transv,suprutil.186
```

See the SHARED and NOSHARED options in Section 7.11.2 for more information on linking with shareable runtime libraries.

7.6 LINK 86 Command Options

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

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

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

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

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

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

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

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

LINK 86 command options are grouped into the following categories:

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

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

Table 7-1. LINK 86 Command Options

Option	Abbreviation	Meaning
CODE	C	controls contents of CODE section of command file
DATA	D	controls contents of DATA section of command file
EXTRA	E	controls contents of EXTRA section of command file

Table 7-1. (continued)

Option	Abbreviation	Meaning
STACK	ST	controls contents of STACK section of command file
X1	X1	controls contents of X1 section of command file
X2	X2	controls contents of X2 section of command file
X3	X3	controls contents of X3 section of command file
X4	X4	controls contents of X4 section of command file
FILL	F	zero fill and include uninitialized data in command file
NOFILL	NOF	do not include uninitialized data in command file
HARD8087	HA	create a command file requiring an 8087 coprocessor.
SIM8087	SI	create a command file using 8087 software emulation routines.

Table 7-1. (continued)

Option	Abbreviation	Meaning
AUTO8087	AU	create a command file that can decide at run-time to use an 8087 coprocessor if it is available.
CODESHARED	CODES	mark group as shared in the 286 file header
LIBSYMS	LI	include symbols from library files in SYM file
NOLIBSYMS	NOLI	do not include symbols from library files in SYM file
LOCALS	LO	include local symbols in SYM file
NOLOCALS	NOLO	do not include local symbols in SYM file
LINES	LIN	create LIN file with line number symbols
NOLINES	NOLIN	do not create LIN file
MAP	M	create a MAP file
SEARCH	S	search library and only link referenced modules

Table 7-1. (continued)

Option	Abbreviation	Meaning
SHARED	SH	force a shareable runtime library (SRTL) to be treated as shared.
NOSHARED	NOSH	force a shareable runtime library (SRTL) to be treated as a normal unshared library.
INPUT	I	read command line from disk file
ECHO	ECHO	echo contents of INP file on console
CUMULATIVE	CUM	do not overlay data when creating overlay file
NOCUMULATIVE	NOCUM	overlay both code and data when loading overlay file

7.7 Command File Options

The options described in this section affect the contents of the command file created by LINK 86.

Most command file options can appear after any filename in the command line. The only exceptions are the HARD8087, SIM8087, and AUTO8087 options which, if they appear, must appear after the first filename.

7.7.1 CODE / DATA / STACK / EXTRA / X1 / X2 / X3 / X4

A command file consists of a 128-byte header record followed by up to eight sections, each of which can be up to 1 megabyte in length. These sections are called CODE, DATA, STACK, EXTRA, X1, X2, X3, and X4. Each of these sections correspond to a LINK 86 command option of the same name. The header contains information such as the length of each section of the command file, its minimum and maximum memory requirements, and its load address. This information is used by the operating system to properly load the file (see the system guide for your operating system).

These options allow you to identify a section in a command file. The parameters described below allow you to alter the information in that section.

File Section Option Parameters

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

LINK 86 option parameters are specified using the form:

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

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

Table 7-2. Command File Option Parameters

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

GROUP, CLASS, SEGMENT

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

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

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

ABSOLUTE, ADDITIONAL, MAXIMUM

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

Each parameter is a hexadecimal number enclosed in square brackets.

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

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

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

For example, the command

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

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

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

ORIGIN

The ORIGIN parameter is a hexadecimal value that indicates the byte offset where the indicated section of the command file should begin. LINK 86 assumes a default ORIGIN value of 0 for each section except the DATA section, which has a default value of 100H to reserve space for the Base Page (see the system guide for your operating system).

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

Table 7-3. Default Values for Command File Options and Parameters

OPTION	GROUP	CLASS	SEGMENT	ABSLT	ADDT'L	MAX	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.

7.7.2 FILL / NOFILL

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

7.7.3 HARD8087 / SIM8087 / AUTO8087

The options described in this section are used with programs containing 8087 opcodes.

You can use the HARD8087 option if the program will always run on a system with an 8087. This option tells LINK 86 to use the 8087 opcodes generated by the compiler, and not to replace them with 8087 software emulation routines. Using this option saves about 16K bytes of space that would be used by the emulation routines.

You can use the SIM8087 option if the program will always run on a system without an 8087. This option tells LINK 86 to replace the 8087 opcodes in the OBJ file with interrupts that vector into the 8087 emulation routines.

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

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

FILE Parameter

You can specify that the 8087 emulation routines be loaded into your command file only at runtime by specifying the FILE parameter with the SIM8087 or AUTO8087 option.

For example, if the program contained in the file: PROG1 has interrupts into 8087 emulation routines and you do not want to include those routines in the command file, you could enter the command:

```
A>link86 progl [sim8087[file]]
```

This command links PROG1 without the 8087 emulation routines. When the command to execute PROG1 is given, the 8087 emulation routines are loaded into PROG1, prior to execution.

7.7.4 CODESHARED

The CODESHARED option marks the group in the 286 file header with a group descriptor type 09h (shared code). The default code group descriptor is 01h (non-shared code).

7.8 SYM File Options

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

- LOCALS
- NOLOCALS
- LIBSYMS
- NOLIBSYMS

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

7.8.1 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 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 / NOLIBSYMS

The LIBSYMS option directs LINK 86 to include in the SYM file any symbols coming from a library searched during the link operation. The NOLIBSYMS option directs LINK 86 not to include those symbols in the SYM file. Typically, such a library search involves the runtime subroutine library of a high-level language such as C. Because the symbols in such a library are usually of no interest to the programmer, the default is NOLIBSYMS.

7.9 LIN File Options

Some compilers provide an option allowing you to put line numbers into object files. If line numbers are present in the object file, LINK 86 creates a file containing line numbers and their offsets, which can be used when debugging with SID-286. The following options specify whether or not a LIN file is created by LINK 86:

```
LINES
NOLINES
```

The LINES option, which is active by default, directs LINK 86 to create a LIN file, if possible. If no line information is present in the object file, then LINK 86 does not create the LIN file. The NOLINES option directs LINK 86 not to create a LIN file, even if line numbers are present in the object file.

7.10 MAP File Option

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

OBJMAP	NOOBJMAP
L86MAP	NOL86MAP
ALL	NOCOMMON

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

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

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

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

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

7.11 L86 File Options

The following command options determine how the library files are used by LINK 86:

SEARCH
SHARED
NOSHARED

7.11.1 SEARCH

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

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

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

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

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

7.11.2 SHARED / NOSHARED

The SHARED and NOSHARED options determine whether or not a library file is to be used as a shareable runtime library (SRTL). When a runtime library is NOSHARED, both the code and the data from that library are linked with the object files. When a runtime library is SHARED, only the data from that library is linked with the object files and a single copy of the library code resides in a special command file called an Executable Shared Runtime Library (XSRTL). The code stored in an XSRTL file can be accessed by any executable file linked as a user of the SRTL.

When a SRTL is created, it is given an attribute that determines if the library is to be treated as SHARED or NOSHARED. (See Appendix B for a description on how to create and modify SRTLs.)

If a SRTL has a default attribute of NOSHARED, LINK 86 treats it as an ordinary library file. You can force LINK 86 to treat the SRTL as shareable by specifying the SHARED option after the SRTL name. For example:

```
A>link86 myprog=main,part1,part2,util.l86 [shared]
```

creates a file called MYPROG.CMD that uses the runtime library UTIL.L86 as a SRTL.

Unlike other options, the SHARED option will not remain set until explicitly reset.

If a SRTL has a default attribute of SHARED, you can force LINK 86 to treat it as a normal library by specifying the NOSHARED option. This forces the referenced SRTL routines to be resident in the user's code file and the loader doesn't need to perform any load-time resolution of external references.

As an example of the NOSHARED option, the command:

```
A>link86 myprog=main,part1,part2,util.l86 [noshared,search]
```

causes LINK 86 to treat the shareable runtime library UTIL.L86 as a normal library.

When using the NOSHARED option, you should also specify the SEARCH option. By using the SEARCH option, LINK 86 includes only referenced segments, and does not include the library attribute segment (LIBATTR) that defines the library as shareable. If SEARCH is not specified, LINK 86 includes every module and segment in the library, whether it is used or not.

7.12 Input File Options

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

```
INPUT
ECHO
```

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

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

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

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

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

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

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

7.13 I/O Option

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

```
$Tdrive
```

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

File Types

LINK 86 recognizes five file types:

C - Command File (CMD, 286, or OVR)
L - Library File (L86)
M - Map File (MAP)
O - Object File (OBJ or L86)
S - Symbol File (SYM and LIN)

Drive Specifications

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

X - Console
Y - Printer
Z - No Output

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

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

A>link86 part1[$szodlb],part2

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

The value of a $ option remains in effect until LINK 86 encounters a countermanding option as it processes the command line from left to right. This is useful when linking overlays, in that you do not have to specify the drive for each overlay file. See Section 8 for more information on overlays.

For example, the command

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

suppresses the SYM files generated when the overlay files: OV1, OV2 and OV3 are linked. When LINK 86 links OV4, it places the SYM and LIN files on drive A.

7.13.1 $C (Command) Option

The $C option uses the form:

```
$Cdrive
```

LINK 86 normally generates the command file on the same drive as the first object file in the command line. The $C option instructs LINK 86 to place the command file on the drive specified by the **drive** character following the $C ($CZ suppresses the generation of a command file). This option also applies to OVR files if you are using LINK 86 to create overlays. Refer to Section 8 for more information on overlays.

7.13.2 $L (Library) Option

The $L option uses the form:

```
$Ldrive
```

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

7.13.3 $M (Map) Option

The $M option uses the form:

```
$Mdrive
```

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

7.13.4 $O (Object) Option

The $O option uses the form:

```
$Odrive
```

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

```
A>link86 p[$od],q,r,s,t,u.186,b:v
```

tells LINK 86 that all the object files except the last one are located on drive D. Note that this does not apply to libraries linked automatically (see Section 7.13.2).

7.13.5 $S (Symbol and Line Number) Option

The $S option uses the form:

```
$Sdrive
```

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

7.14 Overlay Options

The following command options determine how the overlay manager arranges data when creating an overlay file:

CUMULATIVE
NOCUMULATIVE

The CUMULATIVE option tells the overlay manager not to overlay data when loading an overlay file. In this case, the data becomes cumulative. This approach uses more memory, but is generally a safer programming practice. The CUMULATIVE option is active by default.

The NOCUMULATIVE option tells the overlay manager to overlay both code and data when loading an overlay file. With this option, the data in successive overlays is not cumulative, but overwrites existing data. Overlays on the same level share data areas. This approach uses less memory, but risks one overlay destroying the data required by another overlay.

The CUMULATIVE and NOCUMULATIVE options can be specified only for the root file using the overlay files. For example, if you have a file named MYFILE and you want to overlay the files OVER1, OVER2, and OVER3 using the NOCUMULATIVE option, you enter:

```
A>link86 myfile [NOCUM] (over1) (over2) (over3)
```

Refer to Section 8 for more information on overlays.

7.15 The Link Process

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

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

Segment	A 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 command 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 the type of boundary the segment is to begin. The Align types are byte, word, paragraph and page. LINK 86 uses the align type when it combines parts of segments from separate files into one segment. The align type is also used when LINK 86 combines segments into groups, sections, or segments of the command file.
Combine type	The Combine type determines how LINK 86 combines segments with the same name from different files into a single segment. The Combine types are: public, common, stack, absolute, and local.

Group	A Group is a collection of segments with different names grouped into a single segment. By grouping segments, you can combine library modules and other modules of similar type with your object file modules into a single segment. By combining the contents of individual segments into one large segment, the pointer need only be a 16-bit offset into a single segment. This technique results in shorter and faster code than addressing individual segments with 32-bit pointers.

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

7.15.1 Phase 1 - Collection

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

Combine Types

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

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

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

For example, suppose there are three object files: FILEA.OBJ, FILEB.OBJ, and FILEC.OBJ, and each file defines a data segment, named Dataseg, with the public combine type. Figure 7-2 illustrates how LINK 86 combines this segment using the default combine type, public.

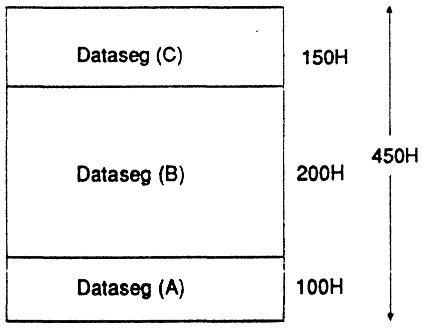


Figure 7-2. Combining Segments with the Public Combine Type

Figure 7-3 illustrates the Common Combine type. Suppose the three files: FILEA.OBJ, FILEB.OBJ, and FILEC.OBJ each contain a data segment, named Dataseg, with the Common Combine type. LINK 86 combines these data segments so all parts of the segments from the separate files being linked have the same low address in memory. The Common Combine type overlays the data or code from the various object files, making it common to all of the linked routines in the executable file. Note that this corresponds to a common block in high-level languages.

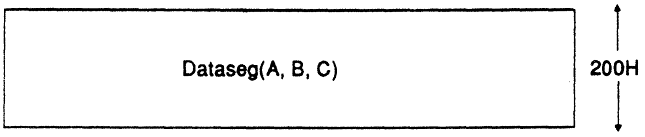


Figure 7-3. Combining Segments with the Common Combine Type

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

For example, suppose the three files FILEA.OBJ, FILEB.OBJ, and FILEC.OBJ each contain a segment named Stkseg with the Stack Combine type. Figure 7-4 illustrates how they are combined by LINK 86.

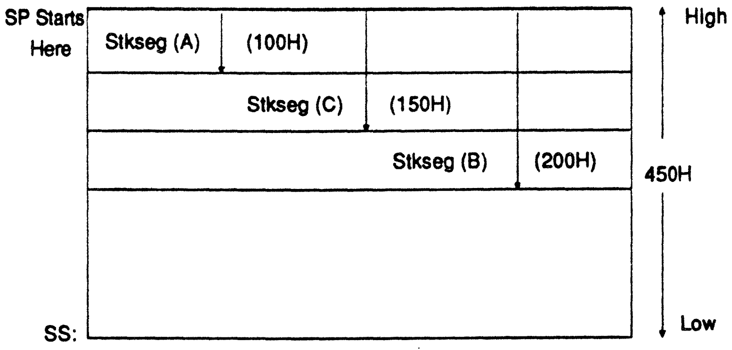


Figure 7-4. Combining Segments with Stack Combination

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

Align Type

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

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

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

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

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

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

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

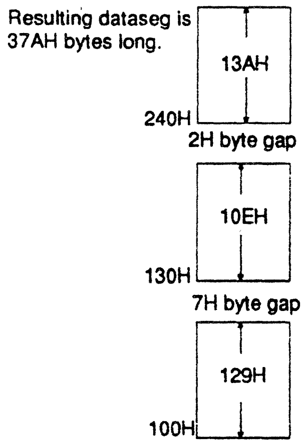
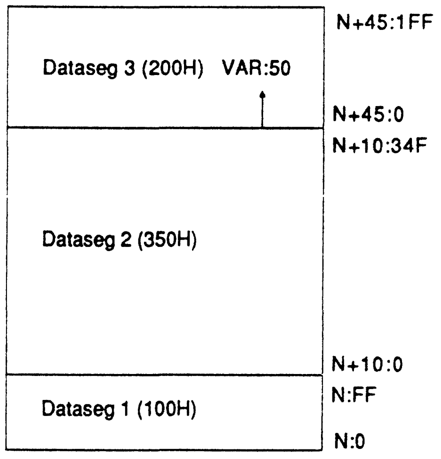


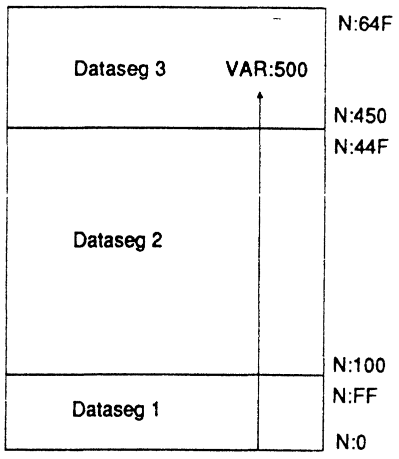
Figure 7-5. Paragraph Alignment

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

Grouping

LINK 86 combines segments into groups. When segments are grouped, intersegment gaps are determined using the same Align types as those used to combine segments. Figure 7-6 illustrates how LINK 86 combines segments into groups.





7-6a. Segments Without Groups 7-6b. Segments Within A Group

Figure 7-6. The Effect of Grouping Segments

In Figure 7-6, N:0 is the base address where the segments are loaded at run-time (paragraph N, offset 0). Figure 7-6a shows that each segment not contained in a group begins at offset zero, and thus can be up to 64K long. The offset of any given location, in this case the variable VAR, is relative to the base of the segment. Thus, in order to access VAR at run-time, the program must load a segment register with the base address of the data segment Dataseg3 and point to an offset of 50H.

In Figure 7-6b, the same segments are combined in a group. The offsets of the segments are now cumulative and thus cannot extend past 64K bytes (FFFFH). The offset of VAR is 500H relative to the base of the group. At run-time, the program does not need to reload a segment register to point to the base of Dataseg3, but can access VAR directly by pointing to an offset of 500H.

7.15.2 Phase 2 - Create Command File

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

1. LINK 86 first processes any segments, groups or classes the user placed in a specific section by means of the command line options described in Section 7.6.
2. Segments belonging to the group CGROUP are placed in the CODE section of the command file.
3. Segments belonging to the group DGROUP are placed in the DATA section of the command file. Note that the group names CGROUP and DGROUP are automatically generated by PL/I-86™, CB86™, and other high-level language compilers.
4. If there are any segments not processed according to (1), (2), and (3), LINK 86 places them in the command file according to their class name, as shown in Table 7-4. This table also shows the RASM-86 segment directives that produce the class names as defaults.
5. Segments not processed by any of the above means are omitted from the command file because LINK 86 does not have sufficient information to position them.

Table 7-4. LINK 86 Usage of Class Names

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

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

See Appendix H for a list of LINK 86 error messages.

End of Section 7

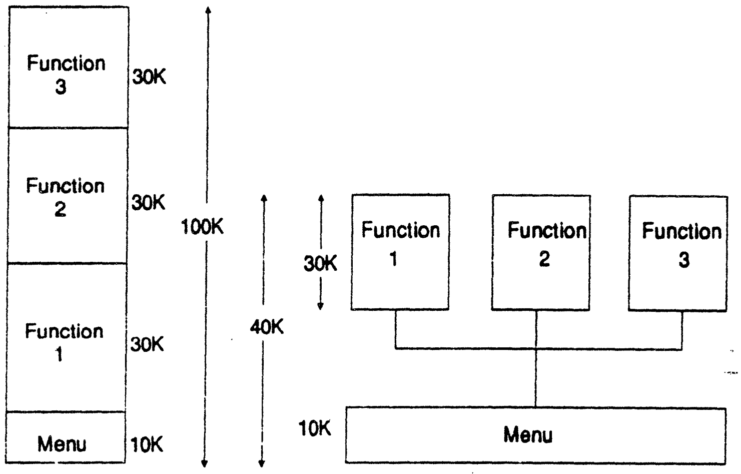
Overlays

8.1 Introduction

This section describes how LINK 86 creates programs with separate files called overlays. Each overlay file is a separate program module loaded into memory when needed by the program. By loading only program modules needed at a particular time, the amount of memory used by the program is minimized.

As an example, many application programs are menu-driven, with the user selecting one of a number of functions to perform. Because the program's modules are separate and invoked sequentially, there is no reason for them to reside in memory simultaneously. Using overlays, each function on the menu can be a separate subprogram stored on disk and loaded only when required. When one function is complete, control returns to the menu portion of the program, from which the user selects the next function.

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 locations in memory.



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 of five. Section 8.2 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 much less memory than would be required if all the functions and subfunctions had to reside in memory simultaneously.

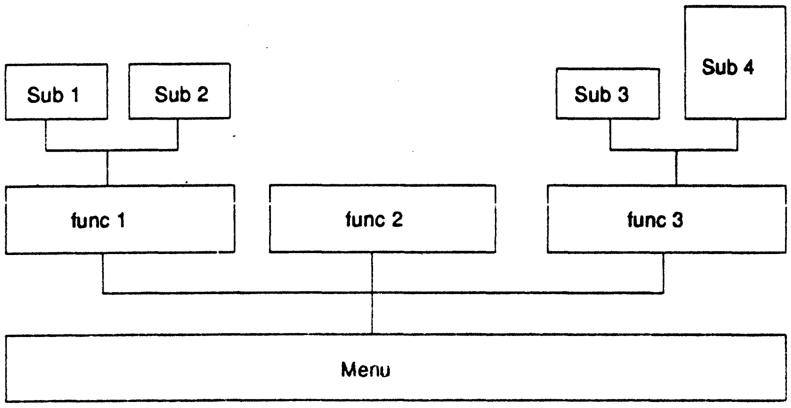


Figure 8-2. Tree Structure of Overlays

8.2 Overlay Syntax

An overlay manager is provided with the runtime library in all of Digital Research's high-level language compilers. You specify overlays in the LINK 86 command line by enclosing each overlay specification in parentheses.

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

