Microsoft®
Macro Assembler

User's Guide

for the MS™-DOS Operating System

Microsoft Corporation

Document Number: 8450L-300-00

Part Number: 016-014-009

Contents

Chapter 1
Introduction

1.1 Overview

The Microsoft Macro Assembler User's Guide explains how to create and debug assembly language programs using the Microsoft Macro Assembler, MASM, and the other utilities in the Macro Assembler package.

The Macro Assembler package consists of the following programs and files:

MASM.EXE	Microsoft Macro Assembler
LINK.EXE	Microsoft Link Utility
SYMDEB.EXE	Microsoft Symbolic Debugger
MAPSYM.EXE	Microsoft Symbol Map Utility
CREF.EXE	Microsoft Cross Reference Utility
LIB.EXE	Microsoft Library Manager
MAKE.EXE	Microsoft Program Maintainer
README.DOC	Additional information about MASM and the other utilities

The function of each program and an explanation of how to invoke and operate the programs is given in the remaining chapters of this guide.

The following sections explain what you need to create assembly language programs, what steps you need to take to create these programs, and what document conventions you will find when reading this guide.

1.2 What You Need

To make an assembly language program, you need a text editor and you need to know the correct syntax and format of assembly language source files. You also need to know the function and operation of the instructions in the instruction sets of the 8086/186/286 family of microprocessors.

The Microsoft Macro Assembler creates programs that can be executed under the 8086/186/286 family of microprocessors. It provides a logical program syntax that is ideally suited for the segmented architecture of the 8086. This syntax is fully explained in the *Microsoft Macro Assembler Reference Manual.* The manual describes the syntax and function of assembly language directives, operands, and expressions.

The Microsoft Macro Assembler supports the instruction sets of all processors in the 8086/186/286 family. This means you can assemble programs for computers having the 8086, 8088, 186, and 286 microprocessors and the 8087 and 287 coprocessors. For an explanation of these instructions, you will need to turn to one of the many books that define them. For your convenience, the *Microsoft Macro Assembler Reference Manual* defines the syntax and function of all instructions for all processors in this family.

1.3 How To Begin

You begin by creating an assembly language source file, then carrying out the steps needed to make an executable program. There are four steps:

1. Use a text editor to create an assembly language source file.

2. Use MASM to assemble the source file.

3. Use LINK to link the assembled file with other assembled files or with routines from libraries.

4. Use SYMDEB to test the resulting program.

You can automate these steps by using MAKE to create a description file containing the commands needed to invoke each step. You can make debugging easier by using CREF to make a cross reference listing of all symbols in your program. You can use LIB to construct the program libraries you may need to create your executable programs.

Once you have tested the program, you can invoke it from the MS™ DOS command line at any time. Programs that you create, like all other MS-DOS programs, can accept command parameters, be copied to other systems, and be invoked from batch files.

1.4 Notational Conventions

This manual uses the following notational conventions to define command syntax:

Convention	Meaning
Roman	Indicates command or parameter names that must be typed as shown. In most cases, upper and lowercase letters can be freely intermixed.
Italics	Indicates a placeholder, that is, a name that you must replace with the value or filename required by the program.
...	Ellipses. Indicates that you can repeat the preceding item any number of times.
[]	Brackets. Indicates that the enclosed item is optional. If you do not use the optional item, the program selects a default action to carry out.
|	Vertical bar. Indicates that only one of the separated items can be used. You must make a choice between the items.

Chapter 2
MASM:
A Macro Assembler

2.1 Introduction

The Microsoft Macro Assembler, MASM, assembles 8086, 186, and 286 assembly language source files and creates relocatable object files that can be linked and executed under the MS-DOS operating system. This chapter explains how to invoke MASM and describes the format of assembly listings generated by MASM. For a complete description of the syntax of assembly language source files, see the *Microsoft Macro Assembler Reference Manual.*

2.2 Starting and Using MASM

This section explains how to start and use MASM to assemble your program source files. You can use MASM in two different ways: with a command line or through a series of prompts.

Once you have started MASM, it either processes the files you have supplied or prompts for additional files. You can terminate MASM at any time by pressing the CNTRL-C key.

2.2.1 Assembling a Source File

You can assemble a program source file by typing the MASM command name and the names of the files you wish to process. The command line has the form

MASM [*options*] *source*, [*object*], [*listing*], [*cross-ref*]

The *options* can be any combination of MASM options. The options are described in Section 2.3, "Using MASM Options." Options can be placed anywhere on the command line.

The *source* must be the name of the source file to be assembled. If you do not supply a filename extension, MASM uses .ASM by default.

The optional *object* is the name of the file to receive the relocatable object code. If you do not supply a name, MASM uses a default name. The default filename is the same as the source file, except that the filename extension is replaced with .OBJ.

The optional *listing* is the name of the file to receive the assembly listing. The assembly listing lists the assembled code for each source statement and the names and types of symbols defined in the program. If you do not supply a listing filename, MASM does not create an assembly listing. If you do not supply a filename extension, MASM supplies .LST by default.

The optional *cross-ref* is the name of the file to receive the cross reference output. This output can be processed with CREF, the cross reference utility, to create a cross reference listing of the symbols in the program for use in program debugging. If you do not supply a filename, MASM does not create cross reference output. If you do not supply a filename extension, MASM supplies .CRF by default.

You can use a semicolon (;) in the command line to direct MASM to select defaults for the remaining filenames. A semicolon after the source filename selects a default object filename and suppresses creation of the assembly listing and cross reference files. A semicolon after the object filename suppresses just the listing and cross reference. A semicolon after the assembly listing filename suppresses the cross reference.

Note

Unless a semicolon (;) is used, all the commas in the command line are required. If you do not wish to supply a filename for a given file, place the commas that would otherwise separate the filename from the other names side-by-side (,,).

Spaces in a command line are optional. If you make an error entering any filenames, MASM displays an error message and prompts for new filenames, using the method described in the next section.

Examples

```
MASM  file.asm, file.obj, file.lst, file.crf
```

This example directs MASM to assemble the source file "file.asm." The generated relocatable code is copied to the object file "file.obj." MASM also creates an assembly listing and a cross reference file. These are written to "file.lst" and "file.crf," respectively.

```
MASM  startup, , stest;
```

This example directs MASM to assemble the source file "startup.asm." MASM then outputs the relocatable object code to the default object file, "startup.obj." MASM creates a listing file named "stest.lst," but the semicolon causes MASM to skip creating a cross reference file.

```
MASM  A:\src\build;
```

This example directs MASM to assemble the source file "build.asm" in the directory "\src" on drive A:. The semicolon causes MASM to create an object file named "build.obj" in the current directory, but prevents MASM from creating an assembly listing or cross reference file.

2.2.2 Assembling a Source File With Prompts

You can direct MASM to prompt you for the files it needs by starting MASM with just the command name. Follow these steps:

1. Type

   ```
   MASM
   ```

   and press the RETURN key at the MS-DOS command level. MASM displays a message, then displays the prompt

   ```
   Source filename [.ASM]:
   ```

2. Type the name of the file you wish to assemble and press the RETURN key. If you do not give a filename extension, MASM uses .ASM by default. MASM requires a source file, so you must enter a filename.

   Once you have pressed the RETURN key, MASM displays the prompt

   ```
   Object filename [source.OBJ]:
   ```

3. Type the name of the file to receive the relocatable object code and press the RETURN key. If you do not give a filename extension, MASM uses .OBJ by default. If you want to use the default filename (enclosed in brackets), do not type a filename. Just press the RETURN key. MASM replaces *source* with the filename of the given source file.

   Once you have pressed the RETURN key, MASM displays the prompt

   ```
   Source listing [NUL.LST]:
   ```

4. Type the name of the file to receive the assembly listing and press the RETURN key. If you do not give a filename extension, MASM uses .LST by default. If you do not want to create an assembly listing, do not type a filename. Just press the RETURN key.

   Once you have pressed the RETURN key, MASM displays the prompt

   ```
   Cross reference [NUL.CRF]:
   ```

5. Type the name of the file to receive the cross reference listing and press the RETURN key. If you do not supply a filename extension, MASM uses .CRF by default. If you do not want a cross reference listing, do not type a filename. Just press the RETURN key instead.

Once you have pressed the return key, MASM assembles the given source file.

Notes

You can specify one or more options at the end of each prompt line. Each option must be preceded by a forward slash (/). MASM options are described in section 2.3, "Using MASM Options."

You must use an appropriate pathname or device name for any file that is in another directory or on a different drive.

You can select the default responses by typing a semicolon (;) at any prompt after the source filename. When MASM encounters a semicolon, it immediately selects the default responses for any prompts that have not been answered and starts assembling the source file.

If MASM cannot find or open the named files, it redisplays the appropriate prompt and lets you type a new filename.

Examples

```
MASM

Source filename [.ASM]: file
Object filename [file.OBJ]:
Source listing [NUL.LST]: f123/D
Cross reference [NUL.CRF]: f123
```

This example directs MASM to assemble the source file "file.asm" and place the relocatable object code in the default object file "file.obj." The /D option directs MASM to create a pass 1 listing in the assembly listing file "f123.lst." MASM also creates a cross reference file named "f123.crf."

```
MASM

Source filename [.ASM]: file
Object filename [file.OBJ]: f123;
```

This example directs MASM to assemble the source file "file.asm" and place the relocatable object code in the object file "f123.obj."

The semicolon after the object filename prompt directs MASM to select the default filenames for the remaining prompts. This means MASM creates no assembly or cross reference listing.

2.3 Using MASM Options

The MASM options control the operation of the assembler and the format of the output files it generates.

MASM has the following options:

/D	Pass 1 Listing
/O	Octal Output Radix
/ML	Case Sensitivity in Names
/MX	Case Sensitivity in Public and External Names
/X	False Conditional Listing Toggle
/R	Real Floating Point Instructions
/E	Emulated Floating Point Instructions
/A	Alphabetical Ordering for Segments

You can place options anywhere on a MASM command line. An option affects all relevant files in the command line even if the option appears at the end of the line.

2.3.1 Creating a Pass 1 Listing

The **/D** option directs MASM to add a pass 1 listing to the assembly listing file, making the assembly listing show the results of both assembler passes. A pass 1 listing is typically used to locate and understand program phase errors. Phase errors occur when MASM makes assumptions about the program in pass 1 that are not valid in pass 2.

The **/D** option does not create a pass 1 listing unless you also direct MASM to create an assembly listing. It does direct MASM to display error messages for both pass 1 and pass 2 of the assembly, even if no assembly listing is created.

Example

```
MASM file,, file/D;
```

This example directs MASM to create a pass 1 listing for the source file "file.asm." The listing is placed in the file "file.lst."

2.3.2 Changing the Output Radix to Octal

The **/O** option directs MASM to display all numbers in the assembly listing as octal numbers. The actual code in the object file will be the same as if the **/O** option were not given, but the code in the assembly listing will be in octal.

The **/O** option does not take affect unless you direct MASM to create an assembly listing file.

Note

Future releases of MASM will not support the **/O** option.

Example

```
MASM file,, file/O;
```

This example directs MASM to create an assembly listing that displays all its numbers in the octal radix.

2.3.3 Preserving Lowercase Names

The **/ML** option directs MASM to preserve lowercase letters in label, variable, and symbol names. This means names that have the same spelling but use different case letters are considered unique. For example, with the **/ML** option, "DATA" and "data" are unique. Without the option, MASM automatically converts all lowercase letters in a name to uppercase.

The **/ML** option is typically used when a source file is to be linked with object modules created by a case-sensitive compiler.

Example

```
MASM file /ML,, file;
```

This example directs MASM to preserve lowercase letters in any names defined in the source file "file.asm."

2.3.4 Preserving Lowercase in Public and External Names

The **/MX** option directs MASM to preserve lowercase letters in public and external names only when copying these names to the object file. For all other purposes, MASM converts the lowercase letters to uppercase.

Public and external names are any label, variable, or symbol names that have been defined using the EXTRN or PUBLIC directives. Since MASM converts the letters to uppercase for assembly, these names must have unique spellings. That is, the names "DATA" and "Data" are not unique.

The **/MX** option is used to ensure that the names of routines or variables copied to the object module have the correct spelling. The option is used with any source file that is to be linked with object modules created by a case-sensitive compiler.

The **/MX** option overrides the **/ML** option if both are used in the same command line.

Example

```
MASM file /MX, , file;
```

This example directs MASM to preserve lowercase letters in any public or external names defined in the source file "file.asm."

2.3.5 Listing False Conditionals

The **/X** option directs MASM to copy to the assembly listing all statements forming the body of an IF directive whose expression (or condition) evaluates to false. If you do not give the **/X** option in the command line, MASM suppresses all such statements. The **/X** option lets you display conditionals that do not generate code. This

option applies to all IF directives: IF, IFE, IF1, IF2, IFDEF, IFNDEF, IFB, IFNB, IFIDN, and IFDIF.

The .SFCOND, .LFCOND, and .TFCOND directives modify the effect of the **/X** option. The .SFCOND and .LFCOND directives suppress false conditionals regardless of whether or not **/X** is given in the command line. The .TFCOND directive reverses the normal meaning of the **/X** option. When the **/X** option has been given and MASM encounters a .TFCOND directive in a source file, MASM suppresses all subsequent false conditionals. The next .TFCOND directive restores the listing. The following chart illustrates the effect of the .TFCOND, .SFCOND, and .LFCOND directives on the **/X** option:

If the source file has:	The /X option:
.SFCOND	Has no effect; listing is suppressed
.LFCOND	Has no effect; false conditionals are listed
.TFCOND	Suppresses the listing
No directive	Lists false conditionals

The **/X** option does not affect the assembly listing unless you direct MASM to create an assembly listing file.

Example

```
MASM file,, file/X;
```

If the source file, "file.asm." does not contain a .TFCOND directive, this example directs MASM to list all false conditionals it finds in the source file.

2.3.6 Creating Code For a Floating Point Processor

The **/R** option directs MASM to generate floating point instruction code that can be executed by an 8087 or 287 coprocessor. Programs created using the **/R** option can run only on machines having an 8087 or 287 coprocessor.

Example

```
MASM file/R,, file;
```

This example directs MASM to assemble the source file "file.asm" and create actual 8087 or 287 instruction code for floating point instructions.

2.3.7 Creating Code For a Floating Point Emulator

The **/E** option directs MASM to generate floating point instruction code that emulates the 8087 or 287 coprocessor. Programs created with the **/E** option must be linked with an appropriate math library before being executed. If a library is not specified, LINK cannot create an executable program.

If you intend to execute the program on machines without an 8087 or 287 coprocessor, you must link the program to the appropriate emulation library. This library contains specific routines that use standard 8086 instructions to emulate the floating point operations performed by the 8087 and 287.

If you intend to execute the program on machines that do have an 8087 or 287 coprocessor, you must link the program to the appropriate math library. This library contains specific routines that use the 8087 or 287 coprocessor to carry out floating point operations.

Math libraries are provided with some MS-DOS language products.

Example

```
MASM file /E;
```

This example directs MASM to create emulation code for any floating point instructions it finds in the program.

2.3.8 Outputting Segments in Alphabetical Order

The **/A** option directs MASM to place the assembled segments in alphabetical order before copying them to the object file. If this option is not given, MASM copies the segments in the order encountered in the source file.

Example

```
MASM file /A;
```

This example creates an object file, "file.obj," whose segments are arranged in alphabetical order. Thus, if the source file "file.asm" contains definitions for the segments "DATA," "CODE," and "MEMORY," the assembled segments in the object file have the order "CODE," "DATA," and "MEMORY."

2.4 Reading the Assembly Listing

MASM creates an assembly listing of your source file whenever you give an assembly listing filename on the MASM command line. The assembly listing contains a list of the statements in your program and the object code generated for each statement. The listing also lists the names and values of all labels, variables, and symbols in your source file. MASM creates one or more tables for macros, structures, records, segments, groups, and other symbols and places these tables at the end of the assembly listing.

MASM lists symbols only if it encounters any in the program. If there are no symbols in your program for a particular table, the given table is omitted. For example, if you use no macros in your program, you will not see a macro section in the symbol table.

The assembly listing will also contain error messages if any errors occur during assembly. MASM places the messages below the statements that caused the errors. At the end of the listing, MASM displays the number of error and warning messages it issued.

The following sections explain the format of the assembly listing and the meaning of special symbols used in the listing.

2.4.1 Reading Program Code

MASM lists the program code generated from the statements of a source file. Each line has the form:

[*line-number*] *offset code statement*

The *line-number* is from the first statement in the assembly listing. The line numbers are given only if a cross reference file is also being created. The *offset* is the offset from the beginning of the current segment to the code. The *code* is the actual instruction code or data generated by MASM for the statement. MASM gives the actual

numeric value of the code if possible. Otherwise, it indicates what action needs to be taken to compute the value. The *statement* is the source statement shown exactly as it appears in the source file, or after processing by a MACRO, IRP, or IRPC directive.

If any errors occur during assembly, the error message will be printed directly below the statement where the error occurred.

MASM uses the following special characters to indicate addresses that need to be resolved by the linker or values that were generated in a special way:

Character	Meaning
R	Relocatable address; linker must resolve
E	External address; linker must resolve
----	Segment/group address; linker must resolve
=	EQU or = directive
nn:	Segment override in statement
nn/	REP or LOCK prefix instruction
nn [xx]	DUP expression; nn copies of the value xx
+	Macro expansion
C	Included line from INCLUDE file

Example

```
Microsoft Macro Assembler          Page 1-1    11-01-84

1                                  extrn go:near
2
3       0000                       data    segment public 'DATA'
4                                          assume es:data
5       0000 0002                  s2      dw      2
6       0002                       data    ends
7
8       0000                       code    segment public 'CODE'
9                                          assume cs:code
10      0000                       start:
11      0000 E8 0000 E                             call    go
12      0003 36:A1 0000 R                  mov     ax, s2
13      0007 B4 4C                         mov     ah, 4ch
14      0009 CD 21                         int     21h
15      000B                       code    ends
16
17                                 end start
```

2.4.2 Reading a Macro Table

MASM lists the names and sizes of all macros defined in a source file. The list has two columns: Name and Length.

The Name column lists the names of all macros. The names are listed in alphabetical order and are spelled exactly as given in the source file except that lowercase letters are converted to uppercase (unless the **/ML** option is used). Names longer than 31 characters are truncated.

The Length column lists the size of the macro in terms of 32-byte blocks. This size is in hexadecimal.

Example

```
                Name                    Length
BIOSCALL . . . . . .                    0002
DISPLAY. . . . . .                      0005
DOSCALL. . . . . .                      0002
KEYBOARD . . . . . .                    0003
LOCATE . . . . . .                      0003
SCROLL . . . . . .                      0004
```

2.4.3 Reading a Structure and Record Table

MASM lists the names and dimensions of all structures and records in a source file. The table contains two sets of overlapping columns. The Width and # Fields list information about the structure or record. The Shift, Width, Mask, and Initial columns list information about the structure or record members.

The Name column lists the names of all structures and records. The names are listed in alphabetical order and are spelled exactly as given in the source file except that lowercase letters are converted to uppercase (unless the **/ML** option is used). Names longer than 31 characters are truncated.

For a structure, the Width column lists the size (in bytes) of the structure. The # Fields column lists the number of fields in the structure. Both values are in hexadecimal.

For fields of structures, the Shift column lists the offset (in bytes) from the beginning of the structure to the field. This value is in hexadecimal. The other columns are not used.

Example

```
                        Name            Width  # fields
                                        Shift  Width  Mask  Initial

PARMLIST . . . . . .                     001C   0004
  BUFSIZE. . . . . .                     0000
  NAMESIZE . . . . .                     0001
  NAMETEXT . . . . .                     0002
  TERMINATOR . . . .                     001B
```

For a record, the Width column lists the size (in bits) of the record. The # Fields column lists the number of fields in the record.

For fields in a record, the Shift count lists the offset (in bits) from the lower order bit of the record to the first bit in the field. The Width column lists the number of bits in the field. The Mask column lists the maximum value of the field, expressed in hexadecimal. The Initial column lists the initial value of the field, if any. For each field, the table shows the mask and initial values as if they were placed in the record and all other fields were set to 0.

Example

```
                Name                    Width   # fields
                                        Shift   Width  Mask  Initial

REC0 . . . . . .                        0008    0003
  FLD1 . . . . . .                      0006    0002   00C0  0040
  FLD2 . . . . . .                      0003    0003   0038  0000
  FLD3 . . . . . .                      0000    0003   0007  0003
REC1 . . . . . .                        000B    0002
  FL1. . . . . .                        0003    0008   07F8  0400
  FL2. . . . . .                        0000    0003   0007  0002
```

2.4.4 Reading a Segment and Group Table

MASM lists the names, sizes, and attributes of all segments and groups in a source file. The list has five columns: Name, Size, Align, Combine, and Class.

