Microsoft® Macro Assembler

for the MS-DOS® Operating System

User's Guide

Microsoft Corporation

If you have comments about the software, complete the Software Problem Report at the back of this manual and return it to Microsoft Corporation.

If you have comments about the software documentation, complete the Documentation Feedback reply card at the back of this manual and return it to Microsoft Corporation.

Document Number 410610001-400-R01-1285
Part Number 016-014-023

Contents

Tables

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:

Filename	**Description**
MASM.EXE	Microsoft Macro Assembler
LINK.EXE	Microsoft 8086 Object Linker
SYMDEB.EXE	Microsoft Symbolic Debug Utility
MAPSYM.EXE	Microsoft Symbol File Generator
CREF.EXE	Microsoft Cross-Reference Utility
LIB.EXE	Microsoft Library Manager
MAKE.EXE	Microsoft Program Maintenance Utility
EXEPACK.EXE	Microsoft EXE File Compression Utility
EXEMOD.EXE	Microsoft EXE File Header Utility
COUNT.ASM	Sample source file for **SYMDEB** session
README.DOC	Updated information obtained after the manual was printed

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.

Sections 1.2–1.8 explain what you need to create assembly-language programs, what steps you need to take to create these programs, and documentation conventions followed in this guide.

1.2 What You Need

The Microsoft Macro Assembler creates programs that can be executed under the 8086/80186/80286 family of microprocessors. It provides a logical program syntax ideally suited for the segmented architecture of these processors. Using **MASM** you can assemble programs for computers having the 8086, 8088, 80186, and 80286 microprocessors, and programs for computers with 8087 and 80287 math coprocessors.

In addition to a computer with one of the microprocessors listed above, you must have Version 2.0 or later of the MS-DOS® or PC-DOS operating system. Since these two operating systems are essentially the same, this manual uses the term MS-DOS to include both variations. Your computer system should also have at least 128K of memory. (The Shell command (!) of **SYMDEB** may require more memory.) While it is possible to operate the Macro Assembler with one double-sided disk drive, two disk drives or one disk drive and a hard disk are recommended.

To create assembly-language source files, you need a text editor capable of producing ASCII (American Standard Code for Information Interchange) format files with no control codes. Many text editors that normally use control codes or other special formats for documents also provide a programming or non-document mode for producing ASCII files.

1.3 What You Should Know Before You Begin

In order to use the Macro Assembler, you should be familiar with the following:

- How to use both the assembler itself, and the other programs provided with the Microsoft Macro Assembler package. This information is covered in the *Microsoft Macro Assembler User's Guide* (sometimes abbreviated *User's Guide*).
- How to program in assembly language. This information is covered partially in the *Microsoft Macro Assembler Reference Manual* (sometimes abbreviated *Reference Manual*). The directives, operands, operators, expressions, and other language features understood by **MASM** are explained in the reference manual. However, the reference manual is not designed to teach novice users how to program in assembly language.
- How to use the instruction sets for the 8086/80186/80286 microprocessors (and the 8087/80287 instruction set if you have a math coprocessor). This information is not covered in either the user's guide or the reference manual. The instruction set for the 8086 family of microprocessors is listed in Appendix A of the *Microsoft Macro Assembler Reference Manual.* Also, the Intel® Corporation pocket reference manual for the instruction sets is included with the Macro Assembler package. However, you need to have some knowledge of the instruction sets in order to use these reference tools.

In addition, you may need to know about MS-DOS structure and function calls, and about the basic input and output systems (BIOS) of the computers that will run your programs. This information is not covered in either the *Microsoft Macro Assembler User's Guide* or the *Microsoft Macro Assembler Reference Manual.*

If you are updating from a previous version of the Microsoft or IBM Macro Assembler, or if you will be using the assembler with a Microsoft or IBM high-level language, make sure you read Sections 1.6 and 1.7 for a summary of new features and potential compatibility problems.

Note

Many IBM languages are produced for IBM by Microsoft. Among the IBM languages that are the same or very similar to the corresponding Microsoft languages are IBM Personal Computer Macro Assembler, IBM Personal Computer FORTRAN, IBM Personal Computer Pascal, and IBM Personal Computer BASIC Compiler. These languages are compatible with the Microsoft Macro Assembler Version 4.0 except as noted in Section 1.7.