```
A>link86 root [option] (overlay1)
A>link86 root [option] (overlay1,part2,part3)
A>link86 root [option] (overlay1=part1,part2,part3)
```

where ROOT is the object file that calls the overlay(s) and [OPTION] may include either the CUMULATIVE or NOCUMULATIVE option described in Section 7.14.

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

In order to use overlays with RASM-86 code, you must have a high-level language runtime library on your disk and it must be specified on the LINK 86 command line. For example, if you are using the DR C runtime library, you would specify either the CLEARS.L86 or CLEARL.L86 library on your command line, depending on whether you wish to link using the small or large memory model. For example, to link a small memory model program using the RASM-86 file ROOT.OBJ and the overlay files PART1.OBJ and PART2.OBJ (which are also RASM-86 files), you would enter the command:

```
A>link86 root,clears.l86 (part1, part2)
```

On the command line, a left parenthesis indicates the start of a new overlay specification and the end of the preceding overlay specification. You can use spaces to improve readability, and commas can separate parts of a single overlay. 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
```

The correct command is:

```
A>link86 root,moreroot(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(func1(sub1)(sub2))(func2)(func3(sub3)(sub4))
```

When linking files to be overlayed along with files not to be overlayed, the files to be overlayed are specified last. For example, if you want to create the file ROOTFILE.CMD from the files: ROOTFILE.OBJ, PARTA.OBJ, and PARTB.OBJ and you want to link OVER1.OBJ and OVER2.OBJ as overlays, you enter the command:

```
A>link86 rootfile, parta, partb (over1, over2)
```

8.3 Writing Programs That Use Overlays

There are two methods for writing programs that use overlays. The first method involves no special coding, but has two restrictions. The first is that all overlays must be on the default drive. The second restriction is that the overlay file 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.

8.3.1 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 is any program terminated with a RET instruction.

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

```
;root file
         cseg
         extrn    overlay1:near
root:    mov      bx,offset root_message
         mov      cl,9              ;print string function #
         int      224               ;print the string
         mov      bx,offset ovlay_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 defined as follows:

```
;overlay file
           cseg
overlay1:  mov      cl ,9
           int      224         ;print string 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 transfers 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 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:

- 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.
- 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.
- 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.
- The names of the overlays are fixed. To change the names of the overlays, you must edit, reassemble, and relink the program.
- No nonstandard statements are needed. Thus, you can postpone the decision on whether or not to create overlays until link-time.

8.3.2 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:

```
        cseg
        extrn   ?ovlay:near      ;entry point of overlay manager
root:   mov     bx,offset root_message
        mov     cl,9             ;print string function #
        int     224              ;print the string
        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
        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 from the default drive, (the current value of the variable overlay_name) and transfers control to it. When OVERLAY1 finishes processing, control returns to the statement following the call.

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:

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

8.4 General Overlay Constraints

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

- Each overlay has only one entry point. The Overlay Manager assumes that this entry point is at the load address of the overlay.
- You cannot reference overlay routines from the root module or from overlays "lower" on the tree than the overlay being referenced. For example, based on the overlay structure illustrated in Figure 8-2, you cannot reference an arbitrary routine in any of the overlay modules from MENU. The only possible "upward" reference to an overlay from MENU is through that overlay's main entry point. You can, however, make "downward" references to any routine contained in overlays lower on the tree or in the root module.

- Common segments declared in one module cannot be initialized by a "higher" module. LINK 86 ignores any attempt to do so.
- You can nest overlays to a maximum depth of 5 levels.

End of Section 8

LIB-86 Library Utility

9.1 Introduction

LIB-86 is a utility program for creating and maintaining library files containing 8086 or 80286 object modules. These modules can be produced by Digital Research's 8086 language translators such as RASM-86, DR C, and CB86, or by any other translators that produce modules in Intel's 8086 or 80286 object module format.

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

9.2 LIB-86 Operation

When you invoke LIB-86, it reads the indicated files and produces a Library file, a Cross-reference file, or a Module map file as indicated by the command line. When LIB-86 finishes processing, it displays the Use Factor, a decimal number indicating the percent of available memory LIB-86 uses during processing. Figure 9-1 shows the operation of LIB-86.

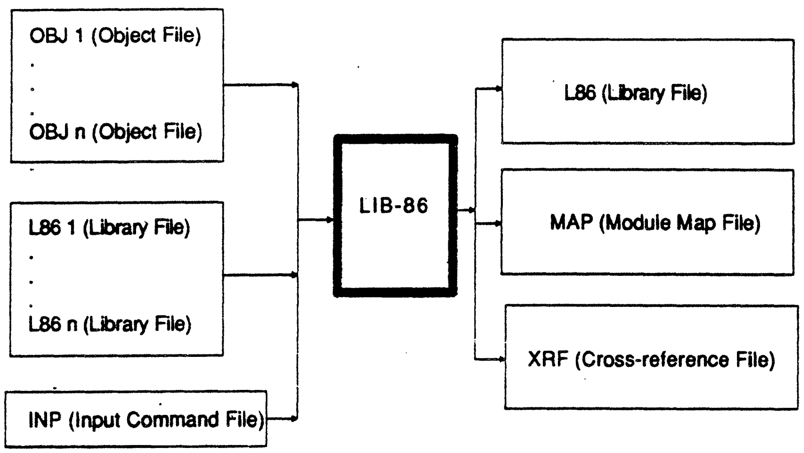


Figure 9-1. LIB-86 Operation

Table 9-1 shows the filetypes recognized by LIB-86.

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

9.3 LIB-86 Command Syntax

LIB-86 uses the command form:

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

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

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

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

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

9.4 Stopping LIB-86

You can press any console key to halt LIB-86, which then displays the message:

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

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

9.5 LIB-86 Command Options

When you invoke LIB-86, you can specify optional parameters in the command line controlling 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	Abbreviation
DELETE	Delete a Module from a Library file	D
EXTERNALS	Show EXTERNALS in a Library file	E
ECHO	Echo contents of INP file on console	
INPUT	Read commands from Input file	I
MAP	Create a Module Map	MA
MODULES	Show Modules in a Library file	MO
NOALPHA	Show Modules in order of occurrence	N
PUBLICS	Show PUBLICS in a Library file	P
REPLACE	Replace a Module in a Library file	R
SEGMENTS	Show Segments in a Module	SEG
SELECT	Select a Module from a Library file	SEL
XREF	Create a Cross-reference file	X

9.6 Creating and Updating Libraries

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

9.6.1 Creating a New Library

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

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

A>lib86 newlib.186 = a.obj,b.obj,c.obj

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

The first two examples are equivalent.

9.6.2 Adding to a Library

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

```
A>lib86 math = math.186,sin,cos,tan

A>lib86 math = sqrt,math.186
```

9.6.3 Replacing a Module

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

LIB86 newlibrary = oldlibrary [REPLACE [replace list]] where NEWLIBRARY is the name of the new library file you wish to create; OLDLIBRARY is the name of the existing library file (that can be the same as NEWLIBRARY) containing the module you want to replace; and REPLACE LIST contains one or more module names of the form:

```
modulename = filename
```

For example, the command

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

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

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

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

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

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

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

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

9.6.4 Deleting a Module

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

```
LIB86 newlibrary = oldlibrary [ DELETE [module specifiers] ]
```

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

```
A>lib86 math = math.186 [delete [sqrt] ]

A>lib86 math = math.186 [delete [add, sub, mul, div] ]

A>lib86 math = math.186 [delete [add - div] ]
```

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

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

9.6.5 Selecting a Module

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

```
LIB86 newlibrary = oldlibrary [ SELECT [module specifiers] ]
```

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

```
A>lib86 arith = math.186 [select [add, sub, mul, div] ]

A>lib86 arith = math.186 [select [add - div] ]
```

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

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

9.7 Displaying Library Information

You can use LIB-86 to obtain information about the contents of a library. LIB-86 can produce two types of listing files: a Cross-reference file and a Library Module Map. Normally, LIB-86 creates these listing files on the default drive, but you can route them directly to the console or the printer by using the command options described in Section 9.5.

9.7.1 Cross-reference File

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

```
LIB86 libraryname [XREF]
```

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

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

9.7.2 Library Module Map

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

```
LIB86 libraryname [MAP]
```

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

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

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

```
A>lib86 math.186 [map,noalpha]
```

9.7.3 Partial Library Maps

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

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

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

```
A>lib86 math.186 [map,noalpha,select [sin,cos,tan]]

A>lib86 math.186 [xref,select [sin,cos,tan]]
```

9.8 LIB-86 Commands on Disk

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

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

```
LIB86 filename [INPUT]
```

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

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

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

Then the command

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

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

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

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

9.9 Redirecting I/O

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

```
A>lib86 e:math = math.186,d:sin,d:cos,d:tan
```

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

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

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

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

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

The $O switch remains in effect as LIB-86 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
```

End of Section 9

SID-286 Operation

10.1 Introduction

SID-286 is a powerful symbolic debugger designed for use with the FlexOS operating system for the Intel 80286 processor. SID-286 features:

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

Before using SID-286, you should be familiar with the 80286 processor, and the FlexOS 286 operating system as described in the FlexOS System Guide and the FlexOS Programmer's Guide.

SID-286 operation is similar to that of SID-86. If you are already familiar with SID-86, you should review the following sections of this guide:

- Command Options in Section 10.3.1
- Command Conventions in Section 12.3
- Commands in Section 12.4
- Sample Sessions in Section 15.

10.2 Typographical Conventions

Several typographical conventions are used in this guide to more clearly illustrate SID-286's command and output structures. The conventions are:

- Commands appear in UPPERCASE characters and their arguments appear in lower case characters. This convention is used to distinguish the command from its arguments. Typically, you enter all SID-286 command characters in lower case.
- When an example of a SID-286 command is given, user input is displayed in **bold print**.
- Some of the examples of SID-286 output use horizontal and/or vertical ellipses (.....) to illustrate the continuation of an output pattern.
- A <ctrl> sign is used to illustrate the CONTROL (or CTRL) key on your keyboard. For example, <ctrl>D instructs you to press and hold down the CONTROL key while you press the "D" key.
- [] are used to signify an optional parameter

10.3 Starting SID-286

You start SID-286 by entering a command in one of the following forms:

```
SID [options]
SID dbfilespec [symfilespec] [options]
```

The first form loads and executes SID-286. After displaying its sign-on message and prompt character (#) SID-286 is ready to accept your commands.

In the second form, **dbfilespec** specifies an optional pathname, followed by the name of the file to be debugged. If you do not specify a pathname, then SID-286 uses the pathname currently specified in the pathname table. If you do not enter a filetype, SID-286 assumes a 286 filetype. **Symfilespec** specifies the optional symbol file, with or without file extension and pathname (a SYM file extension and the current directory are assumed by default). **Options** is one or more of the options described in Section 10.3.1 below. If both **symfilespec** and **options** are used, they must be entered on the command line in the order shown above.

10.3.1 SID-286 Command Options

The command options available for SID-286 are divided into two categories: process control options and windowing options.

Process Control Options

$t	this option is followed by the arguments for the process being debugged. Use this option with non-interactive programs (such as fgrep) where one or more arguments are required for that debugged program.
$m	this option is followed by a decimal value specifying the maximum memory size (in kilobytes) for process being debugged. A large maximum memory size is recommended.
$f	this option is followed by the name of the macro file to be read. A macro file is a file containing a list of macros. When creating a macro file, enter macros using the same form described in Section 12.4.28, only do not enter the preceding colon (:). Each macro you enter into the macro file must start on a new line and end with a carriage return. A macro file can contain up to 29 macros.

Windowing Options

When using SID-286 to debug your program, you often find it necessary to execute sections of your code. The process of executing your code under SID-286 is referred to as the "debug process." SID-286 allows you to dedicate a section of your screen to the debug process. This section of your screen is referred to as the "debug process window."

The following options determine the size of the debug process window during the debug process as well as the size of the window when the debug process is complete and control is returned to the SID-286 command screen. All of the window sizes are entered as decimal values.

$r	the decimal value following this option specifies the maximum horizontal (row) size of the debug process window. The range is typically 0 - 25, but the maximum size can vary between systems. The default value is the same size as SID-286's original window size.
$c	the decimal value following this option specifies the maximum vertical (column) size of the debug process window. The range is typically 0 - 80, but the maximum size can vary between systems. The default value is the same size as SID-286's original window size.
$w	the decimal value following this option specifies the continuous vertical size of the debugged process window. When you establish the debug process window size by means of the $r and $c options, you can use the $w option to specify how much of the debug process window will remain showing at all times. This value can be modified from the SID-286 command line by the O command described in Section 12.4.14.

Options must be located at the end of the command line. Multiple options must be separated by white space (tabs or blank space).

SID-286 does not allow multiple executable or symbol files to be loaded and debugged simultaneously. When the program and symbol files are loaded for execution, the former symbol tables are overwritten.

10.3.2 SID-286 Example Commands

The following are examples of valid SID-286 command lines:

```
A>SID
```

Start SID-286

```
A>SID hello.286
```

Start SID-286 and load the command file hello.286 as the debug process.

```
A>SID b:hello b:hello
```

Start SID-286 and load the command file, hello.286, along with the symbol table file, hello.sym, from the B drive.

```
A>SID fgrep fgrep $m128 $t"trail" *.asm *.plm
```

Start SID-286 and load the command and symbol files for fgrep. Specify the maximum memory size for fgrep as 12800 bytes. The command tail "trail" *.asm *.plm are the arguments for fgrep, which specify fgrep to search for the string "trail" in the files *.asm and *.plm.

```
A>SID fgrep $t"farewell" *.c
```

Start SID-286 and load the command file for fgrep. The command tail "farewell" *.c are given as the arguments.

```
A>SID $w10 $r15 $c50
```

Start SID-286, setting the window size of the debug process to a horizontal size of 15 and a vertical size of 50. The w10 option specifies that 10 rows of the debug process are displayed when SID-286 is invoked.

```
A>SID $fmacfile
```

Start SID-286, reading in the macros listed in the macro file, macfile.

10.4 Exiting SID-286

When you exit SID-286, no files are automatically saved. Therefore, to save the modified version of your file, write the file to disk using the W (write) Command described in Section 12.4.23 before exiting SID-286.

You can exit SID-286 by typing Q in response to the # prompt (see Section 12.4.16). This returns control to the operating system.