The Name column lists the names of all segments and groups. The names in the list are given in alphabetical order, except that the names of segments belonging to a group are placed under the group name. Names are spelled exactly as given in the source file; lowercase letters are converted to uppercase (unless the **/ML** option is used). Names longer than 31 characters are truncated.

The Size column lists the size (in bytes) of each segment. Since a group has no size, only the word GROUP is shown. The size, if given, is in hexadecimal.

The Align column lists the alignment type of the segment. The types can be any of the following:

BYTE
WORD
PARA
PAGE

If the segment is defined with no explicit alignment type, MASM lists the default alignment for that segment.

The Combine column lists the combine type of the segment. The types can be any one of the following:

NONE
PUBLIC
STACK
MEMORY
COMBINE

NONE is given if no explicit combine type is defined for the segment. NONE represents the private combine type.

The Class column lists the class name of the segment. The name is spelled exactly as given in the source file. If no name is given, none is shown.

For a complete explanation of the alignment, combine types, and class names, see Chapter 3, "LINK: A Linker."

Example

```
        Name          Size     Align    Combine Class

AAAXQQ . . . .        0000     WORD      NONE    'CODE'
DGROUP . . . .        GROUP
  DATA . . . .        0024     WORD      PUBLIC  'DATA'
  STACK. . . .        0014     WORD      STACK   'STACK'
  CONST. . . .        0000     WORD      PUBLIC  'CONST'
  HEAP . . . .        0000     WORD      PUBLIC  'MEMORY'
  MEMORY . . .        0000     WORD      PUBLIC  'MEMORY'
ENTXCM . . . .        0037     WORD      NONE    'CODE'
MAIN_STARTUP .        007E     PARA      NONE    'MEMORY'
```

2.4.5 Reading a Symbol Table

MASM lists the names, types, values, and attributes of all symbols in the source file. The table has four columns: Name, Type, Value, and Attr.

The Name column lists the names of all symbols. The names in the list are given in alphabetical order and are spelled exactly as given in the source file, except that lowercase letters are converted to uppercase (unless the **/ML** option is used). Names longer than 31 characters are truncated.

The Type column lists each symbol's type. A type is given as one of the following:

L NEAR	a near label
L FAR	a far label
N PROC	a near procedure label
F PROC	a far procedure label
Number	an absolute label
Alias	an alias for another symbol
Opcode	an instruction opcode
Text	a memory operand, string, or other value

If Type is Number, Opcode, Alias, or Text, the symbol is defined by an EQU directive or an = directive. The Type column also lists the symbol's length if it is known. A length is given as one of the following:

BYTE	one byte (8-bits)
WORD	one word (16-bits)
DWORD	doubleword (2 words)
QWORD	quadword (4 words)
TBYTE	ten-bytes (5 words)

A length can also be given as a number. In this case, the symbol is a structure, and the number defines the length (in bytes) of the structure. For example, the type

```
L 0031
```

identifies a label to a structure that is 31 bytes long.

The Value column shows the numeric value of the symbol. For absolute symbols, the value represents an absolute number. For labels and variable names, the value represents that item's offset from the beginning of the segment in which it is defined. If Type is Number, Opcode, Alias, or Text, the Value column shows the symbol's "value," even if the "value" is simple text. Number shows a constant numeric value. Opcode shows a blank (the symbol is an alias for an instruction mnemonic). Alias shows the name of another symbol. Text shows the "text" the symbol represents. Text is any operand that does not fit one of the other three categories.

The Attr column lists the attributes of the symbol. The attributes include the name of the segment in which the symbol is defined, if any, the scope of the symbol, and the code length. A symbol's scope is given only if the symbol is defined using the EXTRN or PUBLIC directives. The scope can be External or Global. The code length is given only for procedures.

Example

```
Symbols:

                N a m e         Type      Value   Attr

SYM0 . . . . . .                Number    0005
SYM1 . . . . . .                Text      1.234
SYM2 . . . . . .                Number    0008
SYM3 . . . . . .                Alias     SYM4
SYM4 . . . . . .                Text      5[BP][DI]
SYM5 . . . . . .                Opcode
SYM6 . . . . . .                L BYTE    0002  DATA
SYM7 . . . . . .                L WORD    0012  DATA    Global
SYM8 . . . . . .                L DWORD   0022  DATA
SYM9 . . . . . .                L QWORD   0000          External
LAB0 . . . . . .                L FAR     0000          External
LAB1 . . . . . .                L NEAR    0010  CODE
```

2.4.6 Reading a Pass 1 Listing

When you specify the **/D** option in the MASM command line, MASM adds a pass 1 listing to the assembly listing file, making the listing file show the results of both assembler passes. The listing is intended to help locate the source of phase errors.

The following examples illustrate the pass 1 listing for a source file that assembled without error. Although an error was produced on pass 1, MASM corrected the error on pass 2 and completed assembly correctly.

During pass 1, a JLE instruction to a forward reference produces an error message:

```
0017  7E 00               JLE     SMLSTK
 E r r o r   ---          9:Symbol not defined
0019  BB 1000             MOV     BX,4096
001C                SMLSTK:
```

MASM displays this error since it has not yet encountered the definition for the symbol SMLSTK.

By pass 2, SMLSTK has been defined and MASM can fix the instruction so no error occurs:

```
0017  7E 03              JLE     SMLSTK
0019  BB 1000            MOV     BX,4096
001C              SMLSTK:
```

The JLE instruction's code now contains 03 instead of 00. This is a jump of 3 bytes.

Since MASM generated the same amount of code for both passes, there was no phase error. If a phase error had occurred, MASM would have displayed an error message.

In the following program fragment, a mistyped label creates a phase error: In pass 1, the label "go" is used in a forward reference and creates a "Symbol not defined" error. MASM assumes that the symbol will be defined later and generates three bytes of code, reserving two bytes for the symbol's actual value.

```
0000                    code segment
0000 E9 0000  U              jmp   go
 E r r o r  ---          9: Symbol not defined
0003                    go      label   byte
0003 B8 0001                    mov     ax, 1

0006                    code    ends
```

In pass 2, the label "go" is known to be a label of BYTE type which is an illegal type for the JMP instruction. As a result, MASM produces only two bytes of code in pass 2, one less than in pass 1. The result is a phase error.

```
0000                    code segment
 0003 R                      jmp   go
 E r r o r  ---         57:Illegal size for item
0003                    go   label byte
 E r r o r  ---          6:Phase error between passes
0003 B8 0001                mov   ax, 1
0006                    code ends
```

Chapter 3
LINK: A Linker

3.1 Introduction

The Microsoft Linker, LINK, creates executable programs from object files generated by MASM or by high-level language compilers, such as C or Pascal. LINK copies the resulting program to an executable (.EXE) output file. The user can then run the program by typing the file's name on the MS-DOS command line.

To use LINK, you must create one or more object files, then submit these files, along with any required library files, to LINK for processing. LINK combines code and data in the object files and searches the named libraries to resolve external references to routines and variables. It then copies a relocatable, execution image and relocation information to the executable file. Using the relocation information, MS-DOS can load the executable image at any convenient memory location and execute it. LINK can process programs that contain up to 1 Mbyte of code and data.

The following sections explain how to use LINK to create executable programs. They also define the options you can use in a LINK command line to control the linking process. The last section in this chapter explains how LINK creates programs.

3.2 Starting and Using LINK

This section explains how to start and use LINK to create executable programs. You can use LINK in three different ways: through an MS-DOS command line, in response to prompts, or with a response file.

Once you have started LINK, it will either process the files you supplied or prompt you for additional files. You can stop LINK at any time by pressing the CNTRL-C key.

3.2.1 Creating a Program

You can create an executable program by typing LINK followed by the names of the files you wish to process. The command line has the form

LINK [*options*] *object-file*, [*exe-file*], [*map-file*], [*lib-file*]

The *options* are execution options that control the operation of LINK.

The *object-file* is the name or names of object files that you want to link together. The files must have been created using MASM or a high-level language compiler.

The optional *exe-file* is the name of the executable file you wish to create. LINK uses a default name if you do not supply one.

The optional *map-file* is the name of the file to receive the map listing. If you do not name a file and do not specify the **/MAP** or **/LINENUMBERS** option, no map file is created.

The optional *library-file* is the name or names of the libraries containing routines that you wish to link to your program. If you do not a specify library name, you must type a semicolon (;).

The commas separating the different types of files are required, even if no filename is supplied. If you give more than one *object-file* or *library-file*, you must separate the names with spaces or the plus sign (+). You can place options anywhere on the command line as long as they do not appear between filenames.

LINK requires at least one object file name in the command line. If you do not supply an *exe-file*, LINK creates a default name by using the filename of the first object file it finds in the line and appending the the filename extension .EXE to it.

LINK supplies default filename extensions for the files if you do not use extensions. LINK uses .OBJ for object files, .EXE for the executable file, .MAP for the map file, and .LIB for library files.

Examples

```
LINK file.obj,file.exe,file.map,file.lib
```

This example uses the object file "file.obj" to create the executable file "file.exe." LINK searches the library "file.lib" for routines and variables used within the program. It creates a map file named "file.map" that contains a list of the program's segments and groups.

```
LINK startup+file,file,file,;
```

This example creates an executable file named "file.exe" from two object files: "startup.obj" and "file.obj." LINK does create a map file, but does not search any libraries.

```
LINK moda+modb+modc+startup/PAUSE,,abc,\lib\math
```

This example links the object modules "moda.obj," "modb.obj," "modc.obj," and "startup.obj," searching the library file "math.lib" (in the \lib directory) for routines and data used in the program. It then creates an executable file named "moda.exe," and a map file named "abc.map." The /PAUSE option in the command line causes LINK to pause before creating the executable file.

3.2.2 Creating a Program Through Prompts

You can let LINK prompt you for the information you need by typing just the command name at the MS-DOS command level. Follow these steps:

1. Type

   ```
   LINK
   ```

   and press the RETURN key. LINK prompts you for the object files you wish to link by displaying the following message:

   ```
   Object Modules [.OBJ]:
   ```

2. Type the name or names of the object files you wish to link. If do not supply filename extensions, LINK supplies .OBJ by default. If you have more than one name, make sure you separate them with spaces or plus signs (+). If you have more names than can fit on one line, type a plus sign (+) as the last character on the line and press the RETURN key. LINK prompts for additional object files.

   Once you have given all object file names, press the RETURN key. LINK displays the prompt:

```
Run File [filename.EXE]:
```

3. Type the name of the executable file you wish to create and press the RETURN key. If you do not give an extension, LINK supplies .EXE by default. If you want LINK to supply a default executable filename, just press the RETURN key. The filename will be the same as the first object file, but the file will have the extension .EXE.

   Once you have pressed the RETURN key, LINK displays the prompt

```
List File [NUL.MAP]:
```

4. Type the name of the map file you wish to create, then press the RETURN key. If you do not supply a filename extension, LINK uses .MAP by default. If you do not want a map file, do not type a filename. Just press the RETURN key.

   Once you have pressed the RETURN key, LINK displays the prompt

```
Libraries [.LIB]:
```

5. Type the name or names of any libraries containing routines or variables referenced but not defined in your program. If you give more than one name, make sure the names are separated by spaces or plus signs (+). If you do not supply filename extensions, LINK uses .LIB by default. If you have more names than can fit on one line, type a plus sign (+) as the last character on the line and press the RETURN key. LINK prompts for additional filenames.

   After entering all names, press the RETURN key. If you do not want to search any libraries, do not enter any names. Just press the RETURN key.

LINK now creates the executable file.

Notes

When entering filenames, you must give a pathname or a device name for any file that is in another directory or on another disk. If LINK cannot find an object file, it displays a message and waits so that you can change disks if necessary.

You can add options to any input line by typing the option at the end of the line.

You can direct LINK to choose the default responses for all remaining prompts by typing a semicolon (;) after any prompt. (If you type it after the object file prompt, be sure to supply at least one filename.) When LINK encounters the semicolon, it immediately chooses the default responses and creates the executable file.

Example

```
LINK

Object Modules [.OBJ]: moda+modb+
Object Modules [.OBJ]: modc+startup/PAUSE
Run File [moda.EXE]:
List File [NUL.MAP]: abc
Libraries [.LIB]: \lib\math
```

This example links the object modules "moda.obj," "modb.obj," "modc.obj," and "startup.obj," searching the library file "math.lib" (in the \lib directory) for routines and data used in the program. It then creates an executable file named "moda.exe," and a map file named "abc.map." The /PAUSE option in the object file prompt line causes LINK to pause before creating the executable file.

3.2.3 Creating a Program With a Response File

You can create a program by listing the names of all the files to be processed in a "response file" and giving the name of the file on the LINK command line. The command line has the form

LINK @*filename*

The *filename* must be the name of the response file. It must be preceded by an at sign (@). If the file is in another directory or on another disk drive, it must have a pathname or device name.

You can name the response file anything you like. The file content has the general form

object-file
[*exe-file*]
[*map-file*]
[*library-file*]

Each group of filenames must be placed on separate lines. If you have more names than can fit on one line, you can continue the names on the next line by typing a plus sign (+) as the last character in the current line. If you do not supply a filename for a group, you must leave an empty line. Options can be given on any line.

You can place a semicolon (;) on any line in the response file. When LINK encounters the semicolon, it automatically supplies default filenames for all files you have not yet named in the response file. The remainder of the file is ignored.

When you create a program with a response file, LINK displays each response from your response file on the screen. If the response file does not contain names for required files, LINK prompts for the missing names and waits for you to enter responses by hand.

Example

```
moda modb modc startup
/PAUSE
abc
\lib\math
```

This response file tells LINK to link the four object modules "moda," "modb," "modc," and "startup." LINK pauses to permit you to swap disks before producing the executable file "moda.exe." LINK also creates a map file "abc.map," and searches the library "\lib\math.lib."

3.2.4 Giving Search Paths With Libraries

You can direct LINK to search directories and disk drives for the libraries you have named in a command by specifying one or more search paths with the library names, or by assigning the search paths to the environment variable LIB before you invoke LINK.

A search path is simply the path specification of a directory or drive name. You enter search paths along with library names on the LINK command line or in response to the "Libraries" command prompt. You can specify up to 16 search paths. You can also assign the search paths to the LIB variable using the MS-DOS **SET** command. In this case, the search paths must be separated by semicolons (;).

The search paths are used only if a library does not have an explicit path specification or drive name. LINK searches the current directory first. It then searches each of the directories and disk drives specified in the command. Finally, it searches the directories and disk drives specified by the LIB variable. LINK continues the search until it finds a library with the given name or runs out of places to search.

Directories and disk drives are searched in the order in which their search paths appear in the command or are assigned to LIB. If a library name has a path specification or a drive name, LINK searches for that library in the specified directory or disk drive only.

Examples

```
LINK file,,file.map,A:\altlib\math.lib+common+B:+D:\lib\
```

In this example, LINK will search only the "\altlib" directory on drive A: to find the library "math.lib," but to find "common.lib" it will search the current directory on drive A:, the current directory on drive B:, and finally the directory "\lib" on drive D:.

```
SET LIB=C:\lib;U:\system\lib

LINK file,,file.map,math+common
```

In this example, LINK will search the current directory, the directory "\lib" on drive C:, and the directory "\system\lib" on drive U: to find the libraries "math.lib" and "common.lib."

3.2.5 Map File

The map file lists the names, load addresses, and lengths of all segments in a program. It also lists the names and load addresses of any groups in the program, the program start address, and messages about any errors it may have encountered. If the /MAP option is used in the LINK command line, the map file lists the names and load addresses of all public symbols.

Segment information has the general form

```
Start  Stop   Length Name   Class
00000H 0172CH 0172DH TEXT   CODE
01730H 01E19H 006EAH DATA   DATA
```

The "Start" and "Stop" columns show the 20-bit addresses (in hexadecimal) of the first and last byte in each segment. These addresses are relative to the beginning of the load module which is assumed to be address 0000H. The operating system chooses its own starting address when the program is actually loaded. The "Length" column gives the length of the segment in bytes. The "Name" column gives the name of the segment, and the "Class" column gives the segment's class name.

Group information has the general form

```
Origin Group
0000:0 IGROUP
0173:0 DGROUP
```

In this example, IGROUP is the name of the code (instruction) group and DGROUP is the name of the data group.

At the end of the listing file, LINK gives you the address of the program entry point.

If you have given a /MAP option in the LINK command line, LINK adds a public symbol list to the map file. The symbols are presented twice: once in alphabetical order, then in order of load address. The list has the general form

```
ADDRESS                PUBLICS BY NAME
0000:1567      brk
0000:1696      chmod
0000:01DB      chkstk
0000:131C      clearerr
0173:0035      fac

ADDRESS                PUBLICS BY VALUE
0000:01DB      chkstk
0000:131C      clearerr
0000:1567      brk
0000:1696      chmod
0000:0035      fac
```

The address of the public symbols are in segment:offset format. They show the location of the symbol relative to the beginning of the load module, which is assumed to be at address 0000:0000.

When the /HIGH and /DSALLOCATE options are used and the program's code and data combined do not exceed 64 Kbytes, the map file may show symbols that have unusually large segment addresses. These addresses indicate a symbol whose location is below the actual start of the program code and data. For example, the symbol entry

```
FFF0:0A20      template
```

shows that "template" is located below the start of the program. Note that template's 20-bit address is 00920H.

3.2.6 The Temporary Disk File – VM.TMP

LINK normally uses available memory for the link session. If it runs out of available memory, it creates a temporary disk file named "VM.TMP" in the current working directory. When LINK creates this file, it displays the message

```
VM.TMP has been created.
Do not change diskette in drive, <d:>
```

After this message appears, you must not remove the disk from the given drive until the link session ends. After LINK has created the executable file, it deletes the temporary file automatically.

Warning

Do not use the filename VM.TMP for your own files. When LINK creates the temporary file, it destroys any previous file having the same name.

3.3 Using Link Options

The linker options specify and control the tasks performed by LINK. All options begin with the linker option character, the forward slash (/). You can use an option anywhere on a LINK command line.

LINK has the following options:

/PAUSE	Pause During Linking
/MAP	Public Symbol map
/STACK	Stack Size
/CPARMAXALLOC	Maximum Allocation Space
/HIGH	High Load
/DSALLOCATE	Data Group Allocation
/OVERLAYINTERRUPT	Overlay Interrupt
/LINENUMBERS	Line Number
/NOIGNORECASE	Case Sensitivity in Names
/NOGROUPASSOCIATION	Group Association Override
/NODEFAULTLIBRARYSEARCH	Default Library Override
/SEGMENTS	Segment Number Maximum
/DOSSEG	Use MS-DOS Segment Ordering

3.3.1 Pause During Linking

Syntax

/PAUSE

The **/PAUSE** option causes LINK to pause before writing the executable file to disk. This switch allows you to swap disks before LINK outputs the executable (.EXE) file.

If the **/PAUSE** switch is given, LINK displays the following message before creating the run file:

```
About to generate .EXE file
Change disks <hit any key>
```

LINK resumes processing when you press any key. LINK without the **/PAUSE** option performs the linking session from beginning to end without stopping.

Minimum abbreviation: **/P**

Note

Do not remove the disk used for the VM.TMP file, if one has been created.

Example

```
LINK file.obj/PAUSE,file.exe,,\lib\math.lib
```

This command causes LINK to pause just before creating the executable file "file.exe." After creating the executable file, MASM pauses again to let you replace the original disk.

3.3.2 Producing a Public Symbol Map

Syntax

/MAP

The **/MAP** option causes LINK to produce a listing of all public symbols declared in your program. This list is copied to the map file created by LINK. For a complete description of the listing file format, see Section 3.2.4, "Map File."

Note

If you do not specify a map file in a LINK command, you can use the **/MAP** option to force LINK to create a map file by placing the option at or before the "List file" prompt. LINK gives the forced map file the same filename as the first object file specified in the command and the default extension ".MAP."

Minimum abbreviation: **/M**

Example

```
LINK file.obj,file.exe,file.map/MAP,;
```

This command creates a map of all public symbols in the file "file.obj."

3.3.3 Setting the Stack Size

Syntax

/STACK: *size*

The **/STACK** option sets the program stack to the number of bytes given by *size*. The *size* can be any positive integer value in the range 1 to 65,535. The value can be a decimal, octal, or hexadecimal number. Octal numbers must begin with a zero. Hexadecimal numbers must begin with "0x".