1.4 Books on Assembly Language

The following books may be useful in helping you learn how to program in assembly language:

Lafore, Robert, *Assembly Language Primer for the IBM PC & XT.* New York: Plume/Waite, 1984.

An introduction to assembly language including some information on DOS function calls and IBM-type BIOS.

Willen, David, and Jeffrey Krantz, *8088 Assembler Language Programming: The IBM PC.* Indianapolis: Howard W. Sams & Co. Inc, 1983.

An introduction to assembly language including some information on DOS function calls and IBM-type BIOS.

Bradley, David J., *Programming for the IBM Personal Computer.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1983.

Intermediate discussion of assembly language including information on macros, the 8087, MS-DOS (prior to Version 2.0), and IBM BIOS.

Sargent, Murray, III, and Richard L. Shoemaker, *The IBM Personal Computer from the Inside Out.* Menlo Park: Addison-Wesley Publishing Company, 1984.

An introduction to assembly language with an emphasis on using IBM-type hardware features.

Scanlon, Leo J., *IBM PC Assembly Language: A Guide for Programmers.* Bovie, MD: Robert J. Brady Co., 1983.

An introduction to assembly language including information on MS-DOS function calls.

Schneider, Al, *Fundamentals of IBM PC Assembly Language.* Blue Ridge Summit, PA: Tab Books Inc., 1984.

An introduction to assembly language including information on MS-DOS function calls.

Rector, Russel and George Alexy, *The 8086 Book.* Berkeley: Osborne/McGraw Hill, 1980.

Reference book on 8086 instruction set and architecture.

Norton, Peter, *The Peter Norton Programmer's Guide to the IBM PC.* Bellevue, WA: Microsoft Press, 1985.

Information on using IBM-type BIOS and MS-DOS function calls.

Morgan, Christopher and the Waite Group, *Bluebook of Assembly Routines for the IBM PC.* New York: New American Library, 1984.

Sample assembly routines that can be integrated into assembly or high-level-language programs.

iAPX 286 Programmer's Reference Manual. Santa Clara, CA: Intel Corporation, 1984.

Reference manual for all 8086-family instruction sets.

Microsoft MS-DOS Programmer's Reference Manual. Bellevue, WA: Microsoft Corporation.

Reference manual for MS-DOS.

These books are listed for your convenience only. Microsoft Corporation does not endorse these books (with the exception of those published by Microsoft Press) or recommend them over others on the same subjects.

1.5 How To Begin

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

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 simplify debugging 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 other MS-DOS programs, can accept command parameters, can be copied to other systems, and can be invoked from batch files or **MAKE** description files.

1.6 New Features

New features have been added to several of the programs in the Macro Assembler Package.

Version 4.0 of the Microsoft Macro Assembler (**MASM**) has been optimized to improve performance. It now assembles code two to three times faster than any prior release. In addition, the input/output buffers and macro text have been moved out of the symbol space, allowing assembly of larger source files.

Conditional error directives are another new feature of **MASM** 4.0. These directives allow you to check parameters, boundaries, and other assembly-time values, and generate an error if predefined conditions are not true. Conditional error directives are explained in Section 7.3 of the *Microsoft Macro Assembler Reference Manual.*

The following new options have been added to **MASM**:

Option	**Action**
/**B***number*	Sets the file buffer to any size between 1K and 63K in order to minimize disk access.
/**C**	Creates a cross-reference file.
/**L**	Creates an assembly listing.
/**D***symbol*	Defines a symbol (for conditional directives) from the command line or from prompts when starting **MASM**.
/**I***path*	Sets path by which assembler will search for files specified with an **INCLUDE** directive.
/**N**	Suppresses symbol table in listing.
/**P**	Checks for impure code that would cause problems in 80286 protected mode.
/**T**	Suppresses all messages if no errors are encountered.
/**V**	Displays extra statistics to the screen after assembly.
/**Z**	Displays source lines containing errors on the screen (without the option, only the error message is shown). Previous versions of **MASM** always showed both source line and error message.

The /**O** (Octal) option is no longer supported. **MASM** options are discussed in more detail in Section 2.3.

The format of the listing files produced by **MASM** has changed in several ways. See the example and description in Section 2.4. Several new exit codes have been added. See the list of exit codes in Appendix B.

LINK has two new options: the /**EXEPACK** option allows you to pack executable files during linking, while the /**HELP** option allows you to see a list of **LINK** options (see Sections 3.3.1 and 3.3.3). In addition, **LINK** has been optimized to make linking faster.