End of Section 10

SID-286 Expressions

11.1 Introduction

This section describes the types of expressions you can use with the commands described in Section 12.

SID-286 can reference absolute machine addresses through expressions. Expressions can use names from the program's SYM file, which is created when the program is linked using LINK 86. Expressions can also be literal values in hexadecimal, decimal, or ASCII character string form. You can combine these literal values with arithmetic operators to provide access to subscripted and indirectly-addressed data or program areas.

11.2 Literal Hexadecimal Numbers

SID-286 normally accepts and displays values in hexadecimal. Valid hexadecimal digits consist of the decimal digits 0 through 9 and the hexadecimal digits A, B, C, D, E, and F, which correspond to the decimal values 10 through 15, respectively.

A literal hexadecimal number in SID-286 consists of one or more contiguous hexadecimal digits. If you type four digits, the leftmost digit is most significant and the rightmost digit is least significant. If the number contains more than four digits, the rightmost four are recognized as significant, and the remaining leftmost digits are discarded. The following examples show the hexadecimal and the decimal equivalents of the corresponding input values.

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

11.3 Literal Decimal Numbers

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

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

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

11.4 Literal Character Values

SID-286 accepts one or two graphic ASCII characters enclosed in apostrophes as literal values in expressions. Characters remain as typed within the apostrophes (that is, no case translation occurs). The leftmost character is the most significant, and the rightmost character is the least significant. Single character strings are padded on the left with zeros. Strings having more than two characters are not allowed in expressions, except in the S command, as described in Section 12.4.18.

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

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

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

11.5 Register Values

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

```
DDS:BX
```

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

```
DBX
```

Note that when assembling 80286 instructions using the A command, register names are treated differently than in other expressions. In particular, a register name in an assembly language statement entered in the A (Assemble) command refers to the name of a register, and not its contents.

11.6 Stack References

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

```
command ^
```

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

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

```
D^
```

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

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

See Section 12.4.4 for a description of the D command.

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

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

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

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

```
G,CS:IP
```

See Section 12.4.7 for a description of the G Command.

11.7 Symbolic References

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

```
.s
@s
=s
```

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

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

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

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

and given the following memory values:

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

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

Symbol Reference	Hexadecimal Value
.GAMMA	CB0:0100
.DELTA	CB0:0102
@GAMMA	3E02
@DELTA	224D
=GAMMA	0002
=DELTA	004D

11.8 Qualified Symbols

Duplicate symbols can occur in the symbol table due to separately assembled or compiled modules that independently use the same name for different subroutines or data areas. Block structured languages allow nested name definitions that are identical, but nonconflicting. Thus, SID-286 allows reference to "qualified symbols" that take the form

```
S1/S2/ . . . /Sn
```

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

SID-286 always searches the symbol table from the first to last symbol in the order the symbols appear in the symbol file. For a qualified symbol, SID-286 begins by matching the first **S1** symbol, then searches for a match with symbol **S2**, continuing until symbol **Sn** is matched. If this search and match procedure is not successful, SID-286 prints a ? to the console. Suppose, for example, that part of the symbol table has a segment address of D00 appearing in the symbol file as follows:

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

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

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

11.9 Expression Operators

Literal numbers, strings, and symbol references can be combined into symbolic expressions using unary and binary "+" and "-" operators. SID-286 evaluates the expression from left to right, producing a 32-bit address at each step. Overflow and underflow are ignored as the evaluation proceeds. The final value becomes the command parameter, whose interpretation depends upon the particular command letter preceding it.

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

The - symbol causes SID-286 to subtract the literal number or symbol reference from the 16-bit value accumulated thus far in the symbolic expression. If the expression begins with a minus sign, then the initial accumulated value is taken as zero. That is,

-x is computed as 0-x

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

```
0700-100
```

is the same as the address

```
0600
```

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

```
DFD00,+#512
```

displays the memory from offset address FD00 to FF00. SID-286 does not allow use of the unary plus operator at other times.

11.10 Sample Symbolic Expressions

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

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

Given these data areas, the command

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

displays the entire area reserved for sorting as follows:

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

The command

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

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

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

The command

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

displays only LIST(I).

Finally, the command

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

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

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

End of Section 11

SID-286 Commands

12.1 Introduction

This section defines SID-286 commands and their arguments. SID-286 commands give you control of program execution and allow you to display and modify system memory and the CPU state.

12.2 SID-286 Command Summary

Table 12-1 summarizes SID-286 commands. SID-286 commands are defined individually in Section 12.4.

Table 12-1. SID-286 Command Summary

Command	Action
A	enter assembly language statements
B	compare blocks of memory
C	change memory (same as the S command)
D	display memory in hexadecimal and ASCII
E	load program and symbols for execution
F	fill memory block with a constant
G	begin execution with optional breakpoints
H	hexadecimal arithmetic
I	change standard output (stdout) file
K	redefine CTRL-A and CTRL-D keys
L	list memory using 8086 mnemonics
M	move memory block
N	automatic "GO" to next call/callf
O	define size of debug process window
P	set, clear, display pass points

Table 12-1. (continued)

Command	Action
Q	quit SID-286 or stop debug process
R	read disk file into memory
S	set memory to new values
SR	search for string within memory
T	trace program execution
U	untraced program monitoring
V	show memory layout of disk file read
W	write contents of memory block to disk
X	examine and modify CPU state
Z	dump 80287 register.
?	print list of SID-286 commands
??	print list of SID-286 commands with options
=	use a previously defined macro
:	define or redefine a macro

12.3 SID-286 Command Conventions

When SID-286 is ready to accept a command, it prompts you with a pound sign, #. This is the SID-286 prompt after which you can enter one of the SID-286 commands described in this section, or you can type a CTRL-C to end the debugging session (see Section 10.4). A valid SID-286 command can have up to 256 characters and must be terminated with a carriage return.

12.3.1 Command Structure

A SID-286 command can be followed by one or more arguments. The arguments can be symbolic expressions, filenames, or other information, depending on the command. Arguments are separated from each other by commas or spaces. Several commands (D, G, N, P, S, T, and U) can be preceded by a minus sign. The effect of the minus sign varies among commands. See the commands in Section 12.4 for explanations of the effects of the minus sign on each command.

12.3.2 Specifying an Address

Most SID-286 commands require one or more addresses as operands. Enter an address as follows:

```
    ssss:0000
```

where **ssss** represents an optional 16-bit segment number and **0000** is a 16-bit offset. If you omit the segment value, SID-286 uses a default value appropriate to the command being executed, as described in Section 12.4.5.

SID-286 does not allow you to randomly access any area in memory. When using SID-286, your access is limited to areas read into SID-286 using the R (Read) command and areas of a debugged process read into SID-286 using the E (Load for Execution) command or by means of the invocation line. It is not possible to simultaneously have a debugged process (read in by the E command or command line) and a file (read in by the R command) resident in SID-286 at the same time. (See Sections 12.4.5 and 12.4.17 for descriptions of the E and R commands.)

12.3.3 Line Editing Functions

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

CTRL-X	erase from beginning of line to cursor
CTRL-H	move cursor to left
CTRL-R	move cursor to right

SID-286 does not process the command line until you enter a carriage return.

12.4 SID-286 Commands

This section describes each SID-286 command in order of alphabetic precedence.

12.4.1 A (Assemble) Command

The A command assembles 80286 mnemonics directly into memory. It takes the form:

    As

where **s** is the address where assembly begins. SID-286 responds to the A command by displaying the address of the memory location where assembly begins. At this point, you can enter assembly language statements as described in Section 14. When a statement is entered, SID-286 converts it to binary, places the value(s) in memory, and displays the address of the next available memory location. This process continues until you press the carriage return without entering any statement or after entering only a period.

SID-286 responds to invalid statements by displaying a question mark, ?, and redisplaying the current assembly address.

Note that wherever a numeric value is valid in an assembly language statement, you can also enter an expression. There is one difference between expressions in assembly language statements and those appearing elsewhere in SID-286: under the A command, references to registers refer to the names of the registers, while elsewhere they refer to the contents of the registers. When the A command is used, you cannot reference the contents of a register in an expression.

The following is an example of the A command:

```
#a213
```

Assemble at offset 213 of the current default CS value.

```
nnnn:0213 mov ax,#128
```

Set AX register to decimal 128.

```
nnnn:0216 push ax
```

Push AX register on stack.

```
nnnn:0217 call .procl
```

Call procedure whose address is the value of the symbol PROC1.

```
nnnn:021A test byte [.i/i], 80
```

Test the most significant bit of the byte whose address is the value of the second occurrence of the symbol I.

```
nnnn:021E jz .done
```

Jump if zero flag set to the location whose address is the value of the symbol DONE.

```
nnnn:0220 .
```

stop assemble process.

12.4.2 B (Block Compare) Command

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

```
Bs1,f1,s2
```

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

SID-286 displays any differences in the two blocks in the form:

```
a1 b1 a2 b2
```

where the **a1** and the **a2** are the addresses in the blocks; **b1** and **b2** are the values at the indicated addresses. If no differences are displayed, the blocks are identical.

The following are examples of the B command:

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

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

12.4.3 C (Change Memory) Command

The C command changes the contents of memory. This command is identical to the S command described in Section 12.4.18.

12.4.4 D (Display) Command

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

```
D
Ds
Ds,f
DW
DWs
DWs,f
-Dn
-D
```

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

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

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

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

The **D** form displays memory from the current display address for 12 display lines. Form **Ds** is similar to form **D**, except the default display address is changed to address **s**. Form **Ds,f** displays the memory block between locations **s** and **f**. Forms **DW**, **DWs**, and **DWs,f** are identical to forms **D**, **Ds**, and **Ds,f**, except that the contents of memory are displayed as 16-bit words, rather than 8-bit bytes, as follows

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

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

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

The last address displayed becomes the default starting address for the next display unless another starting address is specified. Only memory loaded into SID-286 can be displayed. Once the end of memory loaded into SID-286 is reached, the display stops, regardless of the ending address specified by the user.

The **-Dn** form changes the default number of bytes displayed when the D command is invoked. The number of bytes displayed by the D command is 176 by default. The **-Dn** form changes this default value to one specified by **n**, which can be any number between 0 and 65535. The **-D** form of the D command changes the default number of bytes displayed back to 176.

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

The following are examples of the D command:

`#df00,f23`	Display memory bytes from offset F00H through F23H in the current data segment.
`#d.array+=i,+#10`	Display 10 bytes starting at location ARRAY (i).
`#dwss:sp`	Display the value at the top of stack in word format.
`#d^`	Display the value at the top of stack in byte format.
`#dw#128,#255`	Display memory words from offset 80H through FFH.
`#-d10`	Set the default number of bytes displayed to 16.
`#-d`	Set the default number of bytes displayed to 176.

12.4.5 E (Load Program, Symbols for Execution) Command

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

```
Efilespec
Efilespec  symfilespec
```

The **Efilespec** form loads the command file specified by **filespec**. Where **filespec** is an optional pathname and the name of the command file you want to load. If you do not specify a full pathname, then SID-286 uses the pathname(s) currently specified in the pathname table. If you do not enter a filetype for the file, SID-286 assumes a 286 filetype. SID-286 alters the contents of the CS, DS, ES, and IP registers according to the information in the header of the file loaded. When the file is completely loaded, SID-286 displays the start and end addresses of each segment in the file. You can use the V command to redisplay this information later. See the V Command in Section 12.4.22.

The **Efilespec symfilespec** form loads the command file specified by **filespec** as described above, and then loads a symbol file as specified in **symfilespec**. The default filetype for a symbol file is SYM. SID-286 displays the message

```
SYMBOLS
```

when it begins loading the symbol file. If SID-286 detects an invalid hexadecimal digit or an invalid symbol name, it displays an error message and stops loading the symbol file. You can use the H command to display the symbols loaded when the error occurred to determine the exact location of the error in the SYM file. 64K bytes of memory is available for symbol table storage.

When loading a program file with the E command, SID-286 releases any blocks of memory allocated by any previous E or R command or by programs executed under SID-286. Therefore, only one file at a time can be loaded for execution and that file should be loaded before any symbol tables are read.

SID-286 issues an error message if a file does not exist or cannot be successfully loaded in the available memory.

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

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

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

The following are examples of the E command:

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

12.4.6 F (Fill) Command

The F command fills an area of memory read into SID-286 using an E command, R command, or command line with a byte or word constant. The forms of the F command are:

```
Fs,f,b
FWs,f,w
```

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

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

If **s** is greater than **f**, or the value **b** is greater than 255, SID-286 responds with a question mark.

The following are examples of the F command:

`#f100,13f,0`	Fill memory at the current default display segment from offsets 100H through 13FH with 0.
`#f.array,+255,ff`	Fill the 256-byte block starting at ARRAY with the constant FFH.
`#f-5,+10,'z'`	Fill the 10 bytes beginning at 5 bytes before the current default display address with the ASCII value for z.

12.4.7 G (Go) Command

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

```
G
G,b1
G,b1,b2
Gs
Gs,b1
Gs,b1,b2
-G (with all of the above forms)
```

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

In forms **G**, **G,b1**, and **G,b1,b2**, no starting address is specified, so SID-286 gives the address from the CS and IP registers. Form **G** transfers control to your program without setting any breakpoints. Forms **G,b1** and **G,b1,b2** set one and two breakpoints respectively before passing control to your program. Forms **Gs**, **Gs,b1**, and **Gs,b1,b2** are identical to **G**, **G,b1**, and **G,b1,b2**, except the CS and IP registers are first set to **s**.

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

Once SID-286 transfers control to the program under test, it executes in real time until a breakpoint is encountered. At this point, SID-286 regains control, clears the breakpoints set by the **G** command, and displays the address where the executing program is interrupted. This is done using the format:

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

where **ssss** corresponds to the CS register, **0000** corresponds to the IP register where the break occurs, and **.symbol** is the symbol whose value is equal to **0000**, if such a symbol exists. When a breakpoint returns control to SID-286, the instruction at the breakpoint address has not yet been executed.

The following are examples of the G command:

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

12.4.8 H (Hexadecimal Math) Command

The H command provides several useful arithmetic functions. The forms are as follows:

```
Ha,b
Ha
H
H .symbol
```

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

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

Underflow and overflow are ignored in addition and subtraction.

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

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

The **H** form displays the symbols currently loaded in the SID-286 symbol table. Each symbol is displayed in the following form:

```
nnnn symbolname
```

You can stop the display by pressing any key at the console.

The **H .symbol** form allows you to display the address where the specified symbol is defined in the symbol table.

If the symbol is found in the symbol table, SID-286 responds:

```
The symbol is found at address xxxx:xxxx
```

where xxxx:xxxx is the address. If the symbol is not found, SID responds:

```
The symbol does not appear in the symbol table
```

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

```
ffff + 2
```

equals 1.

The following are examples of the H command:

`#h`	List all symbols and values loaded with the E command(s).
`#h@index`	Show the word contents of the memory location at INDEX in hexadecimal and decimal.
`#h5c28,80`	Show sum, difference, product, and quotient of 5C28H and 80H.

12.4.9 I (Redirect Output) Command

SID output is always sent to the screen. In addition to the screen, the I command allows you to direct your debugging session output to a printer or a file. The forms are as follows:

```
I
Ifilespec
```

The **I** form directs debugging session output to the printer. Additional **I** commands toggle output to the printer between off and on. If you use the **Ifilespec** form to redirect your SID output to a file, the **I** form stops the output to the file.

The **Ifilespec** form directs SID output to a file specified by **filespec**. Where **filespec** is an optional pathname and the name of the file you want the SID output directed to. If you do not provide a pathname, SID creates a file under the current default directory. If the file specified by **filespec** already exists, that file is deleted and a new file created.

You cannot simultaneously send SID output to both a printer and a file.

12.4.10 K (Redefine Keys) Command

During execution of an interactive debug process requiring responses from the keyboard, SID-286 directs keyboard input to the debug process. However, entering either a CTRL-A or CTRL-D keystroke automatically returns control of the keyboard to SID-286. The K command is used when the debug process needs to make use of the CTRL-A and CTRL-D keystrokes, which have the hexadecimal values of 1 and 4.

The K command allows you to redefine the return-to-SID function invoked by either the CTRL-A or CTRL-D keys to another key, thereby allowing these keys to be used as intended by the debug process without causing control to be returned to SID-286. The form is as follows:

```
Kn
```

where **n** is the hexadecimal value of the ASCII character you wish to substitute for the CTRL-A and CTRL-D keys. For example, a 1 corresponds to CTRL-A, 4 corresponds to a CTRL-D, 41 corresponds to an A, etc..

The following are examples of the K command:

#K32	Redefine the function normally invoked by the CTRL-A or CTRL-D keys to be invoked by the 2 key (32H). This allows the CTRL-A and CTRL-D keys to be used as defined by the debug process.
#K#90	Redefine the function normally invoked by the CTRL-A or CTRL-D keys to be invoked by the Z key, which is 90 decimal (5AH).

12.4.11 L (List) Command

The L command lists the contents of memory read into SID-286 using the R command, the E command, or the command line in assembly language. The forms are as follows:

```
L
Ls
Ls,f
-L (with all of the above forms)
```

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

Each disassembled instruction takes the form:

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

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

The **L** form lists 12 disassembled instructions from the current list address. The **Ls** form sets the list address to **s** and then lists 12 instructions. The **Ls,f** form lists disassembled code from **s** through **f**. If you precede any of the **L** command forms with a minus sign, no symbolic information is displayed (the labels and **symbol** fields are omitted). This speeds up the listing if many symbols are present and you have no need to display them.

In all forms, the list address is set to the next unlisted location in preparation for a subsequent **L** command. When SID-286 regains control from a program being tested (see G, T, and U commands), the list address is set to the current value of the CS and IP registers.