LINK usually calculates a program's stack size automatically, basing the size on the size of any stack segments given in the object files. If **/STACK** is given, LINK uses the given *size* in place of any value it may have calculated.

LINK displays an error message if the program has no stack segments. To avoid this message, all programs should define at least one stack segment.

Minimum abbreviation: **/ST**

Examples

```
LINK file.obj/STACK:512,file.exe,,;
```

This example sets the stack size to 512 bytes.

```
LINK moda+modb, run/ST:0xFF,ab.map,\lib\start;
```

This example sets the stack size to 255 (FFH) bytes.

```
LINK startup+file/ST:030,file,,;
```

This example sets the stack size to 24 (30 octal) bytes.

3.3.4 Setting the Maximum Allocation Space

Syntax

/CPARMAXALLOC: *number*

The **/CPARMAXALLOC** option sets the maximum number of 16-byte paragraphs needed by the program when it is loaded into memory. This number is used by the operating system when allocating space for the program prior to loading it. The *number* can be any integer value in the range 1 to 65,535. It must be a decimal, octal, or hexadecimal number. Octal numbers must begin with a zero. Hexadecimal values must begin with "0x".

LINK normally sets the maximum number of paragraphs to 65,535. Since this represents all of memory, the operating system always denies the request and allocates the largest contiguous block of memory it can find. If the **/CPARMAXALLOC** option is used,

the operating system will allocate no more space than given by this option. This means any additional space in memory is free for other programs.

If *number* is less than the minimum number of paragraphs needed by the program, LINK ignores your request and sets the maximum value equal to the minimum. The minimum number of paragraphs needed by a program is never less than the number of paragraphs of code and data in the program.

Minimum abbreviation: **/C**

Examples

```
LINK file.obj/C:15, file.exe,,;
```

This example sets the maximum allocation to 15 paragraphs.

```
LINK moda+modb, run/CPARMAXALLOC:0xff, ab.map,;
```

This example sets the maximum allocation to 255 (FFH) paragraphs.

```
LINK startup+file, file/C:030,,;
```

This example sets the maximum allocation to 24 (30 octal) paragraphs.

3.3.5 Setting a High Start Address

Syntax

/HIGH

The **/HIGH** option sets the program's starting address to the highest possible address in free memory. This option actually causes LINK to add information to the executable file that makes the operating system load the program as high as possible.

If **/HIGH** is not given, the program's starting address is set as low as possible in memory.

Minimum abbreviation: **/H**

Example

```
LINK startup+file/HIGH,file,file, ;
```

This example sets the starting address of the program in "file.exe" to the highest possible address in free memory.

3.3.6 Allocating a Data Group

Syntax

/DSALLOCATE

The **/DSALLOCATE** option directs LINK to reverse its normal processing when assigning addresses to items belonging to the group named "DGROUP." Normally, LINK assigns the offset 0000H to the lowest byte in a group. If **/DSALLOCATE** is given, LINK assigns the offset FFFFH to the highest byte in the group. The result is data that appears to be loaded as high as possible in the memory segment containing DGROUP.

The **/DSALLOCATE** option is typically used with the **/HIGH** option to take advantage of unused memory before the start of the program. LINK assumes that all free bytes in DGROUP occupy the memory immediately before the program. To use the group, a segment register must be set to the start address of DGROUP.

Minimum abbreviation: **/D**

Example

```
LINK startup+file/HIGH/DSALLOCATE,file,file,;
```

This example directs LINK to place the program as high in memory as possible, then adjust the offsets of all data items in DGROUP so that they are loaded as high as possible within the group.

3.3.7 Display Line Numbers

Syntax

/LINENUMBERS

The **/LINENUMBERS** option directs LINK to list the starting address of each program source line. The starting address is actually the address of the instructions that make up the corresponding source line. LINK copies this information to the map file where you can use it for program debugging.

LINK carries out the line numbering only if you give a map file name in the LINK command line, and only if the given object file has line number information. Line numbering is available only in high-level language compilers. If an object file has no line number information, LINK ignores the **/LINENUMBERS** option.

Note

If you do not specify a map file in a LINK command, you can use the **/LINENUMBERS** option to force LINK to create a map file by placing the option at or before the "List file" prompt. LINK gives the forced map file the same filename as the first object file specified in the command and the default extension ".MAP."

Minimum abbreviation: **/LI**

Example

```
LINK file.obj/LINENUMBERS,file.exe,file.map, ;
```

This example causes the line number information in the object file "file.obj" to be copied to the map file "file.map."

3.3.8 Preserving Case

Syntax

/NOIGNORECASE

The **/NOIGNORECASE** option directs LINK to treat upper and lowercase letters in symbol names as distinct letters. Normally, LINK considers upper and lowercase letters to be identical, treating the names "TWO," "two," and "Two" as the same symbol. Using **/NOIGNORECASE** causes the LINK to treat these names as unique symbols.

The **/NOIGNORECASE** option is typically used with object files created by high-level language compilers. Some compilers treat upper and lowercase letters as distinct letters and assume that LINK will too.

Minimum abbreviation: **/NOI**

Example

```
LINK file.obj/NOI,file.exe,file.map,\lib\Slibc.lib;
```

This command causes LINK to treat upper and lowercase letters in symbol names as distinct letters. The object file "file.obj" is linked with routines from the standard C language library "\lib\Slibc.lib." The C language expects upper and lowercase letters to be treated separately.

3.3.9 Ignoring Default Libraries

Syntax

/NODEFAULTLIBRARYSEARCH

The **/NODEFAULTLIBRARYSEARCH** option directs LINK to ignore any library names it may find in an object file. A high-level language compiler may add a library name to an object file to ensure that a default set of libraries are linked with the program. Using this option overrides these default libraries and lets you explicitly name the libraries you want on the LINK command line.

Minimum abbreviation: **/NOD**

Example

```
LINK startup+file/NOD,file.exe,,\lib\math.lib
```

This example links the object files "startup.obj" and "file.obj" with routines from the library "\lib\math.lib." Any default libraries that may have been named in startup.obj or file.obj are ignored.

3.3.10 Removing Groups From a Program

Syntax

/NOGROUPASSOCIATION

The **/NOGROUPASSOCIATION** option directs LINK to ignore group associations when assigning addresses to data and code items.

Note

Use of this option is not recommended.

Minimum abbreviation: **/NOG**

3.3.11 Setting the Overlay Interrupt

Syntax

/OVERLAYINTERRUPT: *number*

The **/OVERLAYINTERRUPT** option sets the interrupt number of the overlay loading routine to *number*. This option overrides the normal overlay interrupt number (03FH).

The *number* can be any integer value in the range 0 to 255. It must be a decimal, octal, or hexadecimal number. Octal numbers must have a leading zero. Hexadecimal numbers must start with "0x".

Note

Using interrupt numbers that conflict with the standard MS-DOS interrupts is not recommended.

Minimum abbreviation: **/O**

Examples

```
LINK file.obj/O:255,file.exe,,;
```

This example sets the overlay interrupt number to 255.

```
LINK moda+modb, run/OVERLAY:0xff,ab.map,;
```

This example sets the overlay interrupt number 255 (FFH).

```
LINK startup+file,file/O:0377,,;
```

This example sets the overlay interrupt to 255 (377 octal).

3.3.12 Setting the Maximum Number of Segments

Syntax

/SEGMENTS: *number*

The **/SEGMENTS** option directs LINK to process no more than *number* segments per program. LINK displays an error message and stops if it encounters more than the given limit. The option is used to override the default limit of 128 segments.

The *number* can be any integer value in the range 1 to 1024. It must be a decimal, octal, or hexadecimal number. Octal numbers must have a leading zero. Hexadecimal numbers must start with "0x".

If **/SEGMENTS** is not given, LINK allocates enough memory space to process up to 128 segments. If a program has more than 128 segments, setting the segment limit higher will increase the number of segments LINK can process.

Minimum abbreviation: **/SE**

Example

```
LINK file.obj/SE:10,file.exe,,;
```

This example sets the segment limit to 10.

```
LINK moda+modb,run/SEGMENTS:0xff,ab.map,;
```

This example sets the segment limit to 255 (FFH).

```
LINK startup+file,file/SE:030,,;
```

This example sets the segment limit to 24 (30 octal).

3.3.13 Using DOS Segment Order

Syntax

/DOSSEG

The **/DOSSEG** option causes LINK to arrange all segments in the executable file according to the MS-DOS segment ordering convention. This convention has the following rules:

1. All segments having the class name, "CODE," are placed at the beginning of the executable file.

2. Any segments that do not belong to the group named "DGROUP" are placed immediately after the "CODE" segments.

3. All segments belonging to "DGROUP" are placed at the end of the file.

Minimum abbreviation: **/DO**

Example

```
LINK start+test/DOSSEG,test,,math+common
```

This command causes LINK to create an executable file, named "file.exe," whose segments are arranged according to the MS-DOS segment ordering convention. The segments in the object files "start.obj" and "test.obj," and any segments copied from the libraries "math.lib" and "common.lib" are arranged in the order specified above.

3.4 How LINK Works

LINK creates an executable file by concatenating a program's code and data segments according to the instructions supplied in the original source files. These concatenated segments form an "executable image" which is copied directly into memory when you invoke the program for execution. Thus, the order and manner in which LINK copies segments to the executable file defines the order and manner in which the segments will be loaded into memory.

You can tell LINK how to link a program's segments by giving segment attributes with a SEGMENT directive or by using the GROUP directive in the program to form segment groups. These directives define group associations, classes, alignments, and combine types that define the order and relative starting addresses of all segments in a program. This information works in addition to any information you supply through command line switches.

The following sections explain the process LINK uses to concatenate segments and resolve references to items in memory.

3.4.1 Alignment of Segments

LINK uses a segment's alignment type to set the starting address for the segment. The alignment types are BYTE, WORD, PARA, and PAGE. These correspond to starting addresses at byte, word, paragraph, and page boundaries, representing addresses that are multiples of 1, 2, 16, and 1024, respectively.

When LINK encounters a segment, it checks the alignment type before copying the segment to the executable file. If the alignment is WORD, PARA, or PAGE, LINK checks the executable image to see if the last byte copied ends at an appropriate boundary. If not, LINK pads the image with extra zero bytes.

3.4.2 Frame Number

LINK computes a starting address for each segment in a program. The starting address is based on a segment's alignment and the size of the segments already copied to the executable file. The address consists of an offset and a "canonical frame number." The canonical frame number specifies the address of the first paragraph in memory that contains one or more bytes of the segment. A frame number is always a multiple of 16 (i.e., a paragraph address). The offset is the number of bytes from the start of the paragraph to the first byte in the segment. For BYTE and WORD alignments, the offset may be non-zero. The offset is always zero for PARA and PAGE alignments.

The frame number of a segment can be obtained from the map file created by LINK when linking the segment. The frame number is the first five hexadecimal digits of the "Start" address specified for the segment.

3.4.3 Order of Segments

LINK copies segments to the executable file in the same order that it encounters them in the object files. This order is maintained throughout the program unless LINK encounters two or more segments having the same class name. Segments having identical class names belong to the same class, and are copied as a contiguous block to the executable file.

For example, in the following program fragment the segments "DATAX" and "DATAZ" form a class. Both segments are copied to the executable file before the "TEXT" segment.

```
DATAX  segment 'DATA'
DATAX  ends

TEXT   segment 'CODE'
TEXT   ends

DATAZ  segment 'DATA'
DATAZ  ends
```

All segments belong to a class. Segments for which no class name is explicitly defined have the "null" class name, and will be loaded as a contiguous block with other segments having the null class name.

LINK imposes no restriction on the number or size of segments in a class. The total size of all segments in a class can exceed 64 Kbytes.

3.4.4 Combined Segments

LINK uses combine types to determine whether or not two or more segments sharing the same segment name should be combined into a single, large segment. The combine types are **public**, **stack**, **memory**, **common**, and **private**.

If a segment has **public** type, LINK will automatically combine it with any other segments having the same name and belonging to the same class. When LINK combines segments, it ensures that the segments are contiguous and that all addresses in the segments can be accessed using an offset from the same frame address. The result is the same as if the segment were defined as a whole in the source file.

LINK preserves each individual segment's alignment type. This means that even though the segments belong to a single, large segment, the code and data in the segments do not lose their original alignment. If the combined segments exceed 64 Kbytes, LINK displays an error message.

If a segment has **stack** or **memory** type, LINK carries out the same combine operation as **public** segments. The only exception is that **stack** segments cause LINK to copy an initial stack pointer value to the executable file. This stack pointer value is the offset to the end of the first stack segment (or combined stack segment) encountered.

If a segment has **common** type, LINK will automatically combine it with any other segments having the same name and belonging to the same class. When LINK combines common segments, however, it places the start of each segment at the same address, creating a series of overlapping segments. The result is a single segment which is no larger than the largest segment combined.

A segment has **private** type only if no explicit combine type is defined for it in the source file. LINK does not combine private segments.

3.4.5 Groups

Groups let segments that are not contiguous and do not belong to the same class be addressable relative to the same frame address. When LINK encounters a group, it adjusts all memory references to items in the group so that they are relative to the same frame address. LINK does not check to see if all elements of a group fit within the same 64 Kbytes of memory.

Segments in a group do not have to be contiguous, do not have to belong to the same class, and do not have to have the same combine type. The only requirement is that all segments in the group fit within 64 Kbytes.

Groups do not affect the order in which the segments are loaded. Unless you use class names and enter object files in the right order, there is no guarantee that the segments will be contiguous. In fact, LINK may place segments that do not belong to the group in the same 64 Kbytes of memory. Although LINK does not explicitly check that all segments in a group fit within 64 Kbytes of memory, LINK is likely to encounter a fixup overflow error if this requirement is not met.

3.4.6 Fixups

Once the starting address of each segment in a program is known and all segment combinations and groups have been established, LINK can "fixup" any unresolved references to labels and variables. To fix up unresolved references, LINK computes an appropriate offset and segment address and replaces the temporary values generated by the assembler with the new values.

LINK carries out fixups for 4 different references:

- Short
- Near Self-Relative
- Near Segment-Relative
- Long

The size of the value to be computed depends on the type of reference. If LINK discovers an error in the anticipated size of a reference, it displays a fixup overflow message. This can happen, for example, if a program attempts to use a 16-bit offset to reach an instruction in a segment having a different frame address. It can also occur if all segments in a group do not fit within a single 64 Kbyte block of memory.

A short reference occurs in JMP instructions that attempt to pass control to labelled instructions that are in the same segment or group. The target instruction must be no more than 128 bytes from the point of reference. LINK computes a signed, 8-bit number for this reference. It displays an error message if the target instruction belongs to a different segment or group (has a different frame address), or the target is more than 128 bytes distant (either direction).

A near self-relative reference occurs in instructions which access data relative to the same segment or group. LINK computes a 16-bit offset for this reference. It displays an error if the data is not in the same segment or group.

A near segment-relative reference occurs in instructions which attempt to access data in a specified segment or group or relative to a specified segment register. LINK computes a 16-bit offset for this reference. It displays an error message if the offset of the target within the specified frame is greater than 64K or less than 0, or the beginning of the canonical frame of the target is not addressable.

A long reference occurs in CALL instructions that attempt to access an instruction in another segment or group. LINK computes a 16-bit frame address and 16-bit offset for this reference. LINK displays an error message if the computed offset is greater than 64K or less than 0, or the beginning of the canonical frame of the target is not addressable.

3.4.7 Controlling the Loading Order

You can control the loading order of the segments in a program by creating and assembling a dummy program file that contains empty segment definitions given in the order you wish to load your real segments. Once this file is assembled, you simply give it as the first object file in any invocation of LINK. LINK will automatically load the segments in the order given.

For example, the following dummy program file defines the loading order of segments in a program having segments named CODE, DATA, STACK, CONST, and MEMORY.

```
CODE   segment 'CODE'
CODE   ends
CONST  segment 'CONST'
CONST  ends
DATA   segment 'DATA'
DATA   ends
STACK  segment stack 'STACK'
STACK  ends
MEMORY segment 'MEMORY'
MEMORY ends
```

The dummy program file must contain definitions for all classes to be used in your program. If it does not, LINK will choose a default loading order which may or may not correspond to the order you desire. When linking your program, the dummy program must be the first object file specified in the LINK command line.

Note

Do not use a dummy program file with C, Pascal, or other high-level language programs. These languages define their own loading order. This order must not be modified.

You can force LINK to load MEMORY segments as the last segments in a program by placing an empty MEMORY segment at the end of your dummy program file. The empty segment should have the form

segment-name SEGMENT MEMORY '*class-name*'
segment-name ENDS

where *segment-name* is the name you intend to use for MEMORY segments and *class-name* is the name you intend to use for the memory class.

Example

```
MEMORY segment memory 'MEMORY'
MEMORY ends
```

Chapter 4
SYMDEB:
A Symbolic Debug Utility

4.1 Introduction

The Microsoft Symbolic Debug Utility (SYMDEB) is a debugging program that provides a controlled testing environment for executable files. SYMDEB lets you load, examine, and modify programs and other binary files. You can display program code, examine registers, set breakpoints, and trace the execution of instructions. SYMDEB can also debug programs that use the floating point emulation conventions used by Microsoft languages.

As a symbolic debug utility, SYMDEB lets you refer to data and instructions by name rather than by address. You can display a variable or execute a routine by providing a name — you do not need to provide an address. SYMDEB accepts symbol map (.SYM) files that contain address information about the global symbols in your program. You can create .SYM files by using MAPSYM, the Symbol Map File Utility, and the map files created by LINK.

SYMDEB breakpoint commands let you set, enable, display, and clear "sticky" breakpoints in your program code. Sticky breakpoints stop execution of your program at any instruction and remain active until you disable or remove them.

The SYMDEB expression evaluator lets you operate on symbols and numbers. You can perform all arithmetic, Boolean, and address operations, including extracting segment and offset values from symbols. You can refer to operands by value, address or port number.

Source Line Display and Numbering. A special feature of SYMDEB is source line display and numbering. This feature lets you debug C, Pascal, and FORTRAN programs at the source file level as well as at the machine level. SYMDEB commands let you choose the display level. You can display the actual source statements of a program, the disassembled machine code of the program, or a combination of source statements and disassembled machine code. SYMDEB will also accept source line numbers as arguments to display and breakpoint commands. This means you can refer to the individual statements of a source file without knowing their exact locations in memory.

This chapter explains how to use SYMDEB. In particular, it explains how to start the debugger, how to prepare and use symbol (.SYM) files, and how to use SYMDEB commands to debug programs.

4.2 Starting SYMDEB

You can start SYMDEB by typing its name at the MS-DOS command prompt. The SYMDEB command line has the form

SYMDEB [/IBM][*symfiles*] [*filespec* [*arglist*]]

where **/IBM** is the IBM-compatible option, *symfiles* are optional file specifications naming symbol files (.SYM), *filespec* is an optional file specification naming a binary or executable file to be loaded, and *arglist* is an optional list of program arguments. SYMDEB makes this argument list available to the loaded program.

Once started, SYMDEB displays a message followed by the SYMDEB command prompt (-). When you see the prompt you can enter SYMDEB commands.

Example

```
A> symdeb
Microsoft Symbolic Debug Utility
Version 3.0
(C)Copyright Microsoft Corp 1984
Processor is [8086]
-
```

In this example, "symdeb" is typed at the MS-DOS command line. The command displays the startup message and the SYMDEB prompt (-).

4.2.1 Starting SYMDEB With a Program File

You can direct SYMDEB to load an executable program file (.EXE or .COM) by giving the name of the file on the SYMDEB command line. For example, to load the file "file.exe" when you start SYMDEB, type

```
symdeb file.exe
```

When you load an program file, SYMDEB prepares a 256-byte header for the program in the lowest available segment in memory, then copies the contents of the file to the free memory just after the header. SYMDEB copies the size of the file (in bytes) to the BX:CX register pair. It then adjusts the segment and other registers to the initial values defined in the file.

4.2.2 Starting SYMDEB With Symbols

You can start SYMDEB for symbolic operation by giving one or more symbol files in the command line. Giving a symbol file directs SYMDEB to load the given file and lets you use the symbols defined by that file in SYMDEB commands. You can create symbol files for loading by using the MAPSYM program. See the section "Preparing a Symbol File" given below.

For example, to load the symbol file "file.sym" along with the program file "file.exe", type

```
symdeb file.sym file.exe
```

SYMDEB copies symbolic information from "file.sym" into memory, then loads "file.exe" after preparing the program header.