Several options have been added to **SYMDEB** since the version released with the Microsoft Macro Assembler, Version 3.0. The new options are listed below:

Option	**Action**
/K	Enables SCROLL LOCK or BREAK key as an interactive break-point key.
/N	Enables non-maskable interrupt break systems for non-IBM computers.
/S	Enables screen swapping between a **SYMDEB** screen and a program screen.
/"*commands*"	Executes the specified *commands* on start-up.

SYMDEB options are discussed in detail in Section 4.4.

CREF now uses all available memory space, allowing the program to process larger cross-reference files.

Two new capabilities and several options have been added to the **MAKE** utility. **MAKE** now supports macro definitions and inference rules. These features and the new **MAKE** options are described in Chapter 7.

The Macro Assembler Package now includes the **EXEPACK** utility, which allows you to pack executable files, and the **EXEMOD** utility, which allows you to modify the MS-DOS file header of .EXE files. These utilities are described in Appendix C.

1.7 Compatibility with Assemblers and Compilers

If you are upgrading from a previous version of the Microsoft or IBM Macro Assembler, you may need to make some adjustments before assembling source code developed with previous versions. The potential compatibility problems are listed below:

- Some previous versions of the IBM Macro Assembler wrote segments to object files in alphabetical order. The current version writes segments to object files in the order encountered in the source file. You can use the /A option to order segments alphabetically if this segment order is crucial in your previous source code. See Section 2.3.1 in this *User's Guide.*

- Some early versions of the Macro Assembler did not have strict type checking. Source code developed with these assemblers may produce error messages when assembled with newer versions. In some cases, listings in magazines and books are developed with the older assemblers. The source code can easily be made compatible using the **PTR** operator. Section 5.6 in the *Microsoft Macro Assembler Reference Manual* describes strict type checking and how to modify source code developed without this feature.

The Microsoft Macro Assembler is compatible with Microsoft (and most IBM) high-level languages. An exception occurs when **LINK** is used with IBM COBOL 1.0, IBM FORTRAN 2.0, or IBM Pascal 2.0. If source code developed with these compilers has overlays, you must use the linker provided with the compiler. Do not use the Microsoft linker.

When using **SYMDEB**, symbols may not be interpreted correctly in programs developed with old versions of FORTRAN and Pascal (Microsoft versions prior to 3.3 or IBM versions prior to 2.0). You can use the Symbol Set command (**Z**) to correct the symbol addresses (see Section 4.6.28).

1.8 Notational Conventions

This manual uses the following notational conventions in defining assembly-language syntax, and in presenting examples:

Convention	Meaning
Bold type	Bold type indicates commands, parameter names, or symbols that must be typed as shown. In most cases, upper- and lowercase letters can be freely intermixed. One exception is text within double quotation marks ("*text*"). Text in quotation marks is usually case-sensitive.

Examples

[*displacement*] [**DI**]
[**DI**+*displacement*]
[**DI**].*displacement*
[**DI**]+*displacement*

Note that in the examples above, the brackets must be typed as shown. The register name **DI** must also be typed as shown, though you could use lowercase letters. The plus sign (+) in both the second and fourth examples, and the period (.) in the third example must be typed as shown.

Italics

Italics indicate a placeholder: a name that you must replace with the value or file name required by the program.

Example

/**I***path*

In the example above, the slash (/) and the letter **I** must be entered as shown (except that the **I** could be lowercase). However, *path* is a placeholder representing a path name supplied by the user. You could enter any path name such as `B:\` or `\MASM\PROJECT1`. When a placeholder is used in a syntax example at the start of a section, the text below usually describes the types of values that can replace the placeholder.

⟦ ⟧

Double brackets indicate that the enclosed item is optional. Don't confuse double brackets with single brackets ([]), which must be typed as shown.

Example

BP ⟦*number*⟧ *address* ⟦*passcount*⟧ ⟦"*commands*"⟧

In the example above, you must enter **BP** as shown. You must also enter a value for the *address* placeholder. Values for the placeholders *number*, *passcount*, and *commands* can be entered if you wish, or they can be left blank. If you enter a value for *commands*, you must enclose the value in quotation marks ("").

,,,

A series of commas indicates that you can repeat the preceding item type if you separate each of the items with commas.

Example

⟦*name*⟧ *recordname* <⟦*initialvalue,