You can stop long displays by typing any key during the list process. Also, CTRL-S temporarily halts the display, which can be resumed by entering CTRL-Q.

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

If the memory location being disassembled is not a valid 80286 instruction, SID-286 displays

```
    ??=  nn
```

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

The following are examples of the L command:

`#1`	Disassemble 12 instructions from the current default list address.
`#-1`	Disassemble 12 instructions, without symbols, from the current default list address.
`#1243c,244e`	Disassemble instructions from 243CH through 244EH.
`#1.find,+20`	Disassemble 20H bytes from the label FIND.
`#1.err+3`	Disassemble 12 lines of code from the label ERR plus 3.
`#1.err,.err1`	Disassemble from label **err** to label **err1**.

12.4.12 M (Move) Command

The M command copies a block of data values read into SID-286 using an E command, R command, or command line from one area of memory to another. The form is as follows:

```
Ms,f,d
```

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

The following are examples of the M command:

```
#m20:2400,+9,30:100
```

Move 10 bytes from 20:2400 to 30:100.

```
#m.array,+#63,.array2
```

Move 64 bytes from ARRAY to ARRAY2.

12.4.13 N (Transfer Control) Command

The N command executes the G (Go) command to transfer control to the program being tested directly before or after the next call or callf. The forms are as follows:

```
-N
N
```

The **-N** form executes a G command directly before the next call or callf.

The **N** form executes a G command directly after returning from the procedure called by the next call or callf.

12.4.14 O (Control Window) Command

The O command, along with the SID-286 command options $r and $c described in Section 10.3.1, allows you to set the size of the debug process window.

The output generated by SID-286 is sent to the current standard output device. The standard output device used by SID-286 is either a disk file or a screen. If the output is sent to a disk file (as specified by using file redirection ">" when invoking SID-286) then none of the windowing functions described in this section is valid. If the output is sent to the screen, then the commands in this section operates as described.

The forms are as follows:

```
Or,c
-O
-Or
```

where **r** determines the row (horizontal) location of the window (the available range is usually 0 - 25) and **c** determines the column (vertical) location of the window (the range is usually 0 - 80). All of the window sizes are entered as decimal numbers.

The **Or,c** form of the command allows you to move the upper left corner of the debug process window to the coordinates specified by **r** and **c**. The larger the **c** value, the further the debug process window appears to the right on your screen.

The **-O** form of the command resets the value specified either by the SID-286 command option $w or by the **-Or** command to zero. This means that, if you have set the debug process window to remain on the screen when the debug process has completed, the **-O** command overrides this instruction and causes the debug process window to disappear when the debug process has finished.

The **-Or** form sets the vertical (row) size of the debug process window when the debug process is finished to a size specified by **r.** This command resets the value specified by the SID-286 command option $w.

The following are examples of the O command:

#O10,40	Move the upper left corner of the debug process window to a horizontal position of 10 and a vertical position of 40. On most terminals, this command cause the debug process window to take up the upper right quarter of the screen.
#-O	Do not display debug process window after completion of the debug process.
#-O5	Set the debug process window to a horizontal position of 5 when the debug process has completed.

12.4.15 P (Permanent Breakpoint) Command

The P command sets, clears, and displays "permanent" breakpoints. The forms are as follows:

```
Pa,n
Pa
-Pa
-P
P
```

A permanent breakpoint remains in effect until you explicitly remove it, as opposed to breakpoints set with the G command that must be reentered with each G command. Pass points have associated pass counts ranging from 1 to 0FFFFH. The pass count indicates how many times the instruction at the pass point executes before the control returns to the console. SID-286 can set up to 30 permanent breakpoints at a time.

Forms **Pa,n** and **Pa** are used to set pass points. The **Pa,n** form sets a pass point at address **a** with a pass count of **n**, where **a** is the address of the pass point, and **n** is the pass count from 1 to 0FFFFH. If a pass point is already active at **a**, the pass count is changed to **n**. SID-286 responds with a question mark if there are already 16 active pass points.

The **Pa** form sets a pass point at **a** with a pass count of 1. If a pass point is already active at **a**, the pass count is changed to 1. SID-286 responds with a question mark if there are already 16 active pass points.

Forms **-Pa** and **-P** are used to clear pass points. The **-Pa** form clears the pass point at location **a**. SID-286 responds with a question mark if there is no pass point set at **a**. The **-P** form clears all the pass points.

The **P** form displays all the active pass points using the form:

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

where **nnnn** is the current pass count for the pass point; **ssss:0000** is the segment and offset of the pass point location, and **.symbol** is the symbolic name of the offset of the pass point, if such a symbol exists. When a pass point is encountered, SID-286 displays the permanent breakpoint information in the form

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

where **nnnn**, **ssss:0000**, and **.symbol** are as previously described. Next, SID-286 displays the CPU state before the instruction at the permanent breakpoint is executed. SID-286 then executes the instruction at the permanent breakpoint. If the pass count is greater than 1, SID-286 decrements the pass count and transfers control back to the program under test.

When the pass count reaches 1, SID-286 displays the break address (that of the next instruction to be executed) in the following form:

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

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

You can suppress the intermediate pass point display with the -G command (see the G Command in Section 12.4.7). When the -G command is used, only the final pass points (when the pass count = 1) are displayed.

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

Normally, SID-286 does not display the segment registers at pass points. You can use the S and -S commands to enable and disable the segment register display (see the S Command in Section 12.4.18).

The following are examples of the P command:

`#p`	Display active permanent breakpoints.
`#p.error`	Set permanent breakpoint at label ERROR.
`#p.print,17`	Set permanent breakpoint at label PRINT with count of 17H.
`#-p`	Clear all permanent breakpoints.
`#-p.error`	Clear permanent breakpoint at label ERROR.

12.4.16 Q (Quit) Command

The Q command terminates SID-286 if no process is being debugged. If a debug process loaded by the E command is running, the Q command stops the process, but does not terminate SID-286. The form is as follows:

```
Q
```

The Q command stops SID-286 if no process loaded by the E command is running.

12.4.17 R (Read) Command

The R command reads a file into a contiguous block of memory. The form is as follows:

```
Rfilespec
```

where **filespec** is an optional pathname and the name of the file you want to read. When you use the R command, SID-286 automatically determines the memory location into which the file is read.

When you enter the R command after a process is loaded with the E command, or from the command line during the debugging process, the process being debugged is stopped. Similarly, entering an E command erases the buffered information formed by the R command.

When you enter the R command, SID-286 reads the file into memory, computes, allocates, and displays the start and end addresses of the block of memory occupied by the file. You can use the V command to redisplay this information at a later time. SID-286 sets the default display pointer for subsequent D commands to the start of the block occupied by the file.

The R command does not free any memory previously allocated by another R command. Therefore, you can read a number of files into memory without them overlapping. When the R command is used, files are concatenated together in memory in the same order in which they were read in.

SID-286 issues an error message if the file does not exist or there is not enough memory to load the file.

The following are examples of the R command:

`#rbanner.286`	Read file BANNER.286 into memory.
`#rtest`	Read file TEST into memory.

12.4.18 S (Set) Command

The S command changes the contents of bytes or words of memory read into SID-286 with the E command, R command, or command line. The forms are as follows:

```
Ss
SWs
S
-S
```

where **s** is the address where the change occurs.

SID-286 displays the memory address and its current contents on the following line. In response to the **Ss** form, the display is

```
ssss:0000 bb
```

where **bb** is the contents of memory in byte format. In response to the **SWs** form, the display is

```
ssss:0000 wwww
```

where **wwww** is the contents of memory in word format.

You can choose to alter the memory location or to leave it unchanged. If you enter a valid expression, the contents of the byte (or word) in memory is replaced with the value of the expression. If you do not enter a value, the contents of memory are unaffected and the contents of the next address are displayed. In either case, SID-286 continues to display successive memory addresses and values until you enter a period on a line by itself or until SID-286 detects an invalid expression.

In response to the **Ss** form, you can enter a string of ASCII characters, beginning with a quotation mark and ending with a carriage return. The characters between the quotation mark and the carriage return are placed in memory starting at the address displayed. No case conversion takes place. The next address displayed is the address following the character string.

SID-286 issues an error message if the value stored in memory cannot be read back successfully, indicating faulty or nonexistent memory at the location indicated.

The forms **S** and **-S** control the display of the segment registers when the CPU state is displayed with the T (Trace) command and at pass points. Form **S** turns on the segment register display and form **-S** turns it off. You can turn off the segment register display while debugging to allow the CPU state display to fit on one line.

The following are examples of the S command.

`#s.array+3`	Begin set at ARRAY (3)
`nnnn:1234 55 0`	Set byte to 0.
`nnnn:1235 55 'abc'`	Set three bytes to a, b, c.
`nnnn:1238 55 #75`	Set byte to decimal 75.
`nnnn:1239 55 .`	Terminate set command.
`#s`	Enable segment register display in CPU state display.
`#-s`	Disable segment register display in CPU state display.

12.4.19 SR (Search for String) Command

The SR command searches for a string of characters of values within memory. The forms are as follows:

```
SRs,f,"string"
SRs,f,value
```

where **s** is the starting address to begin searching and **f** is the finishing address to end searching.

The **SRs,f,"string"** form searches for a string of 1 to 30 printable ASCII characters. The **"string"** parameter specifies the string to be searched for. Note that **string** can use either single (') or double (") quotes.

The **SRs,f,value** form searches for a numerical hex value anywhere between 0 and FFFFFFFFh in size. The **value** parameter must be a hexadecimal number within the range specified above (leading 0's do not need to be specified).

The size of the field searched depends on the value of the **value** parameter according to the following chart:

```
number between:          number of bytes in
                         search pattern:

0        .. FFh              1
100h     .. FFFFh            2
10000h   .. FFFFFFh          3
10000000.. FFFFFFFFh         4
```

Both the **SRs,f,"string"** form and the **SRs,f,value** form of the SR command can search for the same things, since a numerical value also equals an ASCII value.

The following are examples of the SR command:

#SR56:00,56:1ff,D0A
: search memory starting at 56:00 and ending at 56:1FF for a two-byte value consisting of 0Dh (Carriage Return) and 0Ah (line feed).

#SR56:1ff,56:d0ff,"ABCD"
: search memory starting at 56:1FF and ending at 56:d0ff for the character string: ABCD.

#SR56:iff,56:d0ff,41424344
: search memory starting at 56:1FF and ending at 56:d0ff for a four-byte value consisting of 41 (A), 42 (B), 43 (C), and 44 (D).

12.4.20 T (Trace) Command

The T command traces program execution for 1 to 0FFFFH program steps. After each trace, SID-286 displays the current state of the CPU and the next disassembled instruction to be executed. You must read programs into SID-286 with the E command or command line. The forms are as follows:

```
T
Tn
TW
TWn
-T (with all of the above forms)
```

where **n** is the number of program steps to execute before returning control to the console. If you do not specify the number of program steps, SID-286 executes a single program step. Pressing any key stops the trace process.

A program step is generally a single instruction, with the following exceptions:

- If a BDOS interrupt instruction is traced, the entire BDOS function is treated as one program step and executed in real time. This is because SID-286 makes its own BDOS calls and the BDOS is not reentrant.
- If the traced instruction is a MOV or POP whose destination is a segment register, the CPU executes the next instruction immediately. This is due to a feature of the 80286 that disables interrupts, including the Trace Interrupt, for one instruction after a MOV or POP loads a segment register. This allows a sequence such as

  ```
  MOV   SS, STACKSEGMENT
  MOV   SP, STACKOFFSET
  ```

  to be executed with no chance of an interrupt occurring between the two instructions, at which time the stack is undefined. Such a sequence of MOV or POP instructions, plus one instruction after the sequence is considered one program step.
- If you use any of the **TW** forms and the traced instruction is a CALL, CALLF, or INT, the entire called subroutine or interrupt handler (and any subroutines called therein) is treated as one program step and executes in real time.

After each program step is executed, SID-286 displays the current CPU state, the next disassembled instruction to be executed, the symbolic name of the instruction operand (if any), and the contents of the memory location(s) referenced by the instruction (if appropriate). See the X Command in Section 12.4.24 for a detailed description of the CPU state display.

If a symbol has a value equal to the instruction pointer (IP), the symbol name followed by a colon is displayed on the line preceding the CPU state display. The segment registers are normally not displayed with the T command, which allows the entire CPU state to be displayed on one line. Use the S command, as described in Section 12.4.18, to enable the segment register display. With the segment register display enabled, the display of the CPU state is identical to that of the X command.

In all of the forms, control transfers to the program under test at the address indicated by the CS and IP registers. If you do not specify the number of program steps, as in form **T**, one program step is executed. Otherwise, SID-286 executes **n** program steps and displays the CPU state before each step, as in form **Tn**. You can stop a long trace before **n** steps are executed by typing any character at the console.

After **n** steps are executed, SID-286 displays the address of the next instruction to be executed, along with the symbolic value of the IP register (if there is such a symbol) in the following form:

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

Forms **TW** and **TWn** trace execution without breaking for calls to subroutines. The entire subroutine called from the program being traced is treated as a single program step and executed in real time. This allows tracing at a high level of the program, ignoring subroutines already debugged.

If you precede the command with a minus sign, SID-286 omits symbolic labels and symbolic operands from the CPU state display. This can speed up the display by skipping the symbol table lookup when large symbol tables are loaded.

When a single instruction is being traced, interrupts are disabled for the duration of the instruction. This prevents SID-286 from tracing through interrupt handlers when debugging on systems in which interrupts occur frequently.

After a T command, SID-286 sets the list address used in the L command at the address of the next instruction to be executed. SID-286 also sets the default segment values to the CS and DS register values.

The following are examples of the T command:

`#t`	Trace one program step.
`#tffff`	Trace 65535 steps.
`#-t#500`	Trace 500 program steps with symbolic lookup disabled.

12.4.21 U Command

The U command, like the T command, is used to trace program execution. The U command functions in the same way as the T command, except that the CPU state is displayed after the last set of program steps have executed, rather than after every step.

The U command works only with programs loaded by the E command or from the command line. The forms are as follows:

```
U
Un
UW
UWn
-U (with all of the above forms)
```

where **n** is the number of instructions to execute before returning control to the console. You can stop the U command before **n** steps are executed by pressing any key.

Forms **UW** and **UWn** trace execution without calls to subroutines. The entire subroutine called from the program being traced is treated as a single program step and executed in real time. This allows tracing at a high level of the program, ignoring subroutines already debugged.

Preceding any of the **U** command forms with a minus sign causes SID-286 not to print any symbolic reference information. This allows the program to execute faster.

The following are examples of the U command:

#u200	Trace without display 200H steps.
#-u200	Trace without display 200H steps, suppressing the intermediate pass point display.

12.4.22 V (Value) Command

The V command displays information about the last file loaded with the E or R commands, excluding symbol tables loaded with the E command. The form is as follows:

```
V
```

If you load the last file with the R command, the V command displays the start and end addresses of the file. If you read the last file with the E command, the V command displays the start and length in bytes for the code, data, and heap segments.

If an R or E command have not have been used, SID-286 responds to the V command with a question mark (?).

12.4.23 W (Write) Command

The W command writes the contents of a contiguous block of memory to disk. This command requires you to first use the R command to read the data into SID-286. The forms are as follows:

```
Wfilespec
Wfilespec,s,f
```

where **filespec** is an optional pathname and the name of the file you want to receive the data. The **s** and **f** arguments are the first and last addresses of the block to be written. If you do not specify the segment in **f**, SID-286 uses the same value used for **s**.

When you use the **Wfilespec** form, SID-286 assumes the first and last addresses from the files read with an R command. This causes all of the files read with the R command to be written. If no file is read with an R command, SID-286 responds with a question mark, ?. Use the **Wfilespec** form for writing out files after patches are installed, assuming the overall length of the file is unchanged.

The **Wfilespec,s,f** form allows you to write the contents of a specific memory block. The first address of the memory block is specified by **s** and the last address of the memory block is specified by **f**.

If a file with the name specified in the **W** command already exists, SID-286 deletes it before writing a new file.

The following are examples of the W command:

`#wtest.cmd` Write to the file TEST.CMD the contents of the memory block read into by the most recent R command.

`#wb:test.cmd,40:0,3fff`
Write the contents of the memory block 40:0 through 40:3FFF to the file TEST.CMD on drive B.

12.4.24 X (Examine CPU State) Command

The X command allows you to examine and alter the CPU state of the program under test. The forms are as follows:

X
Xr
Xf

where **r** is the name of one of the 80286 CPU registers and **f** is the abbreviation of one of the CPU flags. The **X** form displays the CPU state in the following format:

```
                 AX   BX   CX     . . .    SS   ES   IP
      ---------xxxx xxxx xxxx     . . .   xxxx xxxx xxxx

instruction  symbol name                 memory value
```

The nine hyphens at the beginning of the line indicate the state of the nine CPU flags. Each position can be either a hyphen, indicating that the corresponding flag is not set (0), or a single-character abbreviation of the flag name, indicating that the flag is set (1). The abbreviations of the flag names are shown in Table 3-1.

Table 12-2. Flag Name Abbreviations

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

In the X form, **instruction** is the disassembled instruction at the next location to be executed, which is indicated by the CS and IP registers. If the symbol table contains a symbol whose value is equal to one of the operands in **instruction**, the symbol name appears in the **symbol name** field, preceded by a period. If instruction references memory, the contents of the referenced location(s) appear in the **memory value** field, preceded by an equal sign. Either a byte, word, or double word value is shown, depending on the instruction. In addition to displaying the machine state, the **X** form changes the values of the default segments back to the CS and DS register values, and the default offset for the L command to the IP register value.

The **Xr** form allows you to alter the registers in the CPU state of the program being tested. The **r** following the **X** is the name of one of the 16-bit CPU registers. SID-286 responds by displaying the name of the register followed by its current value. If you type a carriage return, the value of the register does not change. If you type a valid expression, the contents of the register change to the value of the expression. In either case, the next register is then displayed. This process continues until you enter a period or an invalid expression, or the last register is displayed.