You can give more than one symbol file if you wish. Multiple symbol files are typically used with programs that consist of several separately-linked program files. You must make sure, however, that all symbol files are given before the program file. Any files given after the program file are assumed to be program arguments.

You do not have to load a program file when you load symbols. SYMDEB starts just as if you started it without a filename (see the section, "Starting SYMDEB Without a File").

4.2.3 Passing Arguments to a Loaded Program

You can pass one or more program arguments to the program being loaded by typing the arguments immediately after the program filename on the SYMDEB command line. SYMDEB copies all arguments to the program header exactly as you typed them.

For example, to pass the name of a data file "test.dat" and the options "/m" and "/b" to the program file "file.exe," type

```
symdeb file.sym file.exe test.dat /m /b
```

SYMDEB places the string of characters "test.dat /m /b" in the program header, then loads "file.exe." It also loads the symbol file "file.sym." Once arguments are loaded, "file.exe" can read them from the program header at any time.

For more information about the program header, see the section "Name Command" given later in this chapter.

4.2.4 Starting SYMDEB Without a File

You can start SYMDEB without a file by typing just SYMDEB. When you start SYMDEB without a file specification, it creates a program header, but does not attempt to load a program. You are then free to use the **Name** and **Load** commands to name and load whatever files you wish.

When you start SYMDEB without a file, it sets the segment registers to the bottom of free memory, sets the Instruction Pointer to 0100H, clears all flags, and sets the remaining registers to zero.

4.2.5 Preparing a Symbol File

You can prepare symbol files for use with programs to be debugged by SYMDEB by using the MAPSYM program. The program converts the contents of the program's symbol map (.MAP) file into a form suitable for loading with SYMDEB.

The MAPSYM command line has the form

MAPSYM [-l | /l] *filespec*

where *filespec* is the file specification for a symbol map (.MAP) file created during linking. If you do not give a filename extension,

".MAP" will be assumed. The symbol map file can be created by specifying a map file and giving the **/MAP** option with the LINK command. If source line debugging is desired, the **/LINENUMBERS** option must also be given. See Chapter 3, "LINK: A Linker," for details.

The **-l** and **/l** options are identical. They direct MAPSYM to display information about the conversion, such as the names of groups defined in the program, the program start address, and whether or not line numbers are present.

For example, to prepare the symbol file "file.sym" from the map file "file.map," type

```
mapsym file.map
```

The program creates the new file and copies the symbol information to it.

4.2.6 Starting SYMDEB on IBM-Compatible Systems

If your system is not an IBM Personal Computer but is designed to run IBM Personal Computer software, the **/IBM** option should be used to start SYMDEB in the IBM-compatible mode. This mode allows SYMDEB to take advantage of special hardware features of your system, such as the 80/40 column display and the 8259 Interrupt Controller. It does not affect the number or operation of SYMDEB commands. (Systems designed to run IBM software have the same basic input and output system (BIOS) as the IBM Personal Computer.)

The **/IBM** option is not required when starting SYMDEB on an IBM Personal Computer.

4.3 Using Control Keys

You can correct mistakes and stop commands by using the control characters and the special editing functions described in the MS-DOS *User's Guide*. The following sections explain how to stop and suspend commands.

4.3.1 Stopping a SYMDEB Command

You can stop a SYMDEB command at any time by pressing the CNTRL-C key. This key directs SYMDEB to terminate the current command and display the SYMDEB prompt.

The CNTRL-C key is typically used to stop a long **Dump** command. The key can also be used to retrieve control from a **Go** command that has entered an infinite loop, but only if the program is performing an input or output operation.

Note

If SYMDEB input has been redirected to "COM1" using the **Redirection** command, the CNTRL-C key is not available and will be ignored.

4.3.2 Suspending a Command

You can temporarily suspend SYMDEB output by pressing the CNTRL-S key. The CNTRL-S key directs SYMDEB to suspend output of the command until you signal it by pressing another key. To resume suspended output, press the CNTRL-S key again. To cancel any further output, press the CNTRL-C key. The CNTRL-S is typically used to hold the output of a **Dump** or **Unassemble** command so you can examine a particular byte or instruction.

Note

If SYMDEB input has been redirected to "COM1" using the **Redirection** command, the CNTRL-S key is not available and will be ignored.

4.3.3 Using Non-Maskable Interrupts

You can use the non-maskable interrupt (NMI) at any time to stop execution of a program being debugged. When a non-maskable interrupt occurs, SYMDEB stops program execution and displays the contents of registers and flags at the time of the interrupt. It also displays the next instruction to be executed.

To use non-maskable interrupts, your system must be equipped with one of the following:

- IBM® Professional Debugging Facility
- Software Probe (Atron™ Corp.)

On an IBM Personal Computer AT, the System Request (Sys Req) can be used to simulate a non-maskable interrupt. This interrupt is not available if interrupts have been disabled.

4.4 Commands

The following table lists all SYMDEB commands.

?	Display Values, Display Help	E	Enter
<	Redirect Input	F	Fill
>	Redirect Output	G	Go
=	Redirect Input and Output	H	Hex
A	Assemble	I	Input
BP	Breakpoint Set	L	Load
BC	Breakpoint Clear	M	Move
BD	Breakpoint Disable	N	Name
BE	Breakpoint Enable	O	Output
BL	Breakpoint List	P	PTrace
C	Compare	Q	Quit
D	Dump	R	Register
DA	Dump ASCII	S	Search, Set Source Mode
DB	Dump Bytes	T	Trace
DW	Dump Words	U	Unassemble
DD	Dump Doublewords	W	Write
DS	Dump Short Reals	X	Examine Symbol Map
DL	Dump Long Reals	XO	Open Symbol Map
DT	Dump Ten-Byte Reals		

The following sections describe the format for SYMDEB commands and command parameters.

4.4.1 Command Format

All SYMDEB commands have the general form

command-name parameter....

where *command-name* is a one- or two-character name, and *parameter* is a number, symbol, or expression that represents values or addresses to be used by the command. Any combination of uppercase and lowercase letters may be used in commands and parameters.

The number of parameters with each command depends on the command. If a command takes two or more parameters, you can separate them with a single comma (,), or with any number of spaces.

Examples

```
D cs:100 110
U cs:100 110
F ds:100,110 ff,fe,01,00
```

The following sections describe command parameters in detail.

4.4.2 Symbols

Syntax

name

A symbol is a name that represents a register, an absolute value, a segment address, or a segment offset. A symbol consists of one or more characters, but always begins with a letter, underscore (_), question mark (?), at sign (@), or dollar sign ($).

Symbols that name registers are always available (see the section, "Register Command," for a complete list). Other symbols are available only when the .SYM file(s) that define their names and values have been loaded.

Notes

SYMDEB treats corresponding upper- and lowercase letters as the same letter (case-insensitive).

Symbols whose spellings differ only in case are treated as the same symbol. If a symbol map file has two such symbols, only one of the symbols will be recognized by SYMDEB. Any attempt to access information about the other symbol will always return information about the first.

Symbols that have the same spelling as registers are ignored. Register names always take precedence over such symbols.

Examples

```
AX
_main
DGROUP
IP
```

4.4.3 Numbers

Syntax

*digits*Y
*digits*O
*digits*Q
*digits*T
*digits*H

A number represents an integer number. It is a combination of binary, octal, decimal, or hexadecimal *digits* and an optional radix. The *digits* can be one or more digits of the specified radix: Y, O, Q, T, or H. If no radix is given, H (hexadecimal) is assumed. The following table lists the digits that can be used with each radix:

Radix	Type	Digits
Y	Binary	0 1
O Q	Octal	0 1 2 3 4 5 6 7
T	Decimal	0 1 2 3 4 5 6 7 8 9
H	Hexadecimal	0 1 2 3 4 5 6 7 8 9 A B C D E F

Although a number can contain any number of digits, SYMDEB truncates leading digits if the number is greater than 65,535. Leading zeroes, if any, are ignored.

Examples

```
0111111Y      77Q     63T     3FH     3F
01001010100101Y       112450 4773T  12A5H  12A5
```

4.4.4 Addresses

Syntax

segment:offset

An address is a combination of two 16-bit values, one representing a segment address, the other a segment offset. The values combined specify a unique memory location.

A full address has both a segment address and an offset, separated by a colon (:). A partial address is just an offset. In both cases, the *segment* or *offset* can be any number, register name, or symbol. For most commands, a partial address is supplied with a default segment address; the default is the current contents of the DS segment register. For the **Assemble**, **Go**, **Load**, **PTrace**, **Trace**, **Unassemble**, and **Write** commands, the default segment address is the contents of the CS register.

Examples

```
CS:0100
04BA:IP
CS:_main
DGROUP:count
```

4.4.5 Address Range

Syntax

start-address end-address

A range is a combination of two memory addresses that specifies a sequence of contiguous memory locations. The *start-address* and *end-address* specify the first and last addresses in the range.

If a command takes a range but you do not supply a second address, SYMDEB usually assumes a range of 128 bytes. If a command takes a range followed immediately by a third parameter, you must supply a second address. If you do not, SYMDEB uses the third parameter as the second address.

Examples

```
CS:100 110
_main _main+20
```

4.4.6 Object Range

Syntax

start-address L *value*

An object range is a combination of a memory address and a count of "objects" that specifies a range of contiguous bytes, words, instructions or other objects in memory. The *start-address* specifies the address of the first object in the list and L *value* specifies the number of objects in the list.

An object range can be used with the **Dump**, **Fill**, **Search**, and **Unassemble** commands only. Each command determines the size and type of objects in the list: the **Dump Bytes** command has byte objects, the **Dump Words** command has words, the **Unassemble** command has instructions, and so on.

Examples

```
DGROUP:table L 100
_main L 20
```

4.4.7 Line Numbers

Syntax

```
. + | - number
. [filename:] number
.symbol[ + | - number]
```

A line number is a combination of decimal numbers, filenames, and symbols that specifies a unique line of text in a program source file.

In the first form, the combination specifies a relative line number. The *number* is an offset (in lines) from the current source line to the new line. If the plus sign (+) is given, the new line is closer to the end of the source file. If the minus sign (–) is given, the new line is closer to the beginning. SYMDEB displays an error if there is no current line number, or no source line exists for the specified line number.

In the second form, the combination specifies an absolute line number. If a *filename* is given, the specified line is assumed to be in the source file corresponding to the symbol file identified by *filename*. If no *filename* is given, the current instruction address (i.e., current values of the CS and IP registers) determines which source file contains the line. SYMDEB displays an error if *filename* does not exist, or no source line exists for the specified line.

In the third form, the combination specifies a symbolic line number. The *symbol* can be any instruction or procedure label. If *number* is given, the *number*, if given, is an offset (in lines) from the given label or procedure name to the new line. If the plus sign (+) is given, the new line is closer to the end of the source file. If the minus sign (–) is given, the new line is closer to the beginning. SYMDEB displays an error if the *symbol* does not exist, or no source line exists for the specified line number.

Examples

```
.+5             ;5th line down from current line.
.10             ;10th line in the current source file.
.sample:10      ;10th line in the source file named by "sample."
.main           ;First line in the routine "main."
.main+5         ;5th line in the routine "main."
```

The symbol "main" can also be used to specify a line number. In this case, "main" is equal to ".main." Note, however, that "main+3" specifies an address that is three *bytes* from "main," but ".main+3" specifies a source line that is three *lines* from "main."

4.4.8 Strings

Syntax

'characters'
"characters"

A string represents a list of ASCII values. It can be any number and combination of characters enclosed in single (') or double (") quotation marks. The starting and ending quotation marks must be the same type. If a matching quotation mark appears inside the string, it must be given twice to prevent SYMDEB from ending the string too soon.

Examples

```
'This is a string.'
"This is a string."
'This ''string'' is okay.'
"This ""string"" is okay."
'This "string" is okay.'
"This 'string' is okay."
```

4.4.9 Expressions

An expression is a combination of parameters and operators that evaluates to an 8-, 16-, or 32-bit value. Expressions can be used as values in any command.

An expression can combine any symbol, number, or address with any of the following unary and binary operators:

Unary Operators

Operator	Meaning	Precedence
+	Unary plus	Highest
–	Unary minus	
NOT	1's complement	
SEG	Segment address of operand	
OFF	Address offset of operand	
BY	Low-order byte from given address	
WO	Low-order word from given address	
DW	Doubleword from given address	
POI	Pointer (4 bytes) from given address	
PORT	One byte from given port	
WPORT	Word from given port	Lowest

Binary Operators

Operator	Meaning	Precedence
*	Multiplication	Highest
/	Integer division	
MOD	Modulus	
:	Segment override	
+	Addition	
–	Subtraction	
AND	Bitwise Boolean AND	
XOR	Bitwise Boolean exclusive OR	
OR	Bitwise Boolean OR	Lowest

Expressions are evaluated in order of operator precedence. If adjacent operators have equal precedence, the expression is evaluated from left to right. Parentheses can be used to override this order.

Examples

```
4+2*3                   ; equals 10 (0AH)
SEG 0001:0002           ; equals 1
OFF 0001:0002           ; equals 2
4+(2*3)                 ; equals 10 (0AH)
(4+2)*3                 ; equals 18 (12H)
```

4.5 Assemble Command

Syntax

A [*address*]

The **Assemble** command assembles 8086/8087/8088 mnemonics and places the resulting instruction code into memory at the given *address*. The command displays the address and the instruction code at the given address, then prompts for a new instruction. If no *address* is given, the assembly starts at the address given by the current values of the CS and IP registers.

Instructions can be entered in standard 8086/8087/8088 instruction mnemonics. To assemble a new instruction, press the RETURN key after entering the desired mnemonics. SYMDEB assembles the instruction and displays the next available address. To terminate assembly and return to the SYMDEB prompt, press the RETURN key only.

The rules for entry of instruction mnemonics are as follows:

1. Prefix mnemonics, such as WAIT and REP, must be specified before the instruction to which they apply. This must be on a separate line.

2. The far return mnemonic is RETF.

3. String manipulation mnemonics must explicitly state the string size. For example, use MOVSW to move word strings and MOVSB to move byte strings.

4. SYMDEB automatically assembles short, near or far jumps and calls, depending on byte displacement to the destination address. These may be overridden with the NEAR or FAR prefix. Examples:

```
JMP     502
JMP     NEAR 505
JMP     FAR  50A
```

   The NEAR prefix may be abbreviated to NE, but the FAR prefix cannot be abbreviated.

5. SYMDEB cannot tell whether some operands refer to a word memory location or to a byte memory location. In this case, the data type must be explicitly stated with the prefix "WORD PTR" or "BYTE PTR". Acceptable abbreviations are "WO" and "BY". Examples:

```
MOV     WORD PTR [BP], 1
MOV     BYTE PTR [SI-1], SYMBOL
```

6. SYMDEB cannot tell whether an operand refers to a memory location or to an immediate operand. SYMDEB uses the convention that operands enclosed in square brackets refer to memory. Examples:

```
MOV     AX,21
MOV     AX,[21]
```

7. The DB opcode assembles byte values directly into memory. The DW opcode assembles word values directly into memory. Examples:

```
DB      1,2,3,4,"THIS IS AN EXAMPLE"
DB      'THIS IS A QUOTE: "'
DB      "THIS IS A QUOTE: '"
DW      1000,2000,3000,"BACH"
```

8. SYMDEB supports all forms of register indirect commands. Examples:

```
ADD     BX,34[BP+2].[SI-1]
POP     [BP+DI]
PUSH    [SI]
```

9. All opcode synonyms are also supported. For example,

```
LOOPZ   100
LOOPE   100
JA      200
JNBE    200
```

10. For 8087 opcodes, the WAIT or FWAIT must be explicitly specified on a separate line immediately preceding the corresponding instruction. Example:

```
FWAIT
FADD ST,ST(3)
```

If a syntax error is found, SYMDEB displays the message "Error" and redisplays the current assembly address.

Examples

```
A CS:_main
```

This example starts assembly at the address given by "CS:_main."

```
A 04BA:0100
```

This example starts assembly at the address given by "04BA:0100."

4.6 Breakpoint Set Command

Syntax

BP[*n*] *address* [*passes*]

The **Breakpoint Set** command creates a "sticky" breakpoint at the given *address*. When encountered during program execution, sticky breakpoints cause the program to stop and SYMDEB to display the current values of all registers and flags. Sticky breakpoints, unlike breakpoints created by the **Go** command, remain in the program until removed using the **Breakpoint Clear** command, or temporarily disabled using the **Breakpoint Disable** command.

SYMDEB allows up to ten sticky breakpoints (0 through 9). The *n* specifies which breakpoint is to be created. Spaces between the **BP** and *n* are not allowed. If no *n* is given, the first available breakpoint number is used. The *address* can be any valid instruction address (that is, it must be the first byte of an instruction opcode). The *passes* specifies the number of times the breakpoint is to be ignored before being taken. It can be any 16-bit value.

Examples

```
BP _main
```

This example creates a sticky breakpoint at "_main."

```
BP8 _add
```

This example creates breakpoint 8 at address "_add."

```
BP 100 10
```

This example creates a breakpoint at address 100 in the current CS segment. This breakpoint is ignored 16 (10H) times before being taken.

4.7 Breakpoint Clear Command

Syntax

BC *list* | *

The **Breakpoint Clear** removes one or more breakpoints from a program. If *list* is given, the command removes the breakpoints named in the list. The *list* can be any combination of integer values from 0 to 9. If * is given, the command removes all breakpoints.

Examples

```
BC 0 4 8
```

This example removes breakpoints 0, 4, and 8.

```
BC *
```

This example removes all breakpoints.

4.8 Breakpoint Disable Command

Syntax

BD *list* | *

The **Breakpoint Disable** command temporarily disables one or more breakpoints from a program. The breakpoints are not deleted. They can be restored at any time by using the **Breakpoint Enable** command.

If *list* is given, the command disables the breakpoints named in the list. The *list* can be any combination of integer values from 0 to 9. If * is given, the command disables all breakpoints.

Examples

```
BD 0 4 8
```

This example disables breakpoints 0, 4, and 8.

```
BD *
```

This example disables all breakpoints.

4.9 Breakpoint Enable Command

Syntax

BE *list* | *

The **Breakpoint Enable** command restores one or more breakpoints that were temporarily disabled by a **Breakpoint Disable** command.

If *list* is given, the command enables the breakpoints named in the list. The *list* can be any combination of integer values from 0 to 9. If * is given, the command enables all breakpoints.

Examples

```
BE 0 4 8
```

This example enables breakpoints 0, 4, and 8.

```
BE *
```

This example enables all breakpoints.

4.10 Breakpoint List Command

Syntax

```
BL
```

The **Breakpoint List** command lists current information about all breakpoints created by the **Breakpoint** command. The command displays the breakpoint number, the enabled status, the address of the breakpoint, the number of passes remaining, and the initial number of passes (in parentheses).

If a breakpoint is not currently defined, nothing is disabled.

Example

```
BL
```

This example displays all breakpoints. The display should look like this:

```
0 e 04BA:0100
4 d 04BA:0503 4 (10)
8 e 0D2D:0001 3 (3)
```

In this example, breakpoint 0 and 8 are enabled (e) and 4 is disabled (d). Breakpoint 4 has an initial pass count of 10 and has 4 remaining passes to be taken before the breakpoint. Breakpoint 8 has initial pass count of 3 and has all 3 passes remaining. Breakpoint 0 shows no initial pass count. This means it was set to 1.

4.11 Compare Command

Syntax

C *range address*

The **Compare** command compares the bytes in the memory locations specified by *range* with the corresponding bytes in the memory locations beginning at *address*. If all corresponding bytes match, SYMDEB displays its prompt and waits for the next command. If one or more corresponding bytes do not match, each pair of mismatched bytes is displayed.

Examples

```
C 100,1FF   300
```

This example compares the block of memory from 100 to 1FFH with the block of memory from 300 to 3FFH.

```
C 100 L 100   300
```

This example compares the 256 (100H) bytes starting at memory address 100H with the 256 bytes starting at address 300H.

4.12 Display Command

Syntax

? *expression*

The **Display** command displays the value of the given *expression*. The command evaluates the expression, then displays the value in a variety of formats. The formats include a full address, a 16-bit hexadecimal value, a full 32-bit hexadecimal value, a decimal value (enclosed in parentheses), and a string value (enclosed in double quotation marks).

The *expression* can be any combination of numbers, symbols, addresses, and operators. For a list of operators, see the section, "Expressions," given earlier in this chapter.

Examples

```
? 3*4
```

This example displays the value of the expression 3*4.

```
? DS:table
```

This example displays the value of the symbolic address "DS:table."

```
? wo DGROUP:_bufsiz
```

This example displays the word at the symbolic address "DGROUP:_bufsiz."