The **Xf** form allows you to alter one of the flags in the CPU state of the program being tested. SID-286 responds by displaying the name of the flag followed by its current state. If you type a carriage return, the state of the flag does not change. If you type a valid value, the state of the flag changes to that value. You can examine or alter only one flag with each Xf command. You set or reset flags by entering a value of 1 or 0.

The following are examples of the X command.

```
#xbp                Change registers starting with BP.
BP=1000  2b64       Change BP to hex 2B64.
SI=2000  #12345     Change SI to decimal 12345.
CS=0040  .          Terminate X command.
```

12.4.25 Z (Print 8087/80287 Registers) Command

The Z command prints out the contents of 8087 and 80287 registers. The form is as follows:

```
Z
```

The output generated by the Z command looks like the following:

```
 CW    SW    TW        IP          OP
037F  4100  FFFF  FFEF  07FF  FFFF  0000

0  EEC0  A6B6  B596  EB8A  BAD5
1  EEC0  A6B6  B55E  EB8B  BAD5
2  EEC0  A6A6  B5D6  EB8A  BAD5
3  EEC0  A6B6  B55E  EB8B  BAD5
4  EEC0  A6B6  B55E  EB8B  BAD5
5  EEC0  A6B6  B55E  EB8B  BAD5
6  EEC0  A6B6  B55E  EB8B  BAD5
7  EEC0  A6B6  B55E  EB8B  BAD5
```

Where:

CW = Control Word Format
SW = Status Word Format (indicates physical register 0)
TW = Tag Word
IP = 8087 or 80287 Instruction Pointer
OP = Pointer for last operand fetched

An error message appears if no 8087 or 80287 processor is present when the Z command is given.

12.4.26 ? (List Commands) Command

The ? command prints a list of available SID-286 commands, similar to the list appearing in Table 1-1. The form is as follows:

```
?
```

12.4.27 ?? (List Commands Format) Command

The ?? command prints a detailed command list that includes the SID-286 commands, and the available command options. The form is as follows:

```
??
```

12.4.28 : (Define Macro) co‹ımand

The : command defines or redefines a macro. The form is as follows:

```
:name;value
```

where **name** is the name of the macro used to invoke **value**, and **value** is the command the macro defines.

For example, if you wish to create a macro named "s" that, when invoked, prints out the contents of the stack, you would define the macro as follows:

```
#:s;dwss:sp
```

12.4.29 = (Use Macro) Command

The = command causes SID-286 to use a previously defined macro. The forms are as follows:

```
=
= name
```

The = form prints out the list of existing SID-286 macros and their definitions (values).

The = **name** form executes the command associated with **name**.

For example, to invoke the "s" macro defined in the example for the : command, enter

```
#=s
```

and the contents of the stack will print.

End of Section 12

Default Segment Values

13.1 Introduction

SID-286 has an internal mechanism that keeps track of the current segment value, making segment specification optional when entering a SID-286 command. SID-286 divides the command set into two types according to which segment a command defaults if you do not specify a segment value in the command line.

13.2 Type-1 Segment Value

The A (Assemble), L (List Mnemonics), P (Pass Points), and R (Read) commands use the internal type-1 segment value if you do not specify a segment value in the command.

When started, SID-286 sets the type-1 segment value to 0 and changes it when one of the following actions is taken:

- When an E command loads a file, SID-286 sets the type-1 segment value to the value of the CS:IP register.
- When an R command reads a file, SID-286 sets the type-1 segment value to the base segment where the file was read.
- When an X command changes the value of the CS:IP register, SID-286 changes the type-1 segment value to the new value of the CS:IP register.
- When SID-286 regains control from a user program after a G, T, or U command, it sets the type-1 segment value to the value of the CS:IP register.
- When an A or L command explicitly specifies a segment value, SID-286 sets the type-1 segment value to the segment value specified.

13.3 Type-2 Segment Value

The D (Display), F (Fill), M (Move), and S (Set) commands use the internal type-2 segment value if you do not specify a segment value in the command.

When invoked, SID-286 sets the type-2 segment value to 0 and changes it when one of the following actions is taken:

- When an E command loads a file, SID-286 sets the type-2 segment value to the value of the DS register.
- When an R command reads a file, SID-286 sets the type-2 segment value to the base segment where the file was read.
- When a D, F, M, or S command explicitly specifies a segment value, SID-286 sets the type-2 segment value to the segment value specified.

When evaluating programs with identical values in the CS and DS registers, all SID-286 commands default to the same segment value unless explicitly overridden.

Table 13-1 summarizes the SID-286 default segment values.

Table 13-1. SID-286 Default Segment Values

Command	Default Segment Value
A	Current CS:IP of debugged process
B	Current display address
C	Current display address
D	Current display address
E	No default values assumed
F	Current display address
G	Current CS:IP of debugged process
H	No default values assumed
I	No default values assumed

Table 13-1. (continued)

Command	Default Segment Value
L	Current list address
M	Current display address
N	Current CS:IP of debugged process
P	Current CS:IP of debugged process
Q	No default values assumed
R	Default is beginning address of the file
S	Current display address
SR	No default values assumed
T	Current CS:IP of debugged process
U	Current CS:IP of debugged process
V	No default values assumed
W	Default is beginning address of the file
X	No default values assumed
Y	No default values assumed
Z	No default values assumed
:	No default values assumed
=	No default values assumed

End of Section 13

Assembly Language Syntax for A and L Commands

14.1 Assembly Language Exceptions

In general, the syntax of the assembly language statements in the A and L commands is standard 80286 assembly language. Several minor exceptions are listed below.

- Up to three prefixes (LOCK, repeat, segment override) can appear in one statement, but they all must precede the opcode of the statement. Alternately, a prefix can appear on a line by itself.
- The distinction between byte and word string instructions is made as follows:

Byte	Word
LODSB	LODSW
STOSB	STOSW
SCASB	SCASW
MOVSB	MOVSW
CMPSB	CMPSW

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

Short	Normal	Far
JMPS	JMP	JMPF
	CALL	CALLF
	RET	RETF

- If the operand of a CALLF or JMPF instruction is an absolute address, you enter it in the form:

  ssss:0000

  where **ssss** is the segment and **0000** is the offset of the address.

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

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

  Failure to supply a necessary prefix results in an error message.

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

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

- The forms of register indirect memory operands are:

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

  where the pointer registers are BX and BP, and the index registers are SI and DI. Any of these forms can be preceded by a numeric offset. For example:

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

End of Section 14

SID-286 Sample Sessions

15.1 Introduction

The following sample sessions illustrate the commands and procedures used to interactively debug three simple programs.

The first example shows how the RASM file, TEST.A86, is debugged using SID-286.

The second example demonstrates the usage of most of the SID-286 commands when debugging a compiled CBASIC file.

The third session demonstrates how SID-286 is used to patch code in the executable file.

In the SID-286 sessions illustrated below, remarks appear in regular print; user input and SID-286 generated output are shown in oblique print.

15.2 SID-286 Session #1

The following RASM program, TEST.A86, is written to print "polly want a cracker" to the screen fifteen times and then exit to the operating system.

This same file is provided on your distribution disk. As you study this session, try duplicating the steps illustrated by actually using SID-286 to debug the TEST.A86 file.

Type RASM source file, TEST.A86:

```
A> type test.a86

;  test.a86

        dseg
parmblk db      0       ;       parameter block for Write SVC
option  db      0       ;       zero out
flags   dw      200h    ;       flags to write to screen
swi     dw      0,0     ;       no software interrupts
fnum    dw      1,0     ;       logical file num of stdout
bufo    dw      0       ;       message buf address offset
bufs    dw      0       ;       message buffer address
bufsz   dw      23,0    ;       size in bytes of msg buf
offs    dw      0,0

mssg    db      'polly want a cracker',10,13,0

patht   dw      0
;
;
CODE    cseg

start:  mov     ax, ds          ; get value of ds register
print:  mov     bufs, ax        ; that is seg of mssg
        mov     bx, ax          ; also use for int 220 call
        lea     ax, mssg        ; get offset value
        mov     bufo, ax        ; write into param block
        mov     cx, 8           ; write SVC # is 8
loop:   lea     ax, parmblk     ; offset of param block
        int     220             ; write to the screen
        inc     patht           ; loop control
        cmp     word ptr patht, 15
        jne     loop            ; print message out 15 times

exit:   mov     cx, 25          ; exit SVC # is 25
        mov     bx, ds          ; segment val
        lea     ax, swi         ; offset val
        int     220             ; exit and return to the O/S
end
```

Assemble TEST.A86 using RASM-86 by entering the following command:

```
A>rasm86 test
```

RASM-86 creates an OBJ file. Link TEST.OBJ using LINK 86 with the LOCALS option as follows:

```
A>link86 test [locals]
```

LINK 86 generates an executable file with a .CMD filetype and a symbol table file with a .SYM filetype.

After running TEST.CMD, it should print "polly want a cracker" only once before returning control to the operating system. Review the program's source code to see if you can discover the nature of the problem.

Now, use the SID-286 debugger to isolate the cause of the program failure.

Enter SID-286, loading TEST.CMD and TEST.SYM as follows

```
A> sid test test

************************************************
SID-286          12/19/84               Version 1.0
Serial #XXXX-0000-654321        All Rights Reserved
Copyright 1985              Digital Research Inc.
************************************************

Symbols

        address         length
code    3878:0000       00000030
data    387B:0000       00000140
heap    0000:0000       00000000
```

Display 12 disassembled instructions from the current address to get oriented to the sequence of instructions:

```
#l
START:
3878:0000    MOV    AX,DS
PRINT:
3878:0002    MOV    [010E],AX   .BUFS
3878:0005    MOV    BX,AX
3878:0007    LEA    AX,[0118]   .MSSG
3878:000B    MOV    [010C],AX   .BUFO
3878:000E    MOV    CX,0008
LOOP:
3878:0011    LEA    AX,[0100]   .PARMBLK
3878:0015    INT    DC
3878:0017    INC    WORD [012F]   .PATHT
3878:001B    CMP    WORD [012F],000F   .PATHT
3878:0020    JNZ    0011   .LOOP
EXIT:
3878:0022    MOV    CX,0019
```

Looks fine. Trace the non-segment register values through 5 instruction steps.

```
#t5
PRINT:
------I--387B 0000 0000 0000 0060 0000 0000 0000 0002 MOV    [010E],AX   .BUFS
------I--387B 0000 0000 0000 0060 0000 0000 0000 0005 MOV    BX,AX.
------I--387B 387B 0000 0000 0060 0000 0000 0000 0007 LEA    AX,[0118]   .MSSG
------I--0118 387B 0000 0000 0060 0000 0000 0000 000B MOV    [010C],AX   .BUFO
------I--0118 387B 0000 0000 0060 0000 0000 0000 000E MOV    CX,0008
3878:000E
```

No problem here. Check 24 bytes (in word format) of PARMBLK (parameter block) data for the Write SVC:

```
dw.parmblk,+18
387B:0100  0000 0200 0000 0000 0001 0000 0118 387B         ...............{8
387B:0110  0017 0000 0000 0000                             ........
```

Looks good. Trace the program execution for several instruction steps:

```
#t
LOOP:
------I--0118 387B 0008 0000 0060 0000 0000 0000 0011 LEA    AX,[0100]   .PARMBLK
```

```
3878:0011
```

No problem

```
#t
------I--0100 387B 0008 0000 0060 0000 0000 0000 0015 INT    DC
3878:0015
```

The interrupt returned 17H, indicating that all 23 letters were successfully written to the screen. So far, so good.

```
#t
------I--0017 0000 0008 4892 0060 0000 0000 0000 001B CMP    WORD [012F],000F  .PATH
3878:001B
```

The last instruction incremented the variable PATHT. Check that PATHT has a value of 1 by displaying (in word format) 2 bytes of data, starting at location PATHT:

```
#dw.patht,+2
387B:012F  0001                                                  ..
```

Looks fine. Maybe the wrong jump instruction was used. Trace the execution one more step and see how the loop works:

```
#t
C-A-S-I--0017 0000 0008 4892 0060 0000 0000 0000 0020 JNZ    0011  .LOOP
3878:0020
```

No problems. Trace execution for 1 one more instruction step:

```
#t
LOOP:
C-A-S-I--0017 0000 0008 4892 0060 0000 0000 0000 0011 LEA    AX,[0100]  .PARMBLK
3878:0011
```

Display the current instruction list:

```
#l
LOOP:
3878:0011   LEA    AX,[0100]  .PARMBLK
3878:0015   INT    DC
3878:0017   INC    WORD [012F]  .PATHT
3878:001B   CMP    WORD [012F],000F  .PATHT
3878:0020   JNZ    0011  .LOOP
EXIT:
3878:0022   MOV    CX,0019
3878:0025   MOV    BX,DS
3878:0027   LEA    AX,[0104]  .SWI
```

```
3878:002B    INT     DC
3878:002D    ADD     [BX+SI],AL
3878:002F    ADD     [BX],CL
```

One more instruction and the program should interrupt to the Write SVC. Trace the execution of the interrupt instruction:

```
#t
C-A-S-I--0100 0000 0008 4892 0060 0000 0000 0000 0015 INT     DC
3878:0015
```

That's the problem! The BX register is not properly set up for the Write SVC call. The BX register contains a value of 0000, instead of 3878, which is the value of DS. As a result, the program fails to locate the parameter block, PARMBLK, at its actual address: 3878:0100.

Now that we have found the source of the error, exit SID-286:

```
#q
Debugged process aborted

#q
```

How did the location of the parameter block get lost? The BX register was written over by the interrupt return error code. Since no parameter block exists at 0000:0100, the interrupt to the Write SVC returns with an error code and nothing is written. This happens fourteen times before the program terminates.

To fix the problem copy TEST.A86 to a ratable disk. Then re-edit it, putting the value of the DS register (3878) in the BX register each time before calling the interrupt.

The re-edited source file should look like the following:

```
; test.a86:

        dseg
parmblk db      0       ;       parameter block for Write SVC
option  db      0       ;       zero out
flags   dw      200h    ;       flags to write to screen
swi     dw      0,0     ;       no software interrupts
fnum    dw      1,0     ;       logical file num of stdout
bufo    dw      0       ;       message buf address offset
bufs    dw      0       ;       message buffer address
```

```
bufsz   dw      23,0    ;       size in bytes of msg buf
offs    dw      0,0

mssg    db      'polly want a cracker',10,13,0

patht   dw      0
;
:
CODE    cseg

start:  mov     ax, ds              ; get value of ds register
print:  mov     bufs, ax            ; that is seg of mssg
        mov     bx, ax              ; also use for int 220 call
        lea     ax, mssg            ; get offset value
        mov     bufo, ax            ; write into param block
        mov     cx, 8               ; write SVC # is 8
loop:   lea     ax, parmblk         ; offset of param block
        mov     bx, ds              ; high word of param block
        int     220                 ; write to the screen
        inc     patht               ; loop control
        cmp     word ptr patht, 15
        jne     loop                ; print message out 15 times

exit:   mov     cx, 25              ; exit SVC # is 25
        mov     bx, ds              ; segment val
        lea     ax, swi             ; offset val
        int     220                 ; exit and return to the O/S
end
```

Recompile and relink TEST.A86 using RASM-86 and LINK 86. Since the file is on a separate disk, you must designate the correct pathname of the TEST.A86 file when entering the RASM-86 and LINK 86 commands.

Execute TEST.CMD again to confirm it is working correctly.

15.3 SID-286 Session #2

The following example illustrates how a compiled and linked CBASIC file is debugged using SID-286. The procedures used in this example are similar to those used when debugging the command files generated by other high-level programs.

When the executable file is run, nothing happens. The purpose of this SID-286 session is to demonstrate a variety of SID-286 commands in the process of isolating the cause of the program failure.

Begin SID session:

```
A>
A> sid

************************************************
SID-286          12/19/84             Version 1.0
Serial #XXXX-0000-654321         All Rights Reserved
Copyright 1985                 Digital Research Inc.
************************************************
```

Load the command file BANNER.286 and the symbol file BANNER.SYM:)

```
#e banner banner


Symbols
           address          length
code    3DD6:0000        00002050
data    3FDB:0000        00000B40
heap    0000:0000        00000000
```

Note that it was not necessary to enter the .286 or .SYM filetypes for either the command or the symbol file.

In order to become oriented to the program, list 12 disassembled instructions from current list address:

```
#l
3DD6:0000   JMP    0005
3DD6:0003   NOP
3DD6:0004   NOP
3DD6:0005   MOV    DX,0396
3DD6:0008   MOV    BX,092B
3DD6:000B   CALL   05E7   .?INIT
3DD6:000E   MOV    BX,05BA
3DD6:0011   CALL   06A7   .?ONER
3DD6:0014   MOV    DX,092C   .UCOMON
3DD6:0017   MOV    BX,015C
3DD6:001A   CALL   1C16   .?TSMM
3DD6:001D   XOR    AX,AX
```

Display instruction list starting at the first procedure, ?INIT.

```
#l.?init
?INIT:
3DD6:05E7   MOV    [0113],DX   .?SDAT
3DD6:05EB   MOV    [0115],BX   .?EDAT
3DD6:05EF   MOV    WORD [0970],FF01   .?COLUMN
3DD6:05F5   CLD
3DD6:05F6   POP    CX
3DD6:05F7   MOV    AX,DS
3DD6:05F9   MOV    ES,AX
3DD6:05FB   MOV    BX,[0104]   .?MEMRY
3DD6:05FF   ADD    BX,0100
3DD6:0603   JB     05E4
3DD6:0605   CMP    BX,[0006]
3DD6:0609   JNB    05E4
```

Find out what is in ?MEMRY by displaying 2 bytes of data starting at location ?MEMRY:

```
#d.?memry,+2
3FDB:0104  3F 0B                                         ?.
```

Find out what is in ?SDAT by displaying 4 bytes of data starting at location ?SDAT:

```
#d.?sdat,+4
3FDB:0113  00 00 00 00                                   ....
```

Find out what is in ?COLUMN by displaying (in word format) 16 bytes of data starting at location ?COLUMN:

```
#dw .?column,+10
3FDB:0970  0000 0000 0000 0000 0000 0000 0000 0000        ................
```

Examine present condition of CPU:

```
#x
           AX   BX   CX   DX   SP   BP   SI   DI   CS   DS   SS   ES   IP
------I--0000 0000 0000 0000 0060 0000 0000 0000 3DD6 3FDB 3887 3FDB 0000
JMP     0005
```

Display the instruction list from current address:

```
#l
3DD6:0000   JMP    0005
3DD6:0003   NOP
3DD6:0004   NOP
3DD6:0005   MOV    DX,0396
3DD6:0008   MOV    BX,092B
3DD6:000B   CALL   05E7   .?INIT
3DD6:000E   MOV    BX,05BA
3DD6:0011   CALL   06A7   .?ONER
3DD6:0014   MOV    DX,092C   .UCOMON
3DD6:0017   MOV    BX,015C
3DD6:001A   CALL   1C16   .?TSMM
3DD6:001D   XOR    AX,AX
```

Check if the program executes correctly before the ?INIT procedure by setting a breakpoint at ?INIT and executing the program from the beginning:

```
#g,.?init
3DD6:05E7     .?INIT
```

Everything seems OK. Take another look at ?INIT:

```
#l
?INIT:
3DD6:05E7   MOV    [0113],DX   .?SDAT
3DD6:05EB   MOV    [0115],BX   .?EDAT
3DD6:05EF   MOV    WORD [0970],FF01   .?COLUMN
3DD6:05F5   CLD
3DD6:05F6   POP    CX
3DD6:05F7   MOV    AX,DS
3DD6:05F9   MOV    ES,AX
3DD6:05FB   MOV    BX,[0104]   .?MEMRY
3DD6:05FF   ADD    BX,0100
3DD6:0603   JB     05E4
3DD6:0605   CMP    BX,[0006]
3DD6:0609   JNB    05E4
```

Trace execution for 2 instruction steps and display resulting CPU state:

```
#u2
------I--0000 092B 0000 0396 005E 0000 0000 0000 05EB MOV    [0115],BX  .?EDAT
3DD6:05EF
```

Find out what the contents of the DX and BX registers overwrites by displaying (in word format) 4 bytes of data starting at address 0113:

```
#dw113,+4
3FDB:0113  0396 092B                                          ..+.
```

Trace execution for 4 instruction steps and display resulting CPU state:

```
#u4
------I--0000 092B 0000 0396 005E 0000 0000 0000 05F5 CLD
3DD6:05F9
```

Examine the current condition of CPU:

```
#x
            AX   BX   CX   DX   SP   BP   SI   DI   CS   DS   SS   ES   IP
------I--3FDB 092B 000E 0396 0060 0000 0000 0000 3DD6 3FDB 408F 3FDB 05F9
MOV    ES,AX
```

Trace execution for 5 instruction steps, displaying current state of CPU after each trace:

```
#t5
------I--3FDB 092B 000E 0396 0060 0000 0000 0000 05FB MOV    BX,[0104]   .?MEMRY
------I--3FDB 0B3F 000E 0396 0060 0000 0000 0000 05FF ADD    BX,0100
-P----I--3FDB 0C3F 000E 0396 0060 0000 0000 0000 0603 JB     05E4
-P----I--3FDB 0C3F 000E 0396 0060 0000 0000 0000 0605 CMP    BX,[0006]
-P----I--3FDB 0C3F 000E 0396 0060 0000 0000 0000 0609 JNB    05E4
3DD6:0609
```

Display instruction list from current address:

```
#l
3DD6:0609   JNB    05E4
3DD6:060B   MOV    SS,AX
3DD6:060D   MOV    SP,BX
3DD6:060F   PUSH   CX
3DD6:0610   CALL   0699   .?REST
3DD6:0613   CMP    BYTE [0112],01
3DD6:0618   JNZ    068E
3DD6:061A   DEC    BYTE [0112]
3DD6:061E   MOV    [010C],BX   .?FSA
3DD6:0622   MOV    DI,[0106]   .?UCOM
3DD6:0626   SUB    AL,AL
3DD6:0628   MOV    CX,BX
```

Do we jump on JNB? Trace execution for 1 instruction step to find out:

```
#t
-P----I--3FDB 0C3F 000E 0396 0060 0000 0000 0000 05E4 JMP    07B6
3DD6:05E4
```

Yes, the JMP instruction executed as expected. Display instruction list:

```
#l
3DD6:05E4   JMP    07B6
?INIT:
3DD6:05E7   MOV    [0113],DX   .?SDAT
3DD6:05EB   MOV    [0115],BX   .?EDAT
3DD6:05EF   MOV    WORD [0970],FF01   .?COLUMN
3DD6:05F5   CLD
3DD6:05F6   POP    CX
3DD6:05F7   MOV    AX,DS
3DD6:05F9   MOV    ES,AX
3DD6:05FB   MOV    BX,[0104]   .?MEMRY
3DD6:05FF   ADD    BX,0100
```

```
3DD6:0603    JB      05E4
3DD6:0605    CMP     BX,[0006]
```

Determine what happens after the JMP 07B6 instruction by tracing execution for 1 instruction step:

```
-P----I--3FDB 0C3F 000E 0396 0060 0000 0000 0000 07B6 MOV    CX,4F4D
3DD6:07B6
```

Display instruction list:

```
#l
3DD6:07B6    MOV     CX,4F4D
3DD6:07B9    JMP     06D8   .?EROR
?RELS:
3DD6:07BC    DEC     BX
3DD6:07BD    DEC     BX
3DD6:07BE    MOV     CX,[BX]
3DD6:07C0    MOV     SI,[010E]
3DD6:07C4    CMP     SI,BX
3DD6:07C6    JB      07CB
3DD6:07C8    MOV     SI,011A   .?AVAIL
3DD6:07CB    MOV     [010E],SI
3DD6:07CF    MOV     SI,[SI]
3DD6:07D1    OR      SI,SI
```

Trace execution for 2 instruction steps and display resulting CPU state after jumping to the ?EROR procedure:

```
#u2
-P----I--3FDB 0C3F 4F4D 0396 0060 0000 0000 0000 07B ?1 ? ?1 ?9 JMP    06D8  .?EROR
3DD6:06D8
```

Display instruction list starting at the ?EROR procedure:

```
#l
?EROR:
3DD6:06D8    MOV     AX,[0966]
3DD6:06DB    MOV     [0968],AX
3DD6:06DE    MOV     AX,[096A]
3DD6:06E1    OR      AX,AX
3DD6:06E3    JZ      06EF
3DD6:06E5    MOV     [096C],CX
3DD6:06E9    MOV     SP,[010C]   .?FSA
3DD6:06ED    JMP     AX
3DD6:06EF    PUSH    CX
3DD6:06F0    MOV     CL,09
3DD6:06F2    MOV     DX,011E
```

```
3DD6:06F5    CALL    0694   .?BDOS
```

Begin execution at the beginning of the ?EROR procedure and set a breakpoint on the JZ instruction at address 06E3:

```
#g,6e3
3DD6:06E3
```

Does the jump occur ? Trace execution for 1 instruction step and find out:

```
#t
-P-Z--I--0000 0C3F 4F4D 0396 0060 0000 0000 0000 06EF PUSH   CX
3DD6:06EF
```

Yes, the jump occurred. Display the instruction list from current address:

```
#l
3DD6:06EF    PUSH    CX
3DD6:06F0    MOV     CL,09
3DD6:06F2    MOV     DX,011E
3DD6:06F5    CALL    0694   .?BDOS
3DD6:06F8    POP     DX
3DD6:06F9    PUSH    DX
3DD6:06FA    XCHG    DH,DL
3DD6:06FC    MOV     CL,02
3DD6:06FE    CALL    0694   .?BDOS
3DD6:0701    POP     DX
3DD6:0702    MOV     CL,023DD6:0704    CALL    0694    .?BDOS
```

How does the stack look at this point ? Display (in word format) the contents of the stack:

```
#dwss:sp
The stack is presently empty
```

How will the stack look after the PUSH CX instruction ? Trace execution for 1 instruction step .....

```
#t
-P-Z--I--0000 0C3F 4F4D 0396 005E 0000 0000 0000 06F0 MOV    CL,09
3DD6:06F0
```

.... and redisplay the contents of the stack:

```
#dwss:sp
408F:005E  4F4D                                                MO
```

The CX register's value of 4F4D is now on the stack. Begin execution and set a breakpoint on the POP DX instruction at address 06F8:

```
#g,6f8
3DD6:06F8
```

What is the current condition of the CPU ?

```
#x
                AX   BX   CX   DX   SP   BP   SI   DI   CS   DS   SS   ES   IP
-P-Z--I--0000 0C3F 4F09 011E 005E 0000 0000 0000 3DD6 3FDB 408F 3FDB 06F8
POP    DX
```

Check the stack again:

```
#dwss:sp
408F:005E  4F4D                                                     MO
```

Good, no change. Display instruction list from current address:

```
#l
3DD6:06F8    POP     DX
3DD6:06F9    PUSH    DX
3DD6:06FA    XCHG    DH,DL
3DD6:06FC    MOV     CL,02
3DD6:06FE    CALL    0694   .?BDOS
3DD6:0701    POP     DX
3DD6:0702    MOV     CL,02
3DD6:0704    CALL    0694   .?BDOS
3DD6:0707    MOV     DX,0127
3DD6:070A    MOV     CL,09
3DD6:070C    CALL    0694   .?BDOS
3DD6:070F    POP     AX
```

Now, trace what happens with the next two POP and PUSH instructions.

```
#t2
-P-Z--I--0000 0C3F 4F09 4F4D 0060 0000 0000 0000 06F9 PUSH   DX
-P-Z--I--0000 0C3F 4F09 4F4D 005E 0000 0000 0000 06FA XCHG   DH,DL
3DD6:06FA
```

Everything seems alright. Now begin execution and set a breakpoint on the POP DX instruction at address 0701:

```
#g,701
3DD6:0701
```

Display instruction list from current address:

```
#l
3DD6:0701    POP    DX
3DD6:0702    MOV    CL,02
3DD6:0704    CALL   0694   .?BDOS
3DD6:0707    MOV    DX,0127
3DD6:070A    MOV    CL,09
3DD6:070C    CALL   0694   .?BDOS
3DD6:070F    POP    AX
3DD6:0710    DEC    AX
3DD6:0711    DEC    AX
3DD6:0712    DEC    AX
3DD6:0713    PUSH   AX
3DD6:0714    XCHG   AL,AH
```

Check the stack again:

```
#dwss:sp
408F:005E  4F4D                                                    MO
```

Looks OK. Execute and set a breakpoint on the POP AX instruction at address 070F:

```
#g,70f
3DD6:070F
```

Display instruction list:

```
#l
3DD6:070F    POP    AX
3DD6:0710    DEC    AX
3DD6:0711    DEC    AX
3DD6:0712    DEC    AX
3DD6:0713    PUSH   AX
3DD6:0714    XCHG   AL,AH
3DD6:0716    CALL   0727
3DD6:0719    POP    AX
3DD6:071A    CALL   0727
3DD6:071D    MOV    CL,02
3DD6:071F    MOV    DL,48
3DD6:0721    CALL   0694   .?BDOS
```

Before executing the POP AX instruction, check the contents of the stack and see what value will be put into the AX register.

```
#dwss:sp
The stack is presently empty
```

Aha! There is nothing on the stack, the POP AX instruction makes the program fail. This is probably the result of a linker error.

Exit SID

```
#q
Debugged process aborted

#q
```

15.4 SID-286 Session #3

The following SID-286 session operates on a compiled and linked CBASIC program. The resulting command file will not load. Using SID-286, the executable format of the file is changed (patched) without recompiling and relinking the program.

Begin SID-286 session:

```
A> sid

*****************************************************
SID-286          12/19/84                Version 1.0
Serial #XXXX-0000-654321         All Rights Reserved
Copyright 1985                  Digital Research Inc.
*****************************************************
```

Read the file STRIP.286 into SID-286.

```
#r strip.286

  begin       end
33F5:0000   33F5:2C80
```

Display, in word format, the instruction list from the current address:

```
#dw
33F5:0000  7301 0002 7300 0002 0200 0049 0000 0049         .s...s....I...I.
33F5:0010  1000 0000 0000 0000 0000 0000 0000 0000         ................
33F5:0020  0000 0000 0000 0000 0000 0000 0000 0000         ................
33F5:0030  0000 0000 0000 0000 0000 0000 0000 0000         ................
33F5:0040  0000 0000 0000 0000 0000 0000 0000 0000         ................
33F5:0050  0000 0000 0000 0000 0000 0000 0000 0000         ................
33F5:0060  0000 0000 0000 0000 0000 0000 0000 0000         ................
33F5:0070  0000 0000 0000 0000 0000 0000 0000 0000         ................
33F5:0080  02E9 9000 BA90 0238 38BB E802 013F 24BB         ......8..8..?..$
33F5:0090  E801 01F9 77E8 8B1D BAD8 0238 16E8 8B1D         .....w....8.....
33F5:00A0  3816 8B02 3A1E E802 1CC1 0E75 5CBB E801         .8...:....u..\..
```

Set the number of bytes displayed by the D command to 32.

```
#-d20
```

Display 32 bytes of disassembled instructions, starting at offset 0000.

```
#d0
33F5:0000  01 73 02 00 00 73 02 00 00 02 49 00 00 00 49 00   .s...s....I...I.
33F5:0010  00 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

Use the S (Set) command to change the contents of memory, starting at offset 0010:

```
#s10
33F5:0010       00                     carriage return means no change

33F5:0011       10        0            change 10 to 0

33F5:0012       00        .            exit set environment
```

Confirm change by redisplaying 32 bytes of instructions, starting at offset 0000.

```
#d0
33F5:0000  01 73 02 00 00 73 02 00 00 02 49 00 00 00 49 00   .s...s....I...I.
33F5:0010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

Write the altered STRIP.286 file to disk under the filename STRIP1.286.

```
#w strip1.286
```

Exit SID-286

```
#q
```

End of Section 15

Example RASM-86 Source File

This example RASM-86 source files is provided to illustrate some of the characteristics unique to RASM-86.

Listing A-1. RASM-86 Sample for FlexOS 286

```
;       display the contents of an ascii file at the console

eof     equ     1ah
cdosi   equ     220
bsize   equ     80h

cseg

start:
        call    open

loop:   call    filbuf          ;refill buffer from file

        mov     bx,offset inbuff

        call    chkchr
        call    write
        jmps    loop
```

Listing A-1. (continued)

```
chkchr: mov     si,bptr
        cmp     byte ptr[bx+si],eof ;check next character in buffer
        jz      lastrec

        cmp     bptr,bsize

        je      nextb
        inc     bptr
        jmp     chkchr

nextb:  ret

filbuf: mov     cx,7                     ;read a record
        mov     ax,offset readpblk
        mov     bx,seg readpblk
        int     cdosi
        mov     bptr,0
        ret

open:   mov     cx,5                     ;open a file
        mov     ax,offset openpblk
        mov     bx,seg openpblk
        int     cdosi

        mov     lofnum,ax                ;save the file number
        mov     hifnum,bx                ;for the read

        ret

write:  mov     cx,8                     ;write to the screen
        mov     ax,offset wrtpblk
        mov     bx,seg wrtpblk
        int     cdosi
        ret

close:  mov     cx,6
        mov     ax,offset closepblk
        mov     bx,seg closepblk
        int     cdosi
        ret
```

Listing A-1. (continued)

```
lastrec:dec     bptr
        mov     dx,bptr
        mov     bfsize,dx
        call    write
        call    close


done:   mov     cx,25           ;reset function
        mov     ax,0
        mov     bx,0
        int     cdosi

        dseg

bptr            dw      bsize

wrtpblk         db      0               ;mode
                db      0               ;option
                dw      0               ;flags
                dw      0,0             ;swi
                dw      1,0             ;fnum
                dw      offset inbuff   ;buffer
                dw      seg inbuff
bfsize          dw      bsize           ;buffer size
                db      0
                dw      0
                dw      0,0             ;offset = 0
                dw      0,0             ;parm 7

openpblk        db      0               ;mode
                db      0               ;option
                dw      08
                dw      0,0             ;reserved
                dw      offset fname    ;address of
                dw      seg fname       ;file name
                dw      0
fname           db      'b:testfile.dat'
                dw      0
                db      0
                dw      0
```

Listing A-1. (continued)

```
readpblk        db      0                   ;mode
                db      0                   ;option
                dw      0000000100000001b
                dw      0,0                 ;swi
lofnum          dw      0                   ;fnum
hifnum          dw      0
                dw      offset inbuff       ;buffer
                dw      seg inbuff
                dw      128,0               ;buffer size
                dw      0,0                 ;offset = 0
                dw      0,0                 ;delimiters 7
inbuff          rs      128
                db      0

closepblk       db      0                   ;mode
                db      0                   ;option
                dw      0,0                 ;full close
                dw      word ptr lofnum     ;fnum
                dw      word ptr hifnum

sseg
                rs      200h

end
```

End of Appendix A

Creating Shared Runtime Libraries

B.1 Shareable Runtime Libraries

This appendix describes the procedures for creating and modifying shareable runtime libraries (SRTLs). SRTLs allow multiple users to share a single copy of library code at runtime. This makes it unnecessary for each user to store library code in his command file. When libraries are shared, only references to the library code are linked with the user's object files.