4.13 Dump ASCII Command

Syntax

DA [*address* | *range*]

The **Dump ASCII** command displays the ASCII characters at a given *address* or in a given *range* of addresses. The command displays one or more lines of characters, depending on the *address* or *range* given. Up to 48 characters per line are displayed. Non-printable characters, such as newlines or formfeeds, are displayed as a dot (.).

If an *address* is given, the command continues to display ASCII characters until the first zero byte is encountered, or until 128 bytes have been displayed. If a *range* is given, the command continues to display ASCII characters until the end of the range. If no *address* or *range* is given, the command displays all characters up to the first zero byte, or until 128 bytes have been displayed. This display begins at the current dump address, the address immediately after the last byte previously displayed. If the **L** option is used in a range, the **Dump ASCII** command continues to display characters until the given number of characters have been displayed.

Examples

```
DA cs:100 110
```

This example displays the ASCII values of the bytes from "cs:100" to "cs:110." The display should look like this:

```
04BA:0100 A string..Text..
```

Non-printable characters are shown as dots.

```
DA
```

This example displays characters at the current dump address. If the last byte in the previous **Dump ASCII** command was 04BA:0110, this command displays the bytes starting at 04BA:0111.

```
DA name
```

This example displays the characters at the symbolic address "name".

4.14 Dump Bytes Command

Syntax

DB [*address* | *range*]

The **Dump Bytes** command displays the hexadecimal and ASCII values of the bytes at the given *address* or in the given *range* of addresses. The command displays one or more lines, depending on the address or range given. Each line displays the address of the first byte in the line, followed by up to 16 hexadecimal byte values. The byte values are immediately followed by the corresponding ASCII values. The hexadecimal values are separated by spaces, except the eighth and ninth values which are separated by a hyphen (-). ASCII values are printed without separation. A non-printable ASCII value is displayed as a dot (.). No more than 16 hexadecimal values are displayed in a line. The command displays values and characters until the end of the *range* or until the first 128 bytes have been displayed.

Examples

```
DB cs:100 110
```

This example displays the byte values from "cs:100" to "110". The display should look like this:

```
04BA:0100 41 20 73 74 72 69 6E 67-04 01 05 54 65 78 0D 0A  A string...Text..
04BA:0110 2E
```

ASCII characters are shown on the right.

```
DB
```

This example displays 128 bytes at the current dump address. If the last byte in the previous **Dump** command was 04BA:0110, this command displays the bytes starting at 04BA:0111.

```
DB table table+5
```

This example displays the first 6 bytes at the symbolic address "table".

4.15 Dump Words Command

Syntax

DW [*address* | *range*]

The **Dump Words** command displays the hexadecimal values of the words (2-byte values) at the given *address* or in the given *range* of addresses. The command displays one or more lines, depending on the address or range given. Each line displays the address of the first word in the line, followed by up to 8 hexadecimal word values. The hexadecimal values are separated by spaces. The command displays values until the end of the *range* or until the first 64 words have been displayed.

Examples

```
DW cs:100 110
```

This example displays the word values from "cs:100" to "cs:110". The display should look like this:

```
04BA:0100 2041 7473 6972 676E 0104 5405 7865 0A0D
04BA:0110 002E
```

No more than eight values per line are displayed.

```
DW
```

This example displays 64 words at the current dump address. If the last byte in the previous **Dump** command was 04BA:0110, this command displays the words starting at 04BA:0111.

```
DW table table+5
```

This example displays the words in a range from the symbolic address "table" to "table+5."

4.16 Dump Doublewords Command

Syntax

DD [*address* | *range*]

The **Dump Doublewords** command displays the hexadecimal values of the doublewords (4-byte values) at the given *address* or in the given *range* of addresses. The command displays one or more lines, depending on the address or range given. Each line displays the address of the first doubleword in the line, followed by up to 4 hexadecimal doubleword values. The hexadecimal values are separated by spaces. The command displays values until the end of the *range* or until the first 32 doublewords have been displayed.

Examples

```
DD cs:100 110
```

This example displays the doubleword values from "cs:100" to "cs:110". The display should look like this:

```
04BA:0100 7473:2041 676E:6972 5405:0104 0A0D:7865
04BA:0110 0000:002E
```

No more than four values per line are displayed.

```
DD
```

This example displays 32 doublewords at the current dump address. If the last byte in the previous **Dump** command was 04BA:0110, this command displays the doublewords starting at 04BA:0111.

```
DD table table+5
```

This example displays the doublewords in a range from the symbolic address "table" to "table+5."

4.17 Dump Short Reals Command

Syntax

DS [*address* | *range*]

The **Dump Short Reals** command displays the hexadecimal and decimal values of the short (4-byte) floating point numbers at the given *address* or in the given *range* of addresses. The command displays one or more lines, depending on the address or range given. Each line displays the address of the floating point number, followed by the hexadecimal values of the bytes in the number, followed by the decimal value of the number. The hexadecimal values are separated by spaces. The decimal value has the form

+ | - .*dd...d*E + | - *mm*

For display purposes, SYMDEB converts short reals to long reals before displaying. Although up to 16 decimal digits are displayed, only the first 7 digits are significant.

The command displays at least one value. If a *range* is given, it displays all values in the range.

Examples

```
DS ds:100
```

This example displays the floating point number at the address "ds:100." The display should look like this:

```
04BA:0100 00 00 20 40  +.25e+1
```

Only one value per line is displayed.

```
DS p1
```

This example displays the short floating point number at the symbolic address "pi."

4.18 Dump Long Reals Command

Syntax

DL [*address* | *range*]

The **Dump Long Reals** command displays the hexadecimal and decimal values of the long (8-byte) floating point numbers at the given *address* or in the given *range* of addresses. The command displays one or more lines, depending on the address or range given. Each line displays the address of the floating point number, followed by the hexadecimal values of the bytes in the number, followed by the decimal value of the number. The hexadecimal values are separated by spaces. The decimal value has the form

+ | - .*dd...d*E + | - *mm*

Up to 16 decimal digits are displayed.

The command displays at least one value. If a *range* is given, it displays all values in the range.

Examples

```
DL ds:100
```

This example displays the floating point number at the address "ds:100." The display should look like this:

```
04BA:0100 86 37 6B F0 BE 2A 57 3F  +.1413999327367877 4e-2
```

Only one value per line is displayed.

```
DL gamma
```

This example displays the long floating point number at the symbolic address "gamma."

4.19 Dump Ten-Byte Reals Command

Syntax

DT [*address* | *range*]

The **Dump Ten-Byte Reals** command displays the hexadecimal and decimal values of the ten-byte floating point numbers at the given *address* or in the given *range* of addresses. The command displays one or more lines, depending on the address or range given. Each line displays the address of the floating point number, followed by the hexadecimal values of the bytes in the number, followed by the decimal value of the number. The hexadecimal values are separated by spaces. The decimal value has the form

+ | - .*dd...d*E + | - *mm*

Up to 16 decimal digits are displayed.

The command displays at least one value. If a *range* is given, it displays all values in the range.

Examples

```
DT ds:100
```

This example displays the floating point number at the address "ds:100." The display should look like this:

```
04BA:0100 86 37 6B F0 BE 2A 57 3F 00 00 +.1413999327367877 4e-2
```

Only one number per line is displayed.

```
DT gamma
```

This example displays the ten-byte floating point number at the symbolic address "gamma."

4.20 Dump Command

Syntax

D [*address* | *range*]

The **Dump** command displays the contents of memory at the given *address* or in the given *range* of addresses. The command displays memory in the same format defined by the previous **DA**, **DB**, **DW**, **DD**, **DS**, **DL**, or **DT** command. The command displays one or more lines, depending on the address or range given. Each line displays the address of the first item displayed. The command always displays at least one value. If a *range* is given, it displays all values in the range. If no *address* or *range* is given, the command displays the contents of memory at the next byte after the last one displayed.

Note

The **Dump** command name must be separated by at least one space from any *address* or *range* value.

Examples

```
DA ds:100
04BA:0100 A string..
D
04BA:010B Text...
```

In this example, the **Dump** command displays the ASCII string at the address immediately following the string displayed by the **Dump ASCII** command.

```
DW ds:100 101
04BA:0100 2041
D ds:324 325
04BA:0324 FE31
```

In this example, the **Dump** command displays the word at the address "ds:324."

```
DS p1
04BA:0100 00 00 20 40  +.25e+1
D gamma
04BA:0104 00 00 20 40  +.25e+1
```

In this example, the **Dump** command displays the short floating point number at the symbolic address "pi."

4.21 Enter Command

Syntax

E *address* [*list*]

The **Enter** command enters one or more byte values into memory at the specified *address*. If the optional *list* is given, the command replaces the byte at the given address and the bytes at each subsequent address until all values in the list have been used. If no *list* is given, the command prompts for a replacement value.

If an error occurs, all byte values remain unchanged.

If you do not supply a *list*, SYMDEB prompts for a new value at *address* by displaying this address and its current value followed by a dot (.). You can then replace the value, skip to the next value, return to a previous value, or exit the command by following these steps:

1. **To replace the byte value**, simply type the new value after the current value. Make sure you typed a 1- or 2-digit hexadecimal number. The command ignores extra trailing digits or other characters.

2. **To skip to the next byte**, press the SPACE bar. Once you have skipped to the next byte, you can change its value or skip to the next byte. If you skip beyond an 8-byte boundary, SYMDEB starts a new display line by displaying the new address and value.

3. **To return to the preceding byte**, type a hyphen (-). When you return to the preceding byte, SYMDEB starts a new display line with the address and value of that byte.

4. **To exit the** E *command*, press the RETURN key. You can exit the command at any time.

Examples

```
E CS:100 1 2B E5
```

This example replaces the three bytes at CS:100, CS:101, and CS:102 to 1, 2B, and E5, respectively.

```
E CS:100
```

This example causes SYMDEB to display a prompt like:

```
04BA:0100  EB._
```

You can then change the value EB to the new value 41 by typing 41:

```
04BA:0100  EB.41_
```

You can then skip to the next byte value by pressing the SPACE bar:

```
04BA:0100  EB.41    10._
```

You can return to the previous value by typing a hyphen:

```
04BA:0100  EB.41    10.-
04BA:0100  41._
```

4.22 Examine Symbol Map Commands

Syntax

X [*]
X? [*mapname*!] [*segname*:] [*symname*]

The **Examine Symbol Map** command displays the name and address of the symbols in the current symbol maps. SYMDEB creates a symbol map for each symbol filename given in the SYMDEB com-

mand line. The **Examine Symbol Map** command can then be used to examine the contents of the maps.

The **X** command displays the name and load segment addresses of the current symbol map and the segments in that map. If the asterisk (*) is specified, the command displays the names and load segment addresses for all symbol maps.

The **X?** command displays the names and addresses of one or more symbols in the symbol map. If a *mapname*! is given, the command displays information for that symbol map. The *mapname* must be the filename (without extension) of the corresponding symbol file. If a *segname*: is given, the command displays the name and load segment address for that segment. The *segname* must be the name of a segment named within the explicitly given or currently open symbol map. If a *symname* is given, the command displays the segment address and segment offset for that symbol. The *symname* must be the name of a symbol in the given segment.

To display information about more than one segment or symbol, the *segname* or *symname* can contain an asterisk (*). The asterisk acts as a wildcard character, and SYMDEB displays information about all segments or symbols whose names start with the same characters that *segname* or *symname* start with. For example, "F*:" matches all segment names that start with the letter "F," and "_*" matches all symbols that start with an underscore (_).

Examples

```
X
```

This example displays the name of the current symbol map and the names and load segment addresses of the segments in that map.

```
X? test!
```

This example displays the load segment address of the symbol map file "test."

```
X? igroup:
```

This example displays the load segment address of the segment named "igroup" in the currently open symbol map.

```
X? test!igroup:
```

This example displays the load segment address of the segment named "igroup" in the symbol map "test."

```
X? start
```

This example displays the segment address and segment offset of the symbol "start" in the currently open symbol map.

```
X? test!start
```

This example displays the segment address and segment offset of the symbol "start" in the symbol map "test."

```
X? test!igroup:start
```

This example displays the segment address and segment offset of the symbol "start" in the segment "igroup" in the symbol map "test."

```
x? code*
```

This example displays the segment address and segment offset of all symbols in the current symbol map that begin with the letters "code."

4.23 Fill Command

Syntax

F *range list*

The **Fill** command fills the addresses in the given *range* with the values in the *list*. If *range* specifies more bytes than the number of values in the list, the list is repeated until all bytes in the range are filled. If *list* has more values than the number of bytes in the range, the command ignores any extra values.

Examples

```
F CS:100 L 100 FF
```

This example fills memory locations CS:100 through CS:1FF with the byte value FFH.

```
F DGROUP:table L 64  42 45 52 54 41
```

This example fills the 100 (64H) bytes starting at "DGROUP:table" with the given byte values. These five values are repeated until all 100 bytes are filled.

4.24 Go Command

Syntax

G [=*start-address*] [*break-address*],,,

The **Go** command passes execution control to the program at the given *start-address*. Execution continues to the end of the program or until a *break-address* is encountered. The program also stops at any breakpoints set using the **Breakpoint Set** command.

If no *start-address* is given, the command passes execution to the address specified by the current values of the CS and IP registers. The equal sign (=) may be used only when a *start-address* is given.

If a *break-address* is given, it must specify an instruction address (that is, the address must contain the first byte of an instruction opcode). Up to ten addresses can be given at one time. The addresses can be given in any order. Only the first address encountered during execution will cause a break. All others are ignored. If you attempt to set more than ten breakpoints, SYMDEB displays an error message.

When program execution reaches a breakpoint, SYMDEB displays the current values of all registers and flags. It also displays the next instruction to be executed. The display has the same form as the **Register** command.

Notes

The **Go** command uses an IRET instruction to pass control to a program. To do so, it must set the user stack pointer and push the user flags, CS register, and IP registers onto the user stack. If the user stack does not have 6 bytes available or is in invalid memory, the **Go** command may cause an operating system crash.

To create a breakpoint, SYMDEB places an INT 3 instruction (interrupt code 0CCH) at each breakpoint address, then restores these addresses to their original instructions when a breakpoint is encountered. If execution continues to the end of the program, however, or is halted by some other means, SYMDEB does not replace the interrupt code. For this reason, you should reload the program with the **Name** and **Load** commands before attempting to run the program again.

SYMDEB displays the message "Program terminated normally" whenever execution reaches the program end.

Examples

```
G =CS:0  CS:7550
```

This example passes execution control to the program at the address CS:0. If the instruction at the breakpoint address CS:7550 is encountered, SYMDEB stops execution and displays the current values of registers and flags.

```
G
```

This example passes control to the instruction pointed to by the current values of the CS and IP registers. This command is typically used to continue execution after a breakpoint has been encountered.

```
G =_main  _add
```

This example passes control to the instruction named by the symbolic address "_main". A breakpoint is set at the address "_add".

4.25 Help Command

Syntax

```
?
```

The **Help** command displays the following list of SYMDEB commands:

```
A [<address>] - assemble
BC <bp> - clear breakpoint(s)
BD <bp> - disable breakpoint(s)
BE <bp> - enable breakpoint(s)
BL <bp> - list breakpoint(s)
BP [bp] <address> - set breakpoint
C <range> <address> - compare
DA[<range>] - dump asciz string
DB[<range>] - dump bytes
DW[<range>] - dump words
DD[<range>] - dump doublets
DS[<range>] - dump short float
DL[<range>] - dump long float
DT[<range>] - dump tempreal float
E [<address>] [<list>] - enter
F <range> <list> - fill
G [=<address> [<address>...]] - go

? - help menu
? <expr> - display expression
> {CON|COM1} - Redirect output
< {CON|COM1} - Redirect input
= {CON|COM1} - Redirect both
H <value> <value> - hexadd
I <value> - input from port
L [<address> [<drive><rec><rec>]] - load
M <range> <address> - move
N <filename> [<filename>...] - name
O <value> <byte> - output to port
P - program step
Q - quit
R [<reg>] - register
S <range> <list> - search
S {-|&|+} - source level debugging
T [=<address>] [<value>] - trace
U [<range>] - unassemble
W [<address> [<drive><rec><rec>]] - write
X[?] <symbol> - examine symbol(s)
XO <symbol> - open map/segment
```

4.26 Hex Command

Syntax

H *value1 value2*

The **Hex** command displays the sum and difference of two hexadecimal numbers. SYMDEB adds *value1* to *value2* and displays the result. It then subtracts *value2* from *value1* and displays that result. The results are displayed on one line and are always in hexadecimal.

To evaluate more general expressions, see section 4.12, "Display Command."

Examples

```
H 3 4
```

This example displays the results 7 and FFFF.

```
H 110 100
```

This example displays the results 210 and 10.

4.27 Input Command

Syntax

I *port*

The **Input** command reads and displays one byte from the given input *port*. The *port* can be any 16-bit port address.

Example

```
I 2F8
```

This example displays the byte value read from input port number 2F8.

4.28 Load Command

Syntax

L [*address* [*drive record count*]]

The **Load** command copies the contents of a named file or the contents of a given number of logical disk records into memory. The contents are copied to the given *address* or to a default address, and the BX:CX register pair is set to the number of bytes loaded.

To load a file, a filename must be supplied before the **Load** command can be used. You can give a name by using the **Name** command (see section 4.30). You can also supply a name by passing it as a program argument when you start SYMDEB (see the section, "Passing Arguments to a Loaded Program," given earlier in this

chapter). If you do not supply a name, **Load** uses whatever name is currently at location DS:5C, where DS is the current value of the DS register. This is the location that receives any filename given with **Name** or any filename passed as a program argument.

If an *address* is given, the command places the contents of the file or sectors at the memory locations starting at *address*. Otherwise, it places the contents at the address given by CS:100, where CS is the current value of the CS register.

To load logical records from a disk, the explicit values for *address*, *drive*, *record*, and *count* must be given. The *drive* must name the drive to be read. It can be any number in the range 0 to 3, representing drives A: (0), B: (1), C: (2), and D: (3). The *record* names the first logical record to be read from the drive. It can be any 1- to 4-digit hexadecimal number. The *count* specifies the number of records to read from the disk. It can be any 1- to 4-digit hexadecimal number.

Notes

If the named file has a .EXE extension, **Load** adjusts the load address to the address given in the .EXE file header. This means that the *address* parameter is always ignored for .EXE files.

Since **Load** strips any header information from an .EXE file before loading, the number of bytes actually loaded will be different than the number of bytes in the .EXE file.

If the named file has a .HEX extension, the **Load** command adds that file's start address to *address* before loading the file. If no *address* is given, the file is loaded at its start address.

Example

```
N file.com
L
```

This example loads the file named "file.com" into memory at the address CS:100. The number of bytes loaded are copied to the BX:CX register pair.

```
L DGROUP:table
```

This example loads a file into the memory locations starting at the symbolic address "DGROUP:table." The command uses whatever filename is currently at location DS:5C.

```
L workspace 2 34 3
```

This example loads 3 logical records, beginning with logical record number 52 (34H), into memory at the symbolic address "workspace."

4.29 Move Command

Syntax

M *range address*

The **Move** command moves the block of memory specified by *range* to the location starting at *address.*

All moves are guaranteed to be performed without data loss, even where the source and destination blocks overlap. This means the destination block is always an exact duplicate of the original source block. If the destination block overlaps some portion of the source block, the original source will be changed.

To prevent data loss, **Move** copies data from the source block's lowest address first whenever the source is at a higher address than the destination. If the source is at a lower address, **Move** copies data from the source's highest address first.

Example

```
M CS:100 110 CS:500
```

This example moves all bytes in the range CS:100 to CS:110 to the memory locations starting at CS:500.

```
M DS:table L 100 workspace
```

This example copies the 256 (100H) bytes at the symbolic address "DS:table" to the address "workspace."

4.30 Name Command

Syntax

N [*filename*] [*arguments*]

The **Name** command sets the filename for subsequent **Load** and **Write** commands, or sets program arguments for subsequent execution of a loaded program.

If *filename* is given, all subsequent **Load** and **Write** commands will use this name when accessing disk files.

If *arguments* are given, the command copies all arguments, including spaces, to the memory location starting at DS:81 and sets the byte at DS:80 to a count of the total number of characters copied. In both cases, DS is the current value of the DS register. Once copied, the arguments are available for access by the program being debugged.

Notes

If the first two *arguments* are also filenames, the command creates File Control Blocks at addresses DS:5C and DS:6C and copies the names (in proper format) to these blocks. The FCBs can then be used by the program being debugged.