Before attempting to create or modify shareable runtime libraries, you should be familiar with the 80286 architecture, memory models, and calling conventions. You should also be familiar with conventions used when writing reentrant code.

You can write most shareable runtime library code in C, or most other high level languages. The only code that cannot be written in a high level language is the transfer vector code, which handles the calls from the user program to the SRTL routines. Transfer vector code should be written in 80286 assembly language.

See Section 7 in this manual for a description on how to link shareable runtime libraries with your object files.

B.2 SRTL Components

A SRTL consists of two types of files:

Shareable Runtime Library File	This linkable file contains the basic shareable object code created by LIB-86.
XSRTL Code File	This is an executable version of the SRTL created by LINK 86.

The remainder of this appendix describes how to create these two files.

B.3 Creating a SRTL

This section describes the steps for creating a SRTL. The steps listed below are described in detail in the following sections.

1. If you plan to compile the SRTL using a large memory model[2], you must modify the source of the library routines to use the proper calling conventions.
2. Create the transfer vectors.
3. Compile the source for the library routines and transfer vectors to get object modules.
4. Create the LIBATTR module.
5. Use LIB-86 to create the SRTL.
6. Use LINK 86 to create an executable SRTL (XSRTL).

B.3.1 Modifying the Source

You must modify the library source if the SRTL is to operate with the large memory model. If the large memory model is used, two modifications are required:

1. The formal parameter lists of the entry point routines in the SRTL must include three extra words of parameters. These parameters provide "placeholders" for the extra information inserted onto the stack by the transfer vector (see Section B.3.2).

[2]The type of memory model you use is dependent on your compiler. In its usage here, a large memory model refers to a memory model that allocates multiple code, data, and heap segments, as well as a separate segment for the stack.

2. Any intra-library calls to the entry point routines must have three words added to the actual parameter list, preceding the "real" parameters, so local calls emulate the stack activity of external calls through the transfer vectors.

3. If transfer vectors are used, intermediate names must be put in the library. This is also true for small memory model[3] SRTLs using a transfer vector at the beginning of the library in order to make all entry points constant(.

Using C conditional compilation statements, a single set of sources could be used for both shareable and nonshareable libraries, as shown below:

```
#ifdef NonShareable
        strcpy (to, from)
                char *to, *from;
#else
        strcpy (d1, d2, d3, to, from)
                WORD d1, d2, d3;
                char *to, *from;
#endif
   •
   •
   •
#ifdef NonShareable
        strcpy (source, target);
#else
        strcpy (0, 0, 0, source, target);
#endif
```

[3]In its usage here, a small memory model refers to a memory model that allocates a single segment for the program's data, heap, and stack.

B.3.2 Creating the Transfer Vectors

The transfer vector calling conventions described in this appendix are among many possible conventions for handling calls between user programs and SRTLs. The calling conventions you use depend on your application and your programming style. However, the transfer vector calling conventions described in this section have been tested and are recommended by Digital Research.

There are two types of transfer vectors: User and SRTL.

User Transfer Vector

If the user program uses a large memory model SRTL, calls to the SRTL routines must first pass through a transfer vector. This transfer vector, referred to as a User Transfer Vector, stores the value of the user program's DS register and loads the SRTL's DS register prior to entering the SRTL. Upon exiting the SRTL, the transfer vector restores the user program's DS register. The user transfer vector must reside in the user program's code space and must have a separate entry for each entry point into the SRTL.

See Section B.5 for more information on User Transfer Vectors.

SRTL Transfer Vector

If either memory model is being used, you can create an optional SRTL Transfer Vector that resides in the SRTL. This transfer vector is a collection of jumps that establish the SRTL entry points at fixed locations. By making an entry point into the SRTL a fixed location (constant virtual address), the user programs do not have to be relinked each time a change is made to the library.

There must be an entry in the SRTL Transfer Vector for each entry point into the SRTL.

To make SRTL entry points constant, any object modules containing the SRTL transfer vector must appear immediately after the LIBATTR module in the SRTL.

B.3.3 Creating the Object Modules

This step involves compiling the library source files to create library object files. Ensure that any compiler options required to generate the object files using the correct memory model are set.

B.3.4 Creating the LIBATTR Module

Each SRTL is associated with a specific set of attributes. These attributes include:

- the SRTL's name
- the SRTL's version number
- the location of the SRTL's data when the small memory model is used
- whether the SRTL is shared or not shared by default

The attributes of a SRTL are established by including a module with the name "LIBATTR" as the first module in the SRTL library file. The LIBATTR module must always be specified as the first module in the list.

The only contents of the LIBATTR module should be a single data segment which also has the name "LIBATTR". The contents of this segment must have the format indicated by the lib_id and lib_attr structures described in the following listing. The equivalent form in RASM-86 code appears in the example at the end of this appendix.

```
struct lib_id {
        char                li_name[8];
        unsigned short      li_ver_major;
        unsigned short      li_ver_minor;
        unsigned long       li_flags;
};
struct lib_attr {
        lib_id              la_id;
        unsigned short      la_data_offset;
        char                la_share_attr;
};
#define la_s_shared         0x1
```

LIB_ID Structure

The lib_id structure defines the fields used to identify the SRTL including its name, version numbers and some flags. The lib_id fields are:

li_name	This name is the physical name of the XSRTL to be specified by the users of the SRTL. If this field is nonblank, the library is a SRTL, otherwise the library is a normal library. This field must be exactly 8 bytes long, including trailing blanks.
li_ver_major	This is the major version number. It should be updated when there are major changes to a library making it incompatible with previous versions. When a user of this SRTL is loaded, the loader verifies that the major version number requested by the user is identical to that of the XSRTL.
li_ver_minor	This is the minor version number. It is updated when changes to the library do not create incompatibilities with previous versions. The minor version number specified in a user program does not have to exactly match that specified in the XSRTL.

li_flags	This field contains flags that distinguish different variants of the library from one another. Currently, only the lower order 6 bits have been assigned values. The following flags are available:
li_f_cmd (0x0)	This flag informs LINK 86 that the code file should have the filetype CMD. If this, or any other value is specified to a version of LINK 86 that cannot generate that style of code file, an error is generated. For example, the LINK 86 that generates CMD and 286 files cannot generate an EXE file and vice versa.
li_f_286 (0x1)	This flag informs LINK 86 the suffix of the code file should be 286.
li_f_exe (0x2)	This flag informs LINK 86 the suffix of the code file should be EXE.
(0x3 through 0xF)	These values are unassigned and cause an error if specified in a LIBATTR module.
li_f_dup_main (0x10)	This value informs LINK86 that duplicate definitions of the symbol "main" are permitted and should not generate errors.
li_f_ds_stack (0x20)	This value informs LINK86 the stack appears at the low end of the data segment. When this bit is set, the value of the data_offset field represents the size of this stack. If multiple LIBATTR modules are found, the smallest data_offset value is used. Data is allocated above the stack and the public variable ?STACK, if present, is initialized to the size of the stack. The variable ?STACK is assumed to be allocated a word (2 bytes).

LIB_ATTR Structure

The lib_attr structure defines the fields determining the offset memory location of the SRTL, as well as whether or not the SRTL is shareable. The lib_attr fields are:

la_id	This field specifies the library id.
la_data_offset	When the small memory model is used, the value of this field gives the fixed address where the SRTL data MUST appear. This value is also used to prevent LINK 86 from allocating user data at the same location as the SRTL data. When the large memory model is used, this field must have the value 0xFFFF.
la_share_attr	This field tells LINK 86 whether the library by default is shared (value 1) or not shared (value 0), so you do not have to specify the SHARED option every time the library is used. You can override this default as specified in Section 7.5.

When linking a user program, LINK 86 determines that a library is a SRTL by checking the name of the first module in a library. If the first module has the name "LIBATTR", LINK 86 examines the contents of the LIBATTR segment to determine the attributes of the library. Depending on the value of the la_share_attr field and input options, the library is treated either as a SRTL or as a regular library.

Before creating a LIBATTR module, you must determine the filename of the executable shareable runtime library (see Section B.3.6 below) and the data offset if the small memory model is being used. You must also ensure the sizes and offsets of data items correspond to the fields in the LIBATTR definition given above. The LIBATTR segment must contain exactly 19 bytes.

When linking your file(s) with a SRTL, do not include the LIBATTR module in either your user code file or the XSRTL code file. The LIBATTR module is used to define the attributes of the SRTL and should not be treated as a normal module. The LIBATTR module is included in the code file only when the SRTL is linked as a normal, nonshared library (without specifying the SEARCH option).

B.3.5 Creating the SRTL

You create a SRTL using LIB-86. When entering the command line, you must specify the LIBATTR object file as the first file in the library. If the optional SRTL Transfer Vector is used, it must be the second object file in the library.

For example, to create a SRTL named SRTLLIB1 using the LIBATTR file, LIBATTR.OBJ, and the object files: LIB1.OBJ and LIB2.OBJ, enter the command:

```
A>lib86 srtllibl = libattr.obj, libl.obj, lib2.obj
```

Assuming all calling conventions are correct, you can modify an existing nonshareable runtime library to create a shareable library by including a LIBATTR file in the LIB86 command. For example, to make the nonshareable library OLDLIB.L86 into the shareable library NEWLIB using a LIBATTR file named SRTLOLD.OBJ, enter:

```
A>lib86 newlib = srtlold.obj, oldlib.186
```

B.3.6 Creating the XSRTL

A library file cannot be loaded by the operating system and executed. Therefore, an executable version of the library file must be available. This version of a SRTL is called the XSRTL (eXecutable Shared Run Time Library).

Once you create a SRTL, you can then create the XSRTL by linking the SRTL, as shown in the following example. LINK 86 automatically recognizes the library is a SRTL by the presence of the LIBATTR module.

The command:

```
A>link86 doall.186
```

creates a file DOALL.CMD, containing the executable version of the library file DOALL.L86. With the exception of the LIBATTR module, LINK 86 processes all code and data from all modules in the library, in the order they appear in the library. During the link process, the addresses of the data are resolved, but the data is not included in the XSRTL, which contains only the SRTL code. This convention is necessary to ensure that the references in a SRTL user program match the XSRTL, since both are linked separately.

The XSRTL has no main program, but this is not a problem, because LINK 86 knows it is an XSRTL. Also, the XSRTL should have no unresolved external references, as there would be no way to resolve them separately to each of multiple, simultaneous user programs.

Note that the value of the data_offset attribute from the LIBATTR module determines whether the XSRTL uses the small or large memory model. If the attribute value is 0xFFFF, the large memory model is assumed. Any other value indicates the offset of the SRTL data within the data segment, which means the XSRTL uses the small memory model.

B.4 Small & Medium Model SRTLs

When creating a SRTL for use with the Small or Medium Memory Model, you must decide beforehand where the SRTL data appears and code this as the value of the data_offset attribute in the LIBATTR module. The location DS:0 is not acceptable, because the stack overflow detection requires the stack to be at the bottom of the data segment. On the other hand, putting the SRTL data at the top of the data segment does not efficiently utilize memory. The layout of the data segment for a small memory model program should resemble that specified in Figure B-1.

```
          +----------------+
          |       .        |
          |       .        |
          |       .        |
          |      Heap      |
          +----------------+
          |   Local Data   |
          +----------------+
          |   SRTL Data    |
          +----------------+
          |     Stack      |
          |       .        |
          |       .        |
          |       .        |
DS:0      +----------------+
```

Figure B-1. Small Memory Model Data Segment

If there is sufficient room between the stack and the SRTL data, LINK 86 may put the local data below the SRTL data.

If the size of the stack exceeds the starting address of the SRTL data, LINK 86 prints an error message and terminates the link.

B.4.1 Calling Conventions

A file compiled using the small memory model must be linked with a small model SRTL so that both the user file and the SRTL can agree on the location of the SRTL data. This restriction fixes the calling sequence as illustrated in Figure B-2 below.

```
             U S E R                                     S R T L

Calling Sequence   Transfer Vector       Transfer Vector  Prologue/Epilogue


CALLF    X ---------------------------->   X: JMP X' -----> X': PUSH BP
  .        <--------------+                                     MOV  BP,SP
  .                       |                                      .
  .                       |                                      .
                          |                                      .
                          |                                     POP  BP
                          +<------------  <-------------------- RETF
```

Figure B-2. Small Memory Model Calling Sequence

The calling sequence shown in Figure B-2 uses a SRTL Transfer Vector. If no SRTL Transfer Vector is used, the reference to the SRTL routine would go directly to the PUSH BP instruction, rather than the JMP instruction in the transfer vector.

Note that the SRTL code sequence in Figure B-2 can have one or more code segments and, therefore, can be used with the standard medium model consisting of multiple code segments and a single data segment.

B.5 Large Model SRTLs

To have reentrant code, the SRTL data must belong to the user file, rather than the SRTL. When using the large memory model, the user file must inform the SRTL of the location of its data. As a consequence, user calls to large memory model SRTL routines must pass through a User Transfer Vector, as described in Section B.3.2.

B.5.1 Calling Conventions

The SRTL determines the location of the user file's data by mapping all references to the SRTL through a transfer vector local to the user. The transfer vector pushes the user's DS and loads the SRTL DS before calling the SRTL routine, then pops the user's DS after control returns from the library. This code is shown in Figure B-3.

```
              U S E R                                    S R T L

Calling Sequence   Transfer Vector        Transfer Vector   Prologue/Epilogue

CALLF X ----------> X:  PUSH DS
U:      <-----+         MOV  AX,SRTLDS
              |         MOV  DS,AX
              |         CALLF X' ---->    X': JMP X" ----> X": PUSH BP
              |     Y:  POP  DS <-----   <--------+              MOV  BP,SP
              +<------- RETF                      |               .
                                                  |               .
                                                  |               .
                                                  |              POP  BP
                                                  +<-----------  RETF
```

Figure B-3. Large Memory Model Calling Sequence

Note that this calling sequence assumes the creator of the library coded a transfer vector containing the entry points to the library. If this were not the case, the CALLF in the user's transfer vector would directly reference the PUSH BP instruction in the SRTL routine. The transfer vector on the SRTL side, though not necessary, is recommended.

B.6 SRTL Restrictions

When creating SRTLs, keep the following restrictions in mind:

- XSRTLs must not contain overlays.
- A SRTL can contain a maximum of 255 object modules. The LIBATTR module does not count in this figure unless the SRTL is linked as a nonshared library and the search option is not set.

B.7 Example SRTL

The following C program tests a small memory model SRTL by calling the SRTL subroutine LIST_PRINT twice.

```
/* main program */
main()
{
        list_print("In Main 1st Time");
        list_print("In Main 2nd Time");
}
```

Below is the LIBATTR (Library Attribute) module for testing LINK 86. The LIBATTR module defines the attributes used by LINK 86 when linking XSRTLs (eXecutable SRTLs) and when linking to other SRTLs. The LIBATTR module must appear as the first module in a SRTL.

```
        NAME    'LIBATTR'
LIBATTR DSEG PUBLIC

SNAME   DB      'SM1SRTL '          ; The XSRTL's physical file name
                                    ; MUST BE 8 BYTES LONG!!!

VERMAJ  DW      9876H               ; Major version number
VERMIN  DW      5432H               ; Minor version number

FLAGS   DB      00h,00h,00h,00h     ; Flags must be zero for now

DATAOFF DW      0200H               ; The offset of small memory model
                                    ; Must be FFFF for large memory model

SHARED  DB      1                   ; 1 = shared by default
                                    ; 0 = not shared by default

        END
```

The following are SRTL routines that test small memory model SRTLs. The LIST_PRINT routine loads the string into the variable I and calls the CONSOLE_PRINT, which prints the string on the console.

```
list_print (i)
char *i;
{
        console_print("In the SRTL");
        console_print (i);
}

console_print (i)
char *i;
{
        printf ("%s",i);
}
```

End of Appendix B

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 C-1. Mnemonic Differences

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

RASM-86 also uses a different method than Intel for specifying the size of memory operands for 8087 instructions. Intel associates the size with the operand, RASM-86 places it in the instruction. The following table shows the differences:

Table C-2. Memory Operands for 8087 Instruction

RASM-86	Intel	
FLD32	FLD	SYM 32
FLD64	FLD	SYM 64
FLD80	FLD	SYM 80
FST32	FST	
FST64	FST	
FST80	FST	
FILD16	FILD	
FILD32	FILD	
FILD64	FILD	
FIST16	FIST	
FIST32	FIST	
FIST64	FIST	

End of Appendix C

Reserved Words

Table D-1. Reserved Words

Predefined Numbers				
BYTE	WORD	DWORD		
Operators				
AND	LAST	MOD	OR	SHR
EQ	LE	NE	PTR	TYPE
GE	LENGTH	NOT	SEG	XOR
GT	LT	OFFSET	SHL	
Assembler Directives				
AUTO8087	CODEMACRO	ELSE	GROUP	NOIFLIST
RD	ENDIF	END	HARD8087	NOLIST
RS	DB	ENDIF	IF	ORG
RW	DD	ENDM	IFLIST	PAGESIZE
SIMFORM	DSEG	EQU	INCLUDE	PAGEWIDTH
DW	ESEG	LIST	PUBLIC	SSEG
EJECT	EXTRN	NAME	RB	TITLE
Code-macro Directives				
DB	DD	MODRM	RELB	SEGFIX
DBIT	DW	NOSEGFIX	RELW	
8086 Registers				
AH	BL	CL	DI	ES
AL	BP	CS	DL	SI
AX	BX	CX	DS	SP
BH	CH	DH	DX	SS

Table D-1. (continued)

8087 Registers

ST	ST0	ST1	ST2	ST3
	ST4	ST5	ST6	ST7

Default Segment Names

CODE	DATA	EXTRA	STACK

Segment Descriptors

BYTE	LOCAL	PARA	STACK
COMMON	PAGE	PUBLIC	WORD

External Descriptors

ABS	DWORD	NEAR
BYTE	FAR	WORD

Instruction Mnemonics - See Section 13

End of Appendix D

Code-Macro Definition Syntax

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

<name> ::= IDENTIFIER

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

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

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

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

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

<single$range> ::= REGISTER | NUMBERB

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

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

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

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

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

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

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

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

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

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

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

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

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

<form$name> ::= IDENTIFIER



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

End of Appendix E

RASM-86 Error Messages

RASM-86 displays two kinds of error messages:

- nonrecoverable errors
- diagnostics

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

Table F-1. RASM-86 Non-recoverable Errors

Message	Cause
NO FILE	RASM-86 cannot find the indicated source or INCLUDE file on the indicated drive.
DISK FULL	There is not enough disk space for the output files. You should either erase some unnecessary files or get another disk with more room and run RASM-86 again.
DIRECTORY FULL	There is not enough directory space for the output files. You should either erase some unnecessary files or get another disk with more directory space and run RASM-86 again.