Name also treats *filename* as an argument, copying it to DS:81 and creating an FCB for it at DS:5C. Therefore, setting a new filename for **Load** and **Write** destroys any previous program arguments.

Each **Name** command changes or destroys one or more of the following memory locations:

DS:5C	FCB for file 1
DS:6C	FCB for file 2
DS:80	Count of characters
DS:81	All characters typed

Examples

```
N file1.exe
```

This example sets the filename for subsequent **Load** and **Write** commands to "file1.exe."

```
N file2.dat file3.dat /m /b
```

This example sets the program arguments for the program being debugged. The command creates FCBs for the files "file2.dat" and "file3.dat". It also copies the entire command line, except the command letter N, to DS:81.

4.31 Open Map Command

Syntax

XO [*mapname*!] *segname*

The **Open Map** command sets the active symbol map and/or segment. If *mapname*! is given, the command sets the active symbol map to the given map. The *mapname* must be the filename (without extension) of one of the symbol files specified in the SYMDEB command line. If *segname* is given, the command sets the active segment to the named segment. The *segname* must be the name of a segment in the specified symbol map.

Examples

```
XO test!
```

This example activates the symbol map "test."

```
XO  test!igroup
```

This example activates the segment "igroup" in the symbol map "test."

```
XO dgroup
```

This example activates the segment "dgroup" in the current symbol map.

4.32 Output Command

Syntax

O *port byte*

The **Output** command sends the given *byte* to the specified output *port*. The *port* can be any 16-bit port address.

Examples

```
O  2F8   4F
```

This example sends the byte value 4F (hexadecimal) to output port 2F8.

```
O  3  21
```

This example sends the byte value 21 (hexadecimal) to output port 3.

4.33 PTrace Command

Syntax

P [=*start-address*] [*count*]

The **PTrace** command executes the instruction at the given *start-address*, then displays the current values of all registers and flags. The display has the same format as the **Register** command.

If the optional *start-address* is given, the command starts execution

at the given address. Otherwise, it starts execution at the instruction pointed to by the current CS and IP registers. The equal sign (=) may be used only if a *start-address* is given.

If the optional *count* is given, the command continues to execute *count* instructions before stopping. The command displays the current values of the registers and flags for each instruction before executing the next.

Note

The **PTrace** command is identical to the **Trace** command, except that it automatically executes and returns from any calls or software interrupts it encounters. The **Trace** command always stops after executing the call or interrupt, leaving execution control inside the called routine.

Examples

```
P =_main
```

This example executes the instruction at "_main," then displays the current values of the registers and flags. It also displays the next instruction to be executed.

```
P
```

This example executes the instruction pointed to by the current CS and IP register values.

4.34 Quit Command

Syntax

```
Q
```

The **Quit** command terminates the SYMDEB utility and returns control to MS-DOS.

Example

```
-Q
```

This example terminates SYMDEB.

4.35 Redirection Commands

Syntax

< device-name
> device-name
= device-name

The **Redirection** commands redirect the command input and output to the device named by *device-name*. The **<** command causes SYMDEB to read all subsequent command input from the given device. The **>** command causes SYMDEB to write all subsequent command output to the given device. The = command causes SYMDEB to both read and write to the given device.

The *device-name* can be

COM1	Communications port 1
CON	Console

If "COM1" is used, the port's baud rate and other modes must be properly set for the attached terminal.

The **Redirection** commands are typically used to debug programs that require full use of the console screen.

Note

If input is redirected to COM1, the CNTRL-S and CNTRL-C keys are unavailable and will be ignored.

Examples

```
>COM1
```

This example redirects SYMDEB command output to the "COM1" device.

```
=COM1
```

This example redirects command input and output to "COM1."

4.36 Register Command

Syntax

R [*register-name* [*value*]]

The **Register** command displays the contents of CPU registers and allows the contents to be changed to new values.

If no *register-name* is given, the command displays all registers, flags, and the instruction at the address pointed to by the current CS and IP register values.

If a *register-name* is given, the command displays the current value of the given register and prompts for a new value. If both a *register-name* and *value* are given, the command changes the register to the given value.

The *register-name* can be any one of the following names:

AX	BP	SS
BX	SI	CS
CX	DI	IP
DX	DS	PC
SP	ES	F

IP and PC name the same register: the Instruction Pointer. F is a special name for the Flags register.

To change a register value, supply the name of the register when you enter the **Register** command. If you do not also supply a value, the command displays the name of the register, its current value, and a colon prompt. Type the new value and press the RETURN key. If you do not want to change the value, just press the RETURN key. If you enter a illegal register name, the command displays a "Bad Register!" message.

To change a flag value, supply the register name "F" when you enter the **Register** command. The command displays the current value of each flag as a two-letter name. The following is a list of flag values:

FLAG	SET	CLEAR
Overflow	OV	NV
Direction	DN (Decrement)	UP (Increment)
Interrupt	EI (Enabled)	DI (Disabled)
Sign	NG (Negative)	PL (Plus)
Zero	ZR	NZ
Auxiliary Carry	AC	NA
Parity	PE (Even)	PO (Odd)
Carry	CY	NC

At the end of the list of values, the command displays a hyphen (-). Once you see the hyphen, enter new values for the flags you wish to change, then press the RETURN key. You can enter flag values in any order. Spaces between values are not required. Flags for which new values are not entered remain unchanged. If you do not want to change any flags, just press the RETURN key.

If more than one value is entered for a flag, the command displays a "Double flag!" message. If you enter a name other than those shown above, the command returns a "Bad Flag!" message. In both cases, the flags up to the error are changed; flags at and after the error are not.

Examples

```
R
```

This example displays all register and flag values, as well as the instruction at the address pointed to by the CS and IP registers. The display should look like this:

```
AX=0E00 BX=00FF CX=0007 DX=01FF SP=039D BP=0000
SI=005C DI=0000 DS=04BA ES=04BA SS=04BA CS=04BA
IP=011A  NV UP DI NG NZ AC PE NC
04BA:011A  CD21           INT     21
```

The instruction is shown last.

```
R AX
```

This example displays the current value of the AX register and prompts for a new value. The display should look like this:

```
AX 0E00
:
```

You can type any 16-bit value after the colon (:). Press the RETURN key if you do not want to make a change.

```
R F
```

This example displays the current flags values and prompts for changes. The display should look like this:

```
NV UP DI NG NZ AC PE NC -
```

You must use the prompt method to change flag values; any value in the command line is ignored.

```
R IP 100
```

This example changes the IP register to the value 100.

4.37 Search Command

Syntax

S *range list*

The **Search** command searches the given *range* of memory locations for the byte values given in *list*. If the bytes in the list are found, the command displays the addresses of each occurrence of the list. Otherwise, it displays nothing.

The *list* can have any number of bytes. Each must be separated by a space or comma. If the list contains more than one byte, **Search** does not display an address unless the bytes beginning at that address exactly match the value and order of the bytes in the list.

Examples

```
S CS:100 200 41
```

This example displays the address of each memory location in the range CS:100 to CS:200 containing the byte value 41.

```
S table L 100 "Fixup"
```

This example displays the address of each memory location containing the string "Fixup". The command searches the first 256 (100H) bytes at the address given by "table."

4.38 Set Source Mode Command

Syntax

S - | & | +

The **Set Source Mode** command sets the display mode for commands that display instruction code. If the plus sign (+) is given, SYMDEB displays the actual program source line corresponding to the instruction to be displayed. If the minus sign (–) is given, SYMDEB disassembles and displays the instruction code in memory. If the ampersand (&) is given, SYMDEB displays both the actual program source line and the disassembled code.

The **Set Source Mode** command affects instructions displayed by the **Register**, **Trace**, **PTrace**, and **Unassemble** commands. Initially, SYMDEB displays disassembled instruction code only. SYMDEB displays source lines only if the **S+** or **S&** command has been given and a symbol file containing line number information has been loaded. If no symbol file is loaded, or the symbol file does not contain line number information, SYMDEB ignores subsequent requests to display source lines. If the **S&** command is given, SYMDEB displays source lines only when the current instruction address specified by CS:IP matches a line number.

Source lines have the form

linenumber:source

Source lines are always displayed before any disassembled instructions are displayed. If SYMDEB must change the current source file to display a requested line, it displays the name of the new source file before displaying the line.

Note

Although SYMDEB uses the line number information in symbol files to display source lines, it may need to prompt for the actual filename of the source file before these lines can be displayed. Whenever SYMDEB must access a source file for the first time, it displays the prompt

```
External file name for mapname (cr for none)?
```

where *mapname* is the filename of the symbol file. To display source lines, you must type the name of the corresponding source file. The filename must include the filename extension. If SYMDEB cannot find the named file, it prompts for a new name.

At times, you may wish to suppress display of source lines. In such cases, just press the ENTER key when SYMDEB prompts for the filename. SYMDEB will suppress the actual source lines and display a mapname and line number instead.

Examples

```
S+
```

This example sets SYMDEB to source line display mode. On subsequent commands, SYMDEB displays instruction source lines only.

```
S&
```

This example sets SYMDEB to source line and disassembly display mode. On the subsequent commands, SYMDEB displays both the source line and disassembled instruction code.

4.39 Trace Command

Syntax

T [=*start-address*] [*count*]

The **Trace** command executes the instruction at the given *start-address*, then displays the current values of all registers and flags. The display has the same format as the **Register** command.

If the optional *start-address* is given, the command starts execution at the given address. Otherwise, it starts execution at the instruction pointed to by the current CS and IP registers. The equal sign (=) is required if a *start-address* is given.

If the optional *count* is given, the command continues to execute *count* instructions before stopping. The command displays the current values of the registers and flags for each instruction before executing the next.

Notes

The **Trace** command uses the hardware trace mode of the 8086, 8088, 186, or 286 microprocessor. Consequently, you may also trace instructions stored in ROM (Read Only Memory).

Examples

```
T =_main
```

This example executes the instruction at _main, then displays the current values of the registers and flags. It also displays the next instruction to be executed.

```
T
```

This example executes the instruction pointed to by the current CS and IP register values.

```
T =011A 10
```

This example executes sixteen (10H) instructions beginning at 011A in the current CS segment. The command displays the current register and flags values 16 times. It also displays each instruction being traced.

4.40 Unassemble Command

Syntax

U [*range*]

The **Unassemble** command displays the instructions and/or statements of the program being debugged. The format of the display depends on the current display mode set by the **Set Source Mode** command. The display mode can be:

Mode	Meaning
Disassembly (S-)	Display disassembled instruction code. SYMDEB reads memory bytes from the addresses given by *range* and translates these bytes into assembly language statements. The resulting statements have the same format as defined for the **Assemble** command. If a symbol map has been loaded with the program, operands that represent public labels, variables, or absolute symbols are displayed by name instead of address.
Source (S+)	Display lines from the program's actual source file. SYMDEB displays the source lines that correspond to the instructions in the given *range*. No disassembly is performed.
Mixed (S&)	Display source lines and disassembled instructions. SYMDEB displays one source line for each corresponding group of assembly language statements. Source lines are read from the source file. Assembly language statements are translated from memory bytes.

For both Source and Mixed modes, SYMDEB requires that a symbol map be loaded with the program and that line number information for the source file is in the map. If no line number information exists for a given portion of a program, SYMDEB will display nothing.

If the optional *range* is given, the command displays instructions generated from code within the given range. If no *range* is given, the command displays the instructions generated from the first 8 lines of code at the current unassemble address. The current unassemble address is simply the address of the first byte (line) after the last byte (line) displayed by the previous **Unassemble** command.

When using Disassembly and Mixed mode, SYMDEB displays both the hexadecimal and ASCII value of 8-bit immediate operands. The hexadecimal value is shown as part of the instruction; the ASCII value is shown immediately after a semicolon (;) on the same display line.

Only 8086 and 8087 mnemonics can be displayed.

Examples

Disassembly Mode.

```
U CS:02AD
```

This example disassembles 8 lines of disassembled code beginning at the address "CS:02AD." The display should look like this:

```
1156:02AD 55              PUSH BP
1156:02AE 8BEC            MOV  BP,SP
1156:02B0 B80200          MOV  AX,0002
1156:02B3 E893FF          CALL chkstk
1156:02B6 C746FE6100      MOV  Word Ptr [BP-02],0061
1156:02BB FF0EEC05        DEC  Word Ptr [05EC]
1156:02BF 833EEC0500      CMP  Word Ptr [05EC],+00
1156:02C4 7C11            JL   02D7
```

```
U _main L 0A
```

This example displays 10 (0A hexadecimal) lines of disassembled code at the address "_main." The display should look like this:

```
IGROUP:_main:
1156:02AD 55              PUSH  BP
1156:02AE 8BEC            MOV   BP,SP
1156:02B0 B80200          MOV   AX,0002
1156:02B3 E893FF          CALL  chkstk
1156:02B6 C746FE6100      MOV   Word Ptr [BP-02],0061
1156:02BB FF0EEC05        DEC   Word Ptr [05EC]
1156:02BF 833EEC0500      CMP   Word Ptr [05EC],+00
1156:02C4 7C11            JL    _main+2A (02D7)
1156:02C6 8A46FE          MOV   AL,[BP-02]
1156:02C9 8B1EEA05        MOV   BX,[05EA]
```

Source Mode.

```
U CS:02AD
```

This example displays 8 source lines from the program source file. These lines correspond to the instruction code beginning at the address "CS:02AD." The display should look like this:

```
4:{
5:      int i;
6:
7:      for (i='a'; i<'z'; i++)
8:              putchar(i);
9:      for (i='A'; i<'z'; i++)
10:             putchar(i);
11:     for (i='0'; i<'9'; i++)
```

```
U _main L 5
```

This example displays 5 source lines beginning at the address "_main." The display should look like this:

```
4:{
5:      int i;
6:
7:      for (i='a'; i<'z'; i++)
8:              putchar(i);
```

Mixed Mode.

```
U CS:02AD
```

This example displays 8 lines of disassembled instruction code and program source code beginning at "CS:02AD." The display should look like this:

```
4:{
IGROUP:_main:
1156:02AD 55              PUSH  BP
1156:02AE 8BEC            MOV   BP,SP
1156:02B0 B80200          MOV   AX,0002
1156:02B3 E893FF          CALL  chkstk
7:      for (i='a'; i<'z'; i++)
1156:02B6 C746FE6100      MOV   Word Ptr [BP-02],0061
```

```
U _main L 5
```

This example displays 5 lines of disassembled instruction code and program source beginning at the address "_main." The display should look like this:

```
4:{
IGROUP:_main:
1156:02AD 55              PUSH  BP
1156:02AE 8BEC            MOV   BP,SP
1156:02B0 B80200          MOV   AX,0002
```

4.41 Write Command

Syntax

W [*address*[*drive record count*]]

The **Write** command writes the contents of a given memory location to a named file, or to a given logical record on disk.

To write to a file, the filename must be set with a **Name** command, then the BX:CX register pair must be set to the number of bytes to be written with a **Register** command. If no *address* is given, the command copies bytes starting from the address CS:100, where CS is the current value of the CS register. If *address* is given, the com-

mand copies bytes starting at that address.

Note

The filename, length, and starting address for a loaded file must be set to correct values if any **Go**, **Ptrace**, or **Trace** command has been executed. The BX and CX registers must also be set to correct values if they have been modified using a **Register** command.

To write to a logical record on disk, the *address*, *drive*, *record*, and *count* must be given. The *drive* must name the drive to be written to. It can be any number in the range 0 to 3, representing drives A: (0), B: (1), C: (2), and D: (3). The *record* names the first logical record to receive the data. It can be any 1- to 4-digit hexadecimal number. The *count* specifies the number of records to write to the disk. It can be any 1- to 4-digit hexadecimal number.

WARNING

Do not write data to an absolute disk sector unless you are sure the sector is free. Writing to reserved or occupied sectors can destroy the contents of a file or even the entire disk.

Examples

```
N table.bin
R BX
BX 0100
: 0000
R CX
CX D43C
: 0100
W DGROUP:table
```

This example writes 256 (100H) bytes to the file named "table.bin" on disk. The bytes to be written start at the address "DGROUP:table."

```
W workspace 2 34 3
```

This example writes 3 logical records to drive C:, starting at record number 52 (34H). The bytes to be written start at the address "workspace."

4.42 Error Messages

SYMDEB displays an error message whenever it detects a command it cannot complete. SYMDEB displays the command that caused the error, followed by the message "Error". A caret (^) points to the approximate location of the error in the command line. For example, the following display shows an illegal range of values in the **Dump** command.

```
d cs:100 0
         ^ Error
```

At other times, SYMDEB may display other error messages to let you know more about the error. You can receive any of the following error messages. Each error terminates the SYMDEB command under which it occurred, but does not terminate SYMDEB itself.

ERROR CODE	DEFINITION
Bad Flag!	Bad flag. You attempted to alter a flag, but the characters typed were not one of the acceptable pairs of flag values. See the **Register** command for the list of acceptable flag entries.
Too many breakpoints!	You specified more than ten breakpoints as parameters to the **Go** command. Retype the **Go** command with ten or fewer breakpoints.
Bad register!	You typed the **Register** command with an invalid register name. See the **Register** command for the list of valid register names.
Double flag!	You typed two values for one flag. You may specify a flag value only once. See the **Register** command.
Breakpoint list or '*' expected!	You typed a **Breakpoint Clear**, **Breakpoint Disable**, or **Breakpoint Enable** command without giving a list of breakpoints to act on.
Error reading .SYM file!	You named a symbol file in the SYMDEB command line that cannot be read. The file may have 0 length or a disk error may have occurred.

4.43 SYMDEB-Compatible Assemblers and Compilers

The symbolic debugging features of SYMDEB/MAPSYM can be used with programs written in the following languages.

Microsoft FORTRAN version 3.0 and higher
Microsoft Pascal version 3.0 and higher
Microsoft C version 2.0 and higher
Microsoft Macro Assembler version 1.0 and higher
Microsoft BASIC Compiler version 1.0 and higher
Microsoft Business BASIC Compiler version 1.0 and higher

IBM Personal Computer FORTRAN version 2.0 and higher
IBM Personal Computer Pascal version 2.0 and higher
IBM Personal Computer Macro Assembler version 1.0 and higher
IBM Personal Computer BASIC Compiler version 1.0 and higher

The source line display features of SYMDEB/MAPSYM can be used by compilers which generate the proper line number information. The following compilers generate source line number information automatically.

Microsoft FORTRAN version 3.0 and higher
Microsoft Pascal version 3.0 and higher
IBM Personal Computer FORTRAN version 2.0 and higher
IBM Personal Computer Pascal version 2.0 and higher

The following compilers require command line switches to generate source line number information. Please refer to your language user's guide for the proper compiler switches.

Microsoft C version 2.0 and higher

Chapter 5
CREF:
A Cross-Reference Utility

5.1 Introduction

The Microsoft Cross-Reference Utility, CREF, creates a cross-reference listing of all symbols in your assembly language programs. A cross-reference listing is an alphabetical list of symbols in which each symbol is followed by a series of line numbers. The line numbers indicate the lines in the source program that contain a reference to the symbol.

CREF is intended to be used as a debugging aid to help speed up the search for symbols encountered during a debugging session. The cross-reference listing, together with the symbol table created by the assembler, can make debugging and correction of a program simpler and easier.

5.2 Using CREF

CREF creates a cross-reference listing for a program by converting a non-ASCII "cross-reference file," produced by the assembler, into a readable ASCII file. You create the cross-reference file by supplying a cross-reference filename when you invoke the assembler. You create the cross-reference listing by invoking CREF and supplying the name of the cross-reference file.

The following sections explain how to create a cross-reference file for CREF and how to start CREF for creation of a cross-reference listing.

5.2.1 Creating a Cross-Reference File

You can create a cross-reference file by supplying a cross-reference filename when you invoke MASM. MASM offers two ways to name this file: on the command line with other filenames, or in response to a command prompt.

To create a cross-reference file from a command line, place the name as the fourth parameter in the MASM command line. For example, to create a cross-reference file "file.crf" for the program "file.asm," type

```
MASM  file.asm,file.obj,file.lst,file.crf
```

This command also creates object and source listing files for the program. MASM parameters must be separated by commas. Even if you do not supply a name for a given parameter, you still must supply a comma.

To create a cross-reference file using a prompt, invoke MASM, then supply the filename in response to the fourth command prompt. For example, to create a cross-reference file "file.crf" for the program "file.asm," type

```
MASM

Source filename [.ASM]: file.asm
Object filename [file.OBJ]: file.obj
Source listing  [NUL.LST]: file.lst
Cross reference [NUL.CRF]: file.crf
```

If you do not type a filename in response to this prompt, MASM will not create a cross-reference file. If you type only a filename (no extension), MASM uses the extension .CRF by default. This is the extension expected by CREF and is recommended for all cross-reference files.

5.2.2 Creating a Listing From a Command Line

You can start CREF and create a cross-reference listing by typing the CREF command name, the name of the cross-reference file, and the name of the listing file you wish to create all on one command line. The command line has the form

CREF *crf-file* , *ref-file*

where *crf-file* is the name of the cross-reference file created by MASM, and *ref-file* is the name of the readable ASCII file you wish to create.

If you do not supply filename extensions when you name the files, CREF will automatically provide default extensions. It uses the extension .CRF for the *crf-file* and .REF for *ref-file*. If you do not want these extensions, you must supply your own.

You can select a default filename for the listing file by typing a semicolon immediately after *crf-file*. The default filename has the same filename as *crf-file*, but uses the extension .REF instead of .CRF.

You can specify the directory or disk drive into which CREF will place the cross-reference listing by supplying an appropriate pathname or device name for *ref-file*. You can also name output devices such as CON: and LPT:.

Examples

```
CREF  file.crf,file.ref
```

This example uses the cross-reference file "file.crf" to create a cross-reference listing "file.ref."

```
CREF  file,file
```

This example uses the cross-reference file "file.crf" to create a cross-reference listing "file.ref." CREF supplies default filename extensions since no extensions are given.

```
CREF  file.crs,file.rf
```

This example uses the cross-reference file "file.crs" to create a cross-reference listing "file.rf".

```
CREF  file.crf;
```

This example causes MS-CREF to create the cross-reference listing "file.ref."

```
CREF  file,con:
```

This example displays the cross-reference listing at the console.

5.2.3 Creating a Listing From Prompts

You can direct CREF to prompt you for filenames when it starts by typing just the CREF command name. If you start the program in this way, CREF displays a series of prompt messages that direct you to enter filenames.

To start CREF with prompts, follow these steps:

1. Make sure that MS-DOS is displaying its prompt.

2. Type

   ```
   CREF
   ```

   and press the RETURN key. Once CREF starts, it displays the prompt

   ```
   Cross reference [.CRF]:
   ```

3. Type the name of the cross-reference file that you wish to convert to a cross-reference listing, then press the RETURN key. You do not have to supply a filename extension if your cross-reference file has the extension .CRF. If your file does not have this extension, you must supply the correct extension.

   Once you supply a filename, CREF displays the prompt

   ```
   Listing [filename.REF]:
   ```

   where *filename.REF* is the default filename for the cross-reference listing.

4. Press the RETURN key if you wish to use the default name for the cross-reference listing. Otherwise, type the filename that you want and then press the RETURN key. If you do not supply a filename extension, CREF uses .REF by default.

Once you have supplied the filenames, CREF reads the cross-reference file and creates the new listing.

Notes

You can specify the disk drive or device to which CREF will place the cross-reference listing by supplying an appropriate drive or device name.

Typing a semicolon (;) for the listing filename causes CREF to use the default filename.

You can correct typing errors on a line by using the BACKSPACE key. You can abort CREF by pressing the CNTRL-C key.

5.3 Cross-Reference Listing Format

The cross-reference listing contains the name of each symbol defined in your program followed by a list of line numbers. Each line number represents the line in your program in which a symbol is defined or used. The line number in which a symbol is defined is marked with a pound sign (#). Each page in the listing begins with the title of the program. The title is the name or string defined by the TITLE directive in your program's source file.

For example, assume that the following source program is in the file "test.asm:"

```
        TITLE A Test File.

        public start
        extrn  print:near, exit:near

DATA    segment
        assume  ds:DATA
string  db      "A test file.", 0dH, 0aH, 0
DATA    ends

CODE    segment
        assume  cs:CODE
start   proc    near
        mov     ax, DATA
        mov     ds, ax
        mov     ax, offset string
        push    ax
        call    print
        add     sp, 2
        call    exit
start   endp
CODE    ends

        end  start
```

To assemble this program and create a cross-reference file, you can type

```
MASM test.asm, test.obj, test.lst, test.crf
```

The cross-reference file is named "test.crf."

To create a cross-reference listing of this file, you can type

```
CREF test.crf, test.ref
```

The resulting cross-reference listing in the file "test.ref" should ook like this:

```
 A Test File.

 Symbol Cross Reference                (# is definition)    Cref-1

CODE . . . . . .                13#     14      24

DATA . . . . . .                6#      7       11      16

EXIT . . . . . .                4#      22

PRINT . . . . . .               4#      20

START . . . . . .               3       15#     23      26
STRING . . . . . .              8#      18
```

5.4 Error Messages

CREF displays an error message and terminates operation whenever it encounters an error. Control is then returned to the operating system. All error messages have the form

Fatal I/O Error *error-number*
in: *filename*

where *error-number* is one of the numbers listed below, and *filename* is the name of the cross-reference file you supplied.

The following table lists all error numbers and explains the cause and possible remedy of the error:

Error Number	Description
101	Hard data or device name error. This is an unrecoverable disk I/O error. Make sure your cross-reference filename is correct and that you have given the correct drive or device name.
105	Device offline. Make sure that the disk drive door is closed, that the printer is attached, or that the device named for the cross-reference file or cross-reference listing file is online.
108	Disk full. You must make enough room on the given disk for the cross-reference listing file.
110	File not found. Make sure you have typed the cross-reference and cross-reference listing filenames correctly.
111	Disk is write-protected. You cannot create a listing file on a write-protected disk.
103, 104, 106, 112 thru 115	Internal error. Please report to Microsoft Corporation.

Chapter 6
LIB: A Library Manager

6.1 Introduction

The Microsoft Library Manager, LIB, creates, organizes, and maintains program libraries. A program library is a collection of one or more "object modules." Object modules are assembled or compiled instructions and data that are ready for linking. A library stores object modules that other programs may need for execution. Libraries are used by the program linker, LINK, to resolve references to routines and variables used but not defined in a program.

LIB creates a library by copying the contents of one or more "object files" into a library file. An object file contains a single object module, created by MASM or a high-level language compiler. When LIB adds an object module to a library, it places the module's name in the library's table of contents. When LINK searches the library for the names of routines and variables used in a program, it checks the table of contents. When it finds the routine, it extracts a copy of the module containing that routine and links the module to the program. Thus, only the modules that contain routines or variables used by the program are extracted and linked.

This chapter explains how to create libraries, how to organize libraries to make LINK searches more efficient, and how to modify libraries by adding or replacing object modules.

6.2 Starting and Using LIB

This section explains how to start and use LIB to create and maintain libraries. You can use LIB to work on libraries in three different ways: through the MS-DOS command line, in response to LIB prompts, and through responses in a response file.

Once LIB starts, it either carries out the commands you have supplied, or prompts for additional commands. You can stop LIB at any time by pressing the CNTRL-C key.

6.2.1 Starting LIB With a Command Line

You can start LIB and name all the commands and files to be processed on a single MS-DOS command line. The LIB command line has the form

LIB *library* [/PAGESIZE:*n*] *commands...* [, *list-file*, *output-file*]

The *library* names the library file to be worked on. If you do not supply a filename extension, LIB uses .LIB by default. The /PAGESIZE option defines the page size of the library. The default is 16.

The *commands* are one or more LIB commands. They specify what tasks to carry out on the given library.

The optional *list-file* is the name of the cross-reference listing file you wish to make. If no filename is given, LIB does not create a listing file.

The optional *output-file* is the name of the new library file you wish to copy the modified library to. If no filename is given, LIB uses the input library name.

If a file is in another directory or on a different device, you must supply an appropriate pathname or device name.

If you give a listing filename, you must separate it from the last command with a comma (,). If you give an output filename, you must separate it from the listing file name with a comma (,) or from the last command with two commas (,,).

You can use a semicolon after any entry but the first to direct LIB to use the default responses for the remaining entries. The semicolon should be the last character on the command line.

Examples

```
LIB lang -+heap;
```

This example instructs LIB to replace the module "heap" in the library "lang.lib." The semicolon at the end of the command line tells LIB to use the default responses for the listing file and output file. This means that no listing file is created and that the changes are written back to the original library file instead of creating a new library file.

```
LIB lang -+heap,lang.lst,lang1.lib
```

This example creates a new library named "lang1.lib" by modifying the library "lang.lib." The new library is identical to the old one except that the module "heap" has been replaced. LIB also creates a listing file for the library named "lang.lst."

6.2.2 Starting LIB With Prompts

You can let LIB prompt you for the information it needs by typing just the command name at the MS-DOS command level. Follow these steps:

1. Type

   ```
   LIB
   ```

   and press the RETURN key. LIB starts and displays the prompt

   ```
   Library name:
   ```

2. Type the name of the library you wish to work on. If you do not supply a filename extension, LIB supplies .LIB by default. If you wish to create a new library, type the new name. Once you have typed the name, press the RETURN key.

   LIB now examines the library name and either displays the next prompt or asks for permission to create the new library. If LIB must create the library file, it displays the prompt

   ```
   Library file does not exist.  Create?
   ```

   Type "yes" to create the library file. Type "no" to return to the MS-DOS command level.

   Once the library is ready for work, LIB displays the prompt

   ```
   Operations:
   ```

3. Type the command or commands you wish to carry out on the given library and press the RETURN key. If you have more commands than can fit on one line, type an ampersand (&) as the last character on the line and press the RETURN key. LIB prompts for more commands.

   Once you have typed all commands, press the RETURN key. If you only want LIB to check the consistency of the library, do not type any commands—just press the RETURN key. Once you have pressed the RETURN key, LIB displays the prompt

   ```
   List file:
   ```

4. Type the name of the new cross-reference listing file and press the RETURN key. Make sure the filename is exactly as you want it. LIB will not provide a default filename extension. If you do not want a cross-reference listing file, do not type a name—just press the RETURN key.

   Once you have pressed the RETURN key, LIB displays the following prompt only if you have given commands that modify the library:

   ```
   Output library:
   ```

5. Type the name of the output file you wish to create. If you do not supply a filename extension, LIB supplies .LIB by default. If you do not give any filename, LIB uses the name of the current library. In this case, LIB saves a backup copy of the current library by replacing its .LIB extension with .BAK.

   Press the RETURN key when you are ready.

LIB now carries out the commands you have requested.

Notes

You can direct LIB to select the default responses to all remaining prompts by typing a semicolon (;) at any prompt line other than the first. When LIB encounters a semicolon, it immediately chooses the default responses and carries out any given commands.

You must supply a pathname or device name for any file that is in another directory or on another disk.

You can set the library page size when you name the library file. See Section 6.2.6, "Setting the Library Page Size."

Example

```
LIB
Library File: math
Operations: +sin +cos &
Operations: +atan +exp
List file:
Output library: math1
```

This example adds the modules in the object files "sin.obj" "cos.obj" "atan.obj" and "exp.obj" to a new library named "math1.lib." The rest of the library is identical to the contents of the library "math.lib."

6.2.3 Starting LIB With a Response File

You can direct LIB to read commands and filenames from a response file by supplying the name of the response file when you invoke LIB. The command line has the form

LIB @*response-file*

The *response-file* is the name of the response file to be read. It must be preceded by an at sign (@). A pathname or device name must be given if the file is in another directory or on a different drive.

You can name the response file anything you like. The file has the general form

library[/PAGESIZE: n]
[*commands*]
[*list-file*]
[*output-file*]

Each filename must be put on a separate line. Any number of commands can be placed on a line. If you have more commands than can fit on one line, you can extend the line by typing an ampersand (&) at the end of the line.

When you run LIB with a response file, it displays each response file line as it processes it. If the response file does not contain answers for all the prompts, LIB prompts you for the missing names. You can use a semicolon anywhere in the response file to cause LIB to select default filenames for the remaining prompts.

Example

```
PLib
+cursor +heap -heap *stack
cross.lst
```

This response file causes LIB to work on the library "PLib.lib." The commands add the module "cursor," replace the module "heap" and copies the module "stack" to a new object file.

6.2.4 Creating a New Library

You can create a new library by simply giving the name of the new library file when you invoke LIB. The name of the new library must not be the name of an existing file, or LIB will assume you want to modify the existing file. When you give the name of a library file that does not currently exist, LIB looks for the file, then displays the prompt

```
Library file does not exist.  Create?
```

Type "yes" to create the file. Type "no" to abort the library session.

You can specify a page size when you create the library. The default page size is 16 bytes. See Section 6.2.6 "Setting the Library Page Size."

6.2.5 Checking a Library's Consistency

You can check that a library's contents are consistent and usable by running LIB without commands. Simply type LIB, the name of the library you wish to check, and a semicolon. LIB then makes sure that all entries in the library can be accessed. If any problems are discovered, LIB displays an error message. Otherwise, it displays nothing.

For example, the command

```
LIB math;
```

carries out a consistency check on the library "math.lib."

Consistency checks are typically used to verify that the contents of existing libraries are usable. For example, if you copied a library from another disk, you can run a consistency check to verify that the copied library is intact.

Note that LIB automatically checks object modules for consistency before adding them to the library, so you do not need to check the library each time you add a module.

6.2.6 Setting the Library Page Size

You can set the library page size by adding a page size option after the library filename in the LIB command line. The command line has the form

LIB *library-name*/PAGESIZE:*n*

The *n* specifies the new page size. It must be an integer value representing a power of 2 between the values 16 and 32,768.

The page size of a library affects the alignment of modules stored in the library. Modules in the library are aligned to always start at a

position that is a multiple of the page size (in bytes) from the beginning of the file. The default page size is 16 for a new library or the current page size for an existing library.

Note

Because of the indexing technique used by LIB, a library with a large page size can hold more modules than a library with a smaller page size. However, for each module in the library, an average of $n/2$ bytes of storage space is wasted (where n is the page size.) In most cases, a small page size is advantageous; you should use the smallest page size possible.

Example

```
LIB math/PAGESIZE:256 ;
```

This example creates a library named "math.lib" whose page size is 256 bytes.

6.2.7 Creating a Cross-Reference Listing

You direct LIB to create a cross-reference listing whenever you give a listing filename in a LIB command line. A cross-reference listing file contains two lists: a list of all public symbols in the library, and a list of all modules in the library.

In the first list, all symbols are listed alphabetically. Each symbol name is followed by the name of the module in which it is referenced. The list has the form

```
START   ..... main
SUM     ..... add
SUM2    ..... add
EXIT    ..... error
```

In the second list, all modules are listed alphabetically. The module name is followed by an alphabetical listing of the public symbols referenced in that module. The list has the form

```
main   Offset: 00000200H    Code and data size: 20H
    START

add    Offset: 00000400H    Code and data size: 20H
    SUM                SUM2

error  Offset: 00000600H    Code and data size: CH
    EXIT
```

6.3 Using LIB Commands

The LIB commands specify the tasks to be carried out on a given library. The commands add, delete, and replace modules in a given library. They also copy and move modules to new libraries.

Command	Symbol
Add	+
Delete	–
Replace	–+
Copy	*
Move	–*

Commands can be given on the LIB command line or in response to LIB's Operations prompt.

6.3.1 Adding a Module to a Library

Syntax

+object-file

The Add (+) command adds the object module in the given *object-file* to the current library. The *object-file* must be the filename of an object file. If you do not give a filename extension, LIB supplies .OBJ by default. If the file is in another directory or on a different disk, you must supply an appropriate pathname or device name. There must be no spaces between the plus sign (+) and the name.

LIB searches for the file you have named, and adds the file's contents to the current library. LIB then strips the drive name, pathname, and the filename extension (if any) from the object filename and places the resulting name in the library's table of contents.

LIB always adds object modules to the end of the library file.

Examples

```
LIB math +sin.obj
```

This example adds the module in the file "sin.obj" to the library "math.lib."

```
LIB \lib\math +cos, list;
```

This example adds the module in the file "cos.obj" to the library "math.lib" in the \lib directory.

```
LIB math +A:\src\atan ;
```

This example adds the module in the file "atan.obj" to the library "math.lib." The object file is in the \src directory on drive A:.

6.3.2 Deleting Library Modules

Syntax

–module-name

The Delete (–) command deletes the object module named *module-name* from the current library. The *module-name* must be the name of the module you wish to delete. It must be spelled exactly as it appears it the library's table of contents.

Note

LIB carries out all Delete commands before attempting to carry out any Add commands, regardless of the order in which the commands appear in the command line. This order of execution prevents confusion in LIB when a new version of a module replaces an existing version in the library file.

Examples

```
LIB math -sin
```

This example deletes the module "sin" from the library "math.lib."

```
LIB \lib\math -cos, list;
```

This example deletes the module "cos" from the library "math.lib" in the \lib directory.

```
LIB math +A:\src\atan -atan ;
```

This example deletes the module "atan.obj" from the library "math.lib." It then adds the module in the object file "A:\src\atan.obj" to the library. Note that the Delete command is carried out before the Add command.

6.3.3 Replacing Library Modules

Syntax

-+module-name

The Replace (-+) command replaces the module *module-name* with the module in an object file having the same name. The *module-name* must have exactly the same spelling as the name in the library's table of contents. LIB first deletes this module, then searches the current working directory for a file having the same name and the filename extension .OBJ.

If LIB cannot find the file containing the replacement module, it displays an error message.

Example

```
LIB  math  -+cos;
```

This example deletes the module "cos.obj" then adds the contents of the file "cos.obj" to the library.

6.3.4 Copying Library Modules

Syntax

**module-name*

The Copy command extracts a copy of the module given by *module-name* to an object file having the same name. The *module-name* must have exactly the same spelling as the name in the library's table of contents.

When LIB copies the module to an object file, it creates a file whose filename is the same as the module, but whose filename extension is .OBJ. The file is placed in the current working directory.

Example

```
LIB math *cos ;
```

This example creates a file named "cos.obj" in the current working directory. The file contains the object module copied from the "math.lib" library.

6.3.5 Moving Library Modules

Syntax

*–*module-name*

The Move (–*) command moves the module given by *module-name* from the current library to an object file having the same name as the module. The *module-name* must be spelled exactly as it appears in the library's table of contents.

The move is equivalent to copying the module to an object file, as described above, then deleting the module from the library.

Example

```
LIB math -*cos
```

This example moves the module "cos" into an object file named "cos.obj" in the current working directory. The module is deleted from the library "math."

6.3.6 Combining Libraries

Syntax

+library-name

The Add (+) command can also be used to add the contents of one library to the current library. The *library-name* must be the name of the library file you wish to add. You must give the filename extension of the file, otherwise, LIB assumes the file is an object file.

LIB adds the modules of the named library to the end of the current library without destroying the named library or deleting any modules.

Note

LIB can be used to add the contents of XENIX™ and Intel-style libraries to MS-DOS libraries.

Example

```
LIB math1 +math.lib;
```

This example adds the modules contained in the library "math.lib" to the modules in the library "math1.lib."

Chapter 7
MAKE:
A Program Maintainer

7.1 Introduction

The Microsoft Program Maintainer, MAKE, automates the process required to maintain assembly and high-level language programs. MAKE automatically carries out all tasks needed to update a program after one or more of its source files have changed.

Unlike other batch processing programs, MAKE compares the last modification date of the file or files that may need updating with the modification dates of files on which these target files depend. MAKE then carries out the given task only if a target file is out-of-date. MAKE does not assemble, compile, and link all files just because one file has been updated. This can save much time when creating programs that have many source files or take several steps to complete.

The following sections explain how to use MAKE and illustrate how to maintain a sample assembly language program.

7.1.1 Using MAKE

To use MAKE, you must create a make description file that defines the tasks you wish to accomplish and the files on which these tasks depend. Once the description file exists, you simply supply the filename when you invoke MAKE. MAKE then reads the contents of the file and carries out the requested tasks.

The following sections explain how to create a make description file and how to start MAKE.

7.1.2 Creating a Make Description File

You can create a make description file by using a text editor. A make description file consists of one or more target descriptions. Each description has the general form

target-file: *dependent-file...*
    command...

where *target-file* is the name of a file that may need updating, *dependent-file* is the name of a file on which the target file depends, and *command* is an external MS-DOS command.

The *target-file* and *dependent-file* must be valid file specifications. The specifications must include pathnames (or drive names) if the files are not in the same directory (or on the same drive) as the description file.

Any number of dependent files can be given, but only one target name is allowed. Dependent filenames must be separated with at least one space. If you have more dependent files than can fit on one line, you can continue the names on the next line by typing a backslash (\) followed by a new line.

The *command* can be any external MS-DOS command. Internal commands, such as TYPE and COPY, are not allowed. Any number of commands can be given, but each must begin on a new line and must be preceded by a TAB or spaces. The commands are carried out only if one or more of the dependent files have been modified since the target file was created.