Table F-1. (continued)

Message	Cause
DISK READ ERROR	RASM-86 cannot properly read a source or INCLUDE file. This is usually the result of an unexpected end-of-file. Correct the problem in your source file.
CANNOT CLOSE	RASM-86 cannot close an output file. You should take appropriate action after checking to see if the correct disk is in the drive and the disk is not write-protected.
SYMBOL TABLE OVERFLOW	There is not enough memory for the Symbol Table. Either reduce the length or number of symbols, or reassemble on a system with more memory 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	Cause
ERROR NO: 0	**ILLEGAL FIRST ITEM** The first item on a source line is not a valid identifier, directive, or mnemonic. For example, `1234H`
ERROR NO: 1	**MISSING PSEUDO INSTRUCTION** The first item on a source line is a valid identifier, and the second item is not a valid directive that can be preceded by an identifier. For example, `THIS IS A MISTAKE`
ERROR NO: 2	**ILLEGAL PSEUDO INSTRUCTION** Either a required identifier in front of a pseudo instruction is missing, or an identifier appears before a pseudo instruction that does not allow an identifier.

ERROR NO: 3	**DOUBLE DEFINED VARIABLE**
	An identifier used as the name of a variable is used elsewhere in the program as the name of a variable or label. For example,
	`X        DB   5`
	`         . . .`
	`X        DB   123H`

Table F-2. (continued)

Error Message	Cause

ERROR NO: 4	**DOUBLE DEFINED LABEL**
	An identifier used as a label is used elsewhere in the program as a label or variable name. For example,
	`LAB3:  MOV   BX,5`
	`        . . .`
	`LAB3:  CALL  MOVE`

ERROR NO: 5	**UNDEFINED INSTRUCTION** The item following a label on a source line is not a valid instruction. For example, `DONE:  BAD   INSTR`

ERROR NO: 6	**GARBAGE AT END OF LINE - IGNORED**
	Additional items were encountered on a line when RASM-86 was expecting an end of line. For example,
	`NOLIST 4`
	`MOV    AX,4   RET`

Table F-2. (continued)

Error Message	Cause

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+1`

ERROR NO: 9	**MISSING INSTRUCTION** A prefix on a source line is not followed by an instruction. For example, `REPNZ`

ERROR NO: 10	**UNDEFINED ELEMENT OF EXPRESSION**
	An identifier used as an operand is not defined or has been illegally forward referenced. For example,
	`        JMP     X`
	`A       EQU     B`
	`B       EQU     5`
	`        MOV     AL,B`

Table F-2. (continued)

Error Message	Cause

ERROR NO: 11	**ILLEGAL PSEUDO OPERAND**
	The operand in a directive is invalid. For example,
	`X EQU 0AGH`
	`TITLE UNQUOTED STRING`

ERROR NO: 12	**NESTED IF ILLEGAL - IF IGNORED** The maximum nesting level for IF statements has been exceeded.
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.

Table F-2. (continued)

Error Message	Cause
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 * )`

Table F-2. (continued)

Error Message	Cause
ERROR NO: 21	**MISSING TYPE INFORMATION IN OPERAND(S)** Neither instruction operand contains sufficient type information. For example, `MOV     [BX],10`
ERROR NO: 22	**LABEL OUT OF RANGE** The label referred to in a call, jump, or loop instruction is out of range. The label can be defined in a segment other than the segment containing the instruction. In the case of short instructions (JMPS, conditional jumps, and loops), the label is more than 128 bytes from the location of the following instruction.

ERROR NO: 23	**MISSING SEGMENT INFORMATION IN OPERAND**
	The operand in a CALLF or JMPF instruction (or an expression in a DD directive) does not contain segment information. The required segment information can be supplied by including a numeric field in the segment directive as shown:
	`CSEG   1000H`
	`X:`
	`. . .`
	`JMPF   X`
	`DD     X`

Table F-2. (continued)

Error Message	Cause
ERROR NO: 24	**ERROR IN CODEMACRO BUILDING** Either a code-macro contains invalid statements, or a code-macro directive was encountered outside a code-macro.
ERROR NO: 25	**NO MATCHING IF FOR ELSE** An ELSE statement was encountered without a matching IF statement.
ERROR NO: 26	**NO MATCHING ENDIF FOR IF** An IF statement was encountered without a matching ENDIF statement.
ERROR NO: 27	**"HARD8087" USED AFTER FLOATING INSTRUCTION** The HARD8087 directive cannot be specified after a floating point instruction.
ERROR NO: 28	**ATTEMPT TO USE 186/286 INSTRUCTIONS WITHOUT SWITCH** 80186 or 80286 instructions were encountered and the corresponding RASM-86 run-time parameter (186 or 286) was not specified on the RASM-86 command line.
ERROR NO: 29	**Command included not used in source file** The command defined in the file included, via the INCLUDE command, in the RASM-86 source file is not used by the source file.

End of Appendix F

XREF-86 Error Messages

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

Table G-1. XREF-86 Error Messages

Error Message	Meaning
CANNOT CLOSE	XREF-86 cannot close an output file. You should take appropriate action after checking to see if the correct disk is in the drive and the disk is not write-protected.
DIRECTORY FULL	There is not enough directory space for the output files, You should either erase some unnecessary files or get another disk with more directory space and run XREF-86 again.
DISK FULL	There is not enough disk space for the output files. You should either erase some unnecessary files or get another disk with more room and run XREF-86 again.
NO FILE	XREF-86 cannot find the indicated file on the indicated drive.

Table G-1. (continued)

Error Message	Meaning
SYMBOL FILE ERROR	XREF-86 issues this message when it reads an invalid SYM file. Specifically, a line in the SYM file not terminated with a carriage return line-feed causes this error message.
SYMBOL TABLE OVERFLOW	XREF-86 ran out of Symbol Table space. Either reduce the number or length of symbols in the program, or rerun on a system with more memory.

End of 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 listed below.

Table H-1. LINK 86 Error Messages

Message	Meaning
8087 IN OVERLAY, NOT IN ROOT	The 8087 emulator, if used, must be referenced in the root if it is to be referenced in an overlay.
8087 SWITCH OCCURRED AFTER FIRST FILENAME	The HARD8087, AUTO8087, and SIM8087 switches must not appear after the first object file listed on the command line.
8087 TABLE OVERFLOW	The 8087 fixup table needed with the AUTO8087 or SIM8087 options can have a maximum size of 64K.
ALIGN TYPE NOT IMPLEMENTED	The object file contains a segment align type not implemented in LINK 86.
CANNOT CLOSE	LINK 86 cannot close an output file. Check to see if the correct disk is in the drive and the disk is not write-protected or full.
CLASS NOT FOUND	The class name specified in the command line does not appear in any of the files linked.

Table H-1. (continued)

Message	Meaning
COMBINE TYPE NOT IMPLEMENTED	The object file contains a segment align type not implemented in LINK 86.
COMMAND TOO LONG	The total length of input to LINK 86, including the input file, cannot exceed 2048 characters.
DIRECTORY FULL	There is not enough directory space for the output files. You should either erase some unnecessary files or get another disk with more directory space and run LINK 86 again.
DISK READ ERROR	LINK 86 cannot properly read a source or object file. This is usually the result of an unexpected end-of-file character. Correct the problem in your source file.
DISK WRITE ERROR	A file cannot be written properly; the disk is probably full.
ERROR IN LIBATTR MODULE	The LIBATTR module does not conform to established requirements. Fix the LIBATTR module and rebuild the library in question.
FIXUP TYPE NOT IMPLEMENTED	The object file uses a fixup type not implemented in LINK 86. Make sure the object file has not been corrupted.
GROUP NOT FOUND	The group name specified in the command line does not appear in any of the files linked.

Table H-1. (continued)

Message	Meaning
GROUP OVER 64k	The group listed must be made smaller than 64k before relinking. Either delete segments from the group, split it up into 2 or more groups or do not use groups.
GROUP TYPE NOT IMPLEMENTED	LINK 86 only supports segments as elements of a group.
INVALID LIBRARY-REQUESTED SUFFIX	The command file suffix requested by a library is not supported. Verify that the correct library is being used.
LINK-86 ERROR 1	This error indicates an inconsistency in the LINK 86 internal tables, and should never be emitted.
MULTIPLE DEFINITION	The indicated symbol is defined as PUBLIC in more than one module. Correct the problem in the source file, and try again.
MORE THAN ONE MAIN PROGRAM	A program linked by LINK 86 may have at most one main program.
NO FILE	LINK 86 cannot find the indicated source or object file on the indicated drive.
OBJECT FILE ERROR	LINK 86 detected an error in the object file. This is caused by a translator error or by a bad disk file. Try regenerating the file.

Table H-1. (continued)

Message	Meaning
RECORD TYPE NOT IMPLEMENTED	The object file contains a record type not implemented in LINK 86. Make sure the object file has not been corrupted by regenerating it and linking again.
SEGMENT OVER 64k	The segment listed after the error message has a total length greater than 64k bytes. Make the segment smaller, or do not combine it with other PUBLIC segments of the same name.
STACK COLLIDES WITH SRTL DATA	The base address of SRTL data does not allow enough room for the requested amount of stack space. Change the base of the SRTL data in the LIBATTR module or request less stack.
SRTL DATA OVERLAP	The data from 2 SRTLs overlap. Change the base address in the LIBATTR module of one of the SRTLs.
SRTL CANNOT CONTAIN 8087 FIXUPS	A SRTL cannot use the 8087 emulator as currently implemented.
SEGMENT CLASS ERROR	The class of a segment must be CODE, DATA, STACK, EXTRA, X1, X2, X3, or X4.
SEGMENT ATTRIBUTE ERROR	The Combine type of the indicated segment is not the same as the type of the segment in a previously linked file. Regenerate the object file after changing the segment attributes as needed.

Table H-1. (continued)

Message	Meaning
SEGMENT COMBINATION ERROR	An attempt is made to combine segments that cannot be combined, such as LOCAL segments. Change the segment attributes and relink.
SEGMENT NOT FOUND	The segment name specified in the command line does not appear in any of the files linked.
SYMBOL TABLE OVERFLOW	LINK 86 ran out of Symbol Table space. Either reduce the number or length of symbols in the program, or relink on a system with more memory.
SYNTAX ERROR	LINK 86 detected a syntax error in the command line; the error is probably an improper filename or an invalid command option. LINK 86 echoes the command line up to the point where it found the error. Retype the command line or edit the INP file.
TARGET OUT OF RANGE	The target of a fixup cannot be reached from the location of the fixup.
TOO MANY MODULES IN LIBRARY	The library contains more modules than LINK 86 can handle. Split the library up into 2 or more libraries and relink.
TOO MANY MODULES LINKED FROM LIBRARY	A library may supply a maximum of 256 modules during 1 execution of LINK 86. Split the library up into 2 or more smaller libraries.

Table H-1. (continued)

Message	Meaning
UNDEFINED SYMBOLS	The symbols following this message are referenced but not defined in any of the modules being linked.
VERSION 2 REQUIRED	LINK 86 needs a version 2 or later file system because its uses random disk I/O functions.
XSRTL MUST BE LINKED BY ITSELF	When linking an XSRTL, no other files may be linked at the same time.
XSRTLs INCOMPATIBLE WITH OVERLAYS	An XSRTL can not use overlays.

End of Appendix H

Overlay Manager Run-Time Errors

At run-time, the Overlay Manager can display certain error messages. These messages and a brief explanation of their cause are shown in Table I-1.

Table I-1. Overlay Manager Error Messages

Message	Meaning
OVERLAY ERROR, NO FILE d:filename.OVR	The Overlay Manager cannot find the indicated file.
OVERLAY ERROR, DRIVE d:filename.OVR	An invalid drive code was passed as a parameter to ?ovlay.
OVERLAY ERROR, NESTING d:filename.OVR	Loading the indicated overlay would exceed the maximum nesting depth.
OVERLAY ERROR, READ d:filename.OVR	Disk read error during overlay load, probably caused by premature EOF.

End of Appendix I

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 J-1. LIB-286 Error Messages

Message	Meaning
CANNOT CLOSE	LIB-86 cannot close an output file. You should take appropriate action after checking to see if the correct disk is in the drive and the disk is not write-protected.
DIRECTORY FULL	There is not enough directory space for the output files. You should either erase some unnecessary files or get another disk with more directory space and run LIB-86 again.
DISK FULL	There is not enough disk space for the output files. You should either erase some unnecessary files or get another disk with more room and run LIB-86 again.

Table J-1. (continued)

Message	Meaning
DISK READ ERROR	LIB-86 cannot properly read a source or object file. This is usually the result of an unexpected end-of-file. Correct the problem in your source file.
INVALID COMMAND OPTION	LIB-86 encountered an unrecognized option in the command line. Retype the command line or edit the INP file.
MODULE NOT FOUND	The indicated module name, which appeared in a REPLACE, SELECT, or DELETE switch, cannot be found. Retype the command line or edit the INP file.
MULTIPLE DEFINITION	The indicated symbol is defined as PUBLIC in more than one module. Correct the problem in the source file, and try again.
NO FILE	LIB-86 cannot find the indicated file.
OBJECT FILE ERROR	LIB-86 detected an error in the object file. This is caused by a translator error or a bad disk file. Try regenerating the file.

Table J-1. (continued)

Message	Meaning
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 requires a version 2 file system or later.

End of Appendix J

SID-286 Error Messages

Most of the error messages generated by SID-286 result from the inability of SID-286 to access an important table or execute a necessary SVC function call. Thus, many of the SID-286 error messages indicate that the interface to the operating system is not functioning correctly.

The following are the most common error messages generated by SID-286.

Table K-1. SID-86 Error Messages

Message	Meaning
Bad Hex Value	Bad format in the symbol table. This is usually caused by an invalid offset address or the use of discontinuous variable names. Only those symbols located before this error message are read and are accessible for symbolic referencing. You can use the E command to re-execute the program. All of the symbols may be read successfully on the second try. Otherwise, you can use the H command to find out which symbols have been read into the SYM file.
File Not Found	The specified file was not found on the specified pathname.

Table K-1. (continued)

Message	Meaning
Default File Size of 512 Bytes Assumed	SID-286 was unable to access the Disk File Table to find out the size of the file specified by an R command. As a result, only the first 512 bytes of that file were read in.
Error: Cannot Form Buffer	The Malloc SVC function is not working correctly within the Operating System and SID-286 is unable to form the buffer it needs in order to execute that command. As a result, the command is aborted.
Error: Cannot Read the File	The Read SVC function returns an error when SID-286 attempts to read the file specified by the R command. As a result, the R command is aborted. Exit SID-286 and verify that the file you specified was not damaged prior to entering SID-286.
Error: Inadequate Amount of Memory	Your system does not have enough memory to execute the command.
File Close Error	SID-286 was unable to successfully close the specified file.
Create Error	The command is aborted due to a failure of the Create SVC to create a file on the specified directory.

Table K-1. (continued)

Message	Meaning
Unsuccessful Register Write	SID-286 is unable to write the specified changes to the registers belonging to a process. The specified changes are not implemented.
Unsuccessful Register Read	SID-286 was unable to read the registers belonging to a process. Any register values displayed will probably be wrong.

End of Appendix K

Additional FlexOS Utilities

This appendix describes a set of special-purpose programming utilities and special utility options that are specific to FlexOS 286.

L.1 HSET

Forms

```
HSET -1 filespec.286
HSET -9 filespec.286
```

Explanation

The HSET command sets the code group descriptor in the 286 file header as either 09 (shared code) or 01 (non-shared code). The filespec must be a .286 (executable) type file.

L.2 IOMF

Form

```
IOMF -option filenam1[,filenam2,...filenamN]
```

Explanation

IOMF (Intel Object Module Format) displays, in human-readable form, files of Intel's 8086 "MCS-86" Object Module format. This format is described in the Intel Technical Specification "MCS-86 Relocatable Object Module Formats," Version 4.0, 15 January 1981.

If the file extension is omitted and the file filename.OBJ does not exit, IOMF looks for the file filename.L86.

Table L-1. IOMF Options

Option	Description
-nRECNUM	Start dumping after record number RECNUM.
-v	Displays in hexadecimal the file offset (byte number) where each record begins.
-h	Display only the header records, which are used to determine the locations of modules in a library.

L.3 OBJERR

Form OBJERR n

Explanation OBJERR prints an explanation of the object file error messages given by the linker. Valid values for n are 0-29.

L.4 POSTLINK

Form POSTLINK filename.286

Explanation POSTLINK does the fixups on a .286-type executable file and builds an LDT (Local Descriptor Table). Using POSTLINK on a file has two benefits:

- It reduces load time.
- It increases the number of local descriptors permitted.

POSTLINK modifies the source file and is not reversible.

L.5 PSN

Form PSN filespec.SYM filespec.PSN

Explanation PSN is an executable command of type .CMD, designed to run under FlexOS 286. PSN takes a symbol table file (.SYM) and outputs a numerically sorted listing.

End of Appendix L

Index