You can give any number of target descriptions in a description file. You must make sure, however, that the last line in one description is separated from the first line of the next by at least one blank line.

Note

The order in which you place the target descriptions is important. MAKE examines each description in turn and makes its decision to carry out a given task based on the file's current modification date. If a command in a later description modifies a file, MAKE has no way to return to the description in which that file is a target.

Example

```
startup.obj:  startup.asm
        MASM startup,startup,nul,nul

print.obj:    print.asm
        MASM print,print,print,print

print.ref:    print.crf
        CREF print,print

print.exe:    startup.obj print.obj \lib\syscal.lib
        LINK startup+print,print,print/map,\lib\syscal;

print.sym:    print.map
        MAPSYM -l print.map
```

This example defines the actions to be carried out to create five target files. Each file has at least one dependent file and one command. The target descriptions are given in the order in which the target files will be created. Thus, startup.obj and print.obj are examined and created, if necessary, before print.exe.

7.1.3 Starting MAKE

The MAKE command line has the form

MAKE *filename*

where *filename* is the name of a make description file. A make description file, by convention, has the same filename (but with no extension) as the program it describes. Although any filename can be used, this convention is preferred.

Once you start MAKE, it examines each target description in turn. If a given target file is out-of-date with respect to its dependent file or if the file does not exist, MAKE executes the given command or commands. Otherwise, it skips to the next target description.

When MAKE finds an out-of-date target file, it displays the corresponding command or commands before executing them. If MAKE finds a file that does not exist, it displays a message of the form

make: *filename* – file not found

It displays this message even if the file is a target file and is created by subsequent commands.

When MAKE executes a command, it uses the same environment used to invoke MAKE. Thus, environment variables such as PATH are available for these commands.

Example

```
make test
```

This example directs MAKE to take its instructions from the make description file named "test".

7.2 Maintaining a Program: An Example

MAKE is especially useful for programs that are in development, because it offers a quick way to recreate a modified program after small changes. Consider a test program name "test.asm" that is being used to debug the routines in a library file named "math.lib." The purpose of "test.asm" is to call one or more routines in the library so a study of their interaction can be made. Each time "test.asm" is modified, it has to be assembled, a cross-reference listing has to be created, the assembled file has to be linked to the library, and, finally, a symbol file has to be created to use with SYMDEB. The following target descriptions copied to the description file named "test" will carry out all of these tasks:

```
test.obj:     test.asm
        MASM test,test,test,test

test.ref:     test.crf
        CREF test,test

test.exe:     test.obj \lib\math.lib
        LINK test,test,test/map,\lib\math

test.sym:     test.map
        MAPSYM -l test.map
```

These lines define the actions to be carried out to create four target files: "test.obj," "test.ref," "test.exe," and "test.sym." Each file has at least one dependent file and one command. The target descriptions are given in the order in which the target files will be created. Thus, test.sym depends on test.map which is created by LINK; test.exe depends on test.obj which is created by MASM; and test.ref depends on test.crf which is also created by MASM.

Once the description file is in place, you can create test.asm using a text editor, then invoke MAKE to create all other required files. The command line should be

```
MAKE test
```

MAKE carries out the following steps:

1. Compares the modification date of test.asm with test.exe. If test.exe is out of date (or does not exist), MAKE executes the command

   ```
   MASM test,test,test,test
   ```

   Otherwise, it skips to the next target description.

2. MAKE compares the dates of test.ref and test.crf. If test.ref is out of date, it executes the command

   ```
   CREF test, test
   ```

3. Compares test.exe with the dates of test.obj and the library file math.lib. If test.exe is out-of-date with respective to either file, MAKE executes the command

   ```
   LINK test,test,test/map,\lib\math.lib
   ```

4. Compares the test.sym and test.map. If out-of-date, it executes

   ```
   MAPSYM -l test.map
   ```

When test.asm is first created, MAKE will execute all commands, since none of the target files exist. If you invoke MAKE again without changing any of the dependent files, it will skip all commands. If you change the library file math.lib, but make no other

changes, MAKE will execute the LINK command, since test.exe is now out-of-date with respect to math.lib. It will also execute MAP-SYM, since test.map is created by LINK. No other commands are carried out.

Appendix A
Error Messages

A.1 Introduction

This appendix lists and explains the error messages that can be generated by the Macro Assembler, MASM, and the Linker LINK.

A.2 Macro Assembler Messages

This section lists and explains the messages displayed by the Macro Assembler, MASM. MASM displays a message whenever it encounters an error during processing. It displays a warning message whenever it encounters questionable statement syntax.

An end-of-assembly message is displayed at the end of processing, even if no errors occurred. The message contains a count of errors and warning messages it displayed during the assembly. The message has the form

This message is also copied to the source listing.

Error messages are divided into two categories: assembler errors and I/O handler errors. In each category, messages are listed in numerical order with a short explanation where necessary.

Assembler Errors

0: Block nesting error

Nested procedures, segments, structures, macros, IRC, IRP, or REPT are not properly terminated. An example of this error is closing an outer level of nesting with inner level(s) still open.

1: Extra characters on line

This occurs when sufficient information to define the instruction directive has been received on a line and superfluous characters beyond are received.

2: Register already defined
This will only occur if the assembler has internal logic errors.

3: Unknown symbol type
Symbol statement has something in the type field that is unrecognizable.

4: Redefinition of symbol
This error occurs on pass 2 and succeeding definitions of a symbol.

5: Symbol is multi-defined
This error occurs on a symbol that is later redefined.

6: Phase error between passes
The program has ambiguous instruction directives such that the location of a label in the program changed in value between pass 1 and pass 2 of the assembler. An example of this is a forward reference coded without a segment override where one is required. There would be an additional byte (the code segment override) generated in pass 2 causing the next label to change. You can use the /D option to produce a listing to aid in resolving phase errors between passes. See Chapter 2, "MASM: A Macro Assembler."

7: Already had ELSE clause
Attempt to define an ELSE clause within an existing ELSE clause (you cannot nest ELSE without nesting IF...ENDIF).

8: Not in conditional block
An ENDIF or ELSE is specified without a previous conditional assembly directive active.

9: Symbol not defined
A symbol is used that has no definition.

10: Syntax error
The syntax of the statement does not match any recognizable syntax.

11: Type illegal in context
The type specified is of an unacceptable size.

12: Should have been group name
Expecting a group name but something other than this was given.

13: Must be declared in pass 1
An item was referenced before it was defined in Pass 1. For example, "IF DEBUG" is illegal if DEBUG is not previously defined.

14: Symbol type usage illegal
Illegal use of a PUBLIC symbol.

15: Symbol already different kind
Attempt to define a symbol differently from a previous definition.

16: Symbol is reserved word
Attempt to use an assembler reserved word illegally. For example, to declare MOV as a variable.

17: Forward reference is illegal
Attempt to reference something before it is defined in pass 1.

18: Must be register
Register expected as operand but you furnished a symbol -- was not a register.

19: Wrong type of register
Directive or instruction expected one type of register, but another was specified. For example, INC CS.

20: Must be segment or group
Expecting segment or group and something else was specified.

21: Symbol has no segment
Trying to use a variable with SEG, and the variable has no known segment.

22: Must be symbol type
Must be WORD, DW, QW, BYTE, or TB but received something else.

23: Already defined locally
Tried to define a symbol as EXTERNAL that had already been defined locally.

24: Segment parameters are changed
List of arguments to SEGMENT were not identical to the first time this segment was used.

25: Not proper align/combine type
SEGMENT parameters are incorrect.

26: Reference to mult defined
The instruction references something that has been multi-defined.

27: Operand was expected
Assembler is expecting an operand but an operator was received.

28: Operator was expected
Assembler was expecting an operator but an operand was received.

29: Division by 0 or overflow
An expression is given that results in a divide by 0 or a number larger then can be represented.

30: Shift count is negative
A shift expression is generated that results in a negative shift count.

31: Operand types must match
Assembler gets different kinds or sizes of arguments in a case where they must match. For example, MOV.

32: Illegal use of external
Use of an external in some illegal manner. For example, DB M DUP(?) where M is declared external.

33: Must be record field name
Expecting a record field name but got something else.

34: Must be record or field name
Expecting a record name or field name and received something else.

35: Operand must have size
Expected operand to have a size, but it did not.

36: Must be var, label or constant
Expecting a variable, label, or constant but received something else.

37: Must be structure field name
Expecting a structure field name but received something else.

38: Left operand must have segment
Used something in right operand that required a segment in the left operand. (For example, ":.")

39: One operand must be const
This is an illegal use of the addition operator.

40: Operands must be same or 1 abs
Illegal use of the subtraction operator.

41: Normal type operand expected
Received STRUC, FIELDS, NAMES, BYTE, WORD, or DW when expecting a variable label.

42: Constant was expected
Expecting a constant and received an item that does not evaluate to a constant. For example, a variable name or external.

43: Operand must have segment
Illegal use of SEG directive.

44: Must be associated with data
Use of code related item where data related item was expected. For example, MOV AX,<code-label>.

45: Must be associated with code
Use of data related item where code item was expected.

46: Already have base register
Trying to double base register.

47: Already have index register
Trying to double index address.

48: Must be index or base register
Instruction requires a base or index register and some other register was specified in square brackets, [ ].

49: Illegal use of register
Use of a register with an instruction where there is no 8086 or 8088 instruction possible.

50: Value is out of range
Value is too large for expected use. For example, MOV AL,5000.

51: Operand not in IP segment
Access of operand is impossible because it is not in the current IP segment.

52: Improper operand type
Use of an operand such that the opcode cannot be generated.

53: Relative jump out of range
Relative jumps must be within the range -128 to +127 of the current instruction, and the specific jump is beyond this range.

54: Index displ. must be constant
Illegal use of index display.

55: Illegal register value
The register value specified does not fit into the "reg" field (the value is greater than 7).

56: No immediate mode
Immediate mode specified or an opcode that cannot accept the immediate. For example, PUSH.

57: Illegal size for item
Size of referenced item is illegal. For example, shift of a double word.

58: Byte register is illegal
Use of one of the byte registers in context where it is illegal. For example, "PUSH AL," is illegal.

59: CS register illegal usage
Trying to use the CS register illegally. For example, "XCHG CS,AX," is illegal.

60: Must be AX or AL
Specification of some register other than AX or AL where only these are acceptable. For example, the IN instruction.

61: Improper use of segment reg
Specification of a segment register where this is illegal. For example, an immediate move to a segment register.

62: No or unreachable CS
Trying to jump to a label that is unreachable.

63: Operand combination illegal
Specification of a two-operand instruction where the combination specified is illegal.

64: Near JMP/CALL to different CS
Attempt to do a NEAR jump or call to a location in a different CS ASSUME.

65: Label can't have seg. override
Illegal use of segment override.

66: Must have opcode after prefix
Use of a REPE, REPNE, REPZ, or REPNZ instructions without specifying any opcode after it.

67: Can't override ES segment
Trying to override the ES segment in an instruction where this override is not legal. For example, "STOS DS:TARGET" is illegal.

68: Can't reach with segment reg
There is no ASSUME that makes the variable reachable.

69: Must be in segment block
Attempt to generate code when not in a segment.

70: Can't use EVEN on BYTE segment
Segment was declared to be byte segment and attempt to use EVEN was made.

72: Illegal value for DUP count
DUP counts must be a constant that is not 0 or negative.

73: Symbol already external
Attempt to define a symbol as local that is already external.

74: DUP is too large for linker
Nesting of DUPs was such that too large a record was created for the linker.

75: Usage of ? (indeterminate) bad
Improper use of the "?". For example, ?+5.

76: More values than defined with
Too many initial values given when defining a variable using a REC or STRUC type.

77: Only initialize list legal
Attempt to use STRUC name without angle brackets, < >.

78: Directive illegal in STRUC
All statements within STRUC blocks must either be comments preceded by a semicolon (;), or one of the Define directives.

79: Override with DUP is illegal
In a STRUC initialization statement, you tried to use DUP in an override.

80: Field cannot be overridden
In a STRUC initialization statement, you tried to give a value to a field that cannot be overridden.

81: Override is of wrong type
In a STRUC initialization statement, you tried to use the wrong size on override. For example, 'HELLO' for DW field.

82: Register can't be forward ref

83: Circular chain of EQU aliases
An alias EQU eventually points to itself.

84: 8087 opcode can't be emulated
Either the 8087 opcode or the operands you used with it produce an instruction that the emulator cannot support.

85: Unexpected end of file
You forgot an end statement or there is a nesting error.

I/O Handler Errors

These error messages are generated by the I/O handlers. These messages have the form

MASM Error -- *error-message-text*
in: *filename*

The *filename* is the name of the file being handled when the error occurred. The *error-message-text* is one of the following messages:

101: Hard data
102: Device name
103: Operation
104: File system
105: Device offline
106: Lost file
107: File name
108: Device full
109: Unknown device
110: File not found
111: Protected file
112: File in use
113: File not open
114: Data format
115: Line too long

A.3 Linker Messages

This section lists the error messages that can occur when linking programs. The messages are in alphabetical order.

About to generate .EXE file.
Change diskette in drive A: and press ENTER.

This message appears before the .EXE has been written if the /P switch is given. Insert diskette that the .EXE file is to be written to into the specified drive (A: for example).

Ambiguous switch error: '*x*'

User did not enter a unique switch name prefix after the switch indicator /. For example, the command

```
A>LINK /N main;
```

will generate this error. LINK will abort.

Array element size mismatch

A far communal array has been declared with two or more different array element sizes (e.g., declared once as an array of characters and once as an array of reals). NOTE: At the present time, communal arrays are not available in MASM.

Attempt to put segment *name* in more than one group in file *filename*

A segment was declared to be a member of two different groups. Correct the source and recreate the object files.

Bad value for cparMaxAlloc

The number specified using the /CPARMAXALLOC switch does not lie in the range [1,65535].

Cannot find library: *filename*.lib. Enter new file spec:

The linker cannot find *filename*.lib and is requesting a new file name or a new path specification or both. The user should respond to the prompt with a new file name or a new path specification or both.

Cannot open list file

The directory or disk is full. Make space on the disk or in the directory.

Cannot open response file
: User names a response file the linker cannot open. User has probably made a typing mistake.

Cannot nest response files
: User names a response file within a response file. Fix response file.

Cannot open run file
: The directory or disk is full. Make space on the disk or in the directory.

Cannot open temporary file
: The directory or disk is full. Make space on the disk or in the directory.

Cannot reopen list file
: User did not actually replace the original diskette when asked to. Restart the linker.

Common area longer than 65536 bytes
: User's program has more than 64K of communal variables. NOTE: At the present time, only Microsoft C programs can possibly cause this message to be displayed.

Data record too large
: LEDATA record (in an object module) contains more than 1024 bytes of data. This is a translator error. Note the translator (compiler or assembler) that produced the incorrect object module and the circumstances under which it was produced, and report the information to Microsoft.

Dup record too large
: LIDATA record (in an object module) contains more than 512 bytes of data. Most likely, an assembly module contains a struc definition that is very complex, or a series of deeply nested DUP statements (e.g. ARRAY db 10 dup(11 dup (12 dup (13 dup (...))))). Simplify and reassemble.

filename is not a valid library
: The file specified as a library is invalid. LINK will abort.

Fixup overflow near *num* in segment *name* in *filename*(*name*) offset *num*

Some possible causes are: 1) A group is larger than 64K bytes, 2) the user's program contains an intersegment short jump or intersegment short call, 3) the user has a data item whose name conflicts with that of a subroutine in a library included in the link, and 4) the user has an EXTRN declaration inside the body of a segment, for example:

```
CODE   segment public 'code'
extrn  main:far
start  proc   far
       call   main
       ret
start  endp
CODE   ends
```

The following construction is preferred:

```
extrn  main:far
CODE   segment public 'code'
start  proc   far
       call   main
       ret
start  endp
CODE   ends
```

Revise the source and recreate the object file.

Incorrect DOS version, use DOS 2.0 or later

LINK will not run on pre-DOS 2.0, Reboot your system with DOS 2.0 or above, and try linking again.

Insufficient stack space

There is not enough memory to run the linker.

Interrupt number exceeds 255

A number greater than 255 has been given after the /OVERLAYINTERRUPT switch. Try again with a number in the range 4 to 255.

Invalid numeric switch specification
: Typographical error entering value for one of the linker switches, such as entering a character string for a switch that requires a numeric value. LINK will abort.

Invalid object module
: One of the object modules is invalid. Try recompiling. If the error persists, contact Microsoft.

NEAR/HUGE conflict
: Conflicting near and huge definitions for a communal variable. NOTE: At the present time, communal variables are not available in MASM.

Nested left parentheses
: User has made a typing mistake while specifying the contents of an overlay on the command line.

No object modules specified
: User failed to supply the linker with any object file names.

Out of space on list file
: Disk on which list file is being written is full. Free more space on the disk and try again.

Out of space on run file
: Disk on which .EXE is being written is full. Free more space on the disk and try again.

Out of space on scratch file
: Disk in default drive is full. Delete some files on that disk, or replace with another diskette, and restart the linker.

Overlay manager symbol already defined: *name*
: User has defined a symbol name that conflicts with one of the special overlay manager names. Change the offending name and relink.

Please replace original diskette in drive A: and press ENTER.
: This message appears after the .EXE has been written if the /P switch is given. Insert the diskette with the list file so that it can be reopened.

Relocation table overflow

More than 16384 long calls or long jumps or other long pointers in the user's program. Rewrite program replacing long references with short references where possible and recreate object module. NOTE: Pascal and FORTRAN users should first try turning debugging off.

Segment limit set too high

The limit on the number of segments allowed was set too high using the /SEGMENTS switch. LINK will abort.

Segment limit too high

There is insufficient memory for the linker to allocate tables to describe the number of segments requested (either the value specified with /SEGMENTS or the default: 128). Either try the link again using /SEGMENTS to select a smaller number of segments (e.g. 64, if the default were used previously) or free some memory.

Segment size exceeds 64K

User has a small model program with more than 64Kbytes of code, or user has a middle model program with more than 64Kbytes of data. Try compiling and linking middle or large model.

Stack size exceeds 65536 bytes

The size specified for the stack using the /STACK switch is more than 65536 bytes.

Symbol table overflow

The user's program has greater than 256K of symbolic information (Publics, extrns, segments, groups, classes, files, etc). Combine modules and/or segments and recreate the object files. Eliminate as many public symbols as possible.

Terminated by user

The user entered Ctrl-C.

Too many external symbols in one module
: User's object module specified more than the allowed number of external symbols. Break up the module.

Too many group-, segment-, and class-names in one module
: User's program contains too many group, segment, and class names. Reduce the number of groups, segments, or classes and recreate the object files.

Too many groups
: User's program defines more than nine groups. Reduce the number of groups.

Too many GRPDEFs in one module
: LINK encountered more than 9 GRPDEFs in a single module. Reduce the number of GRPDEFs or split up the module.

Too many libraries
: User tried to link with more than 16 libraries. Combine libraries or link modules that require fewer libraries.

Too many overlays
: User's program defines more than sixty-three overlays. Reduce the number of overlays.

Too many segments
: The user's program has too many segments. Relink using the /SEGMENTS switch with an appropriate number of segments specified.

Too many segments in one module
: The user's object module has more than 255 segments. Split the modules or combine segments.

Too many TYPDEFs
: An object module contains to many TYPDEF records. These records are emitted by a compiler to describe communal variables. NOTE: At the present time, communal variables are not available in MASM.

Unexpected end-of-file on library
: The diskette containing the library has probably been removed. Try again after replacing the diskette with the library.

Unexpected end-of-file on scratch file
: Diskette containg VM.TMP was removed. Restart linker.

Unmatched left parenthesis
: User has made a typing mistake while specifying the contents of an overlay on the command line.

Unmatched right parenthesis
: User has made a typing mistake while specifying the contents of an overlay on the command line.

Unrecognized switch error: *'filename'*
: User entered an unrecognized character after the switch indicator /, such as:

```
A>LINK /ABCDEF main;
```

LINK will abort.

VM.TMP is an illegal file name and has been ignored
: User has used VM.TMP as an object file name. Rename file and link again.

Warning: no stack segment
: User's program contains no segment of combine-type stack.

Warning: too many local symbols
: The user has asked for a sorted listing of local symbols in the list file, but there are too many symbols to sort. The linker will produce an unsorted listing of the local symbols.

Warning: too many public symbols
: The user has asked for a sorted listing of public symbols in the list file, but there are too many symbols to sort. The linker will produce an unsorted listing of the public symbols.

Index

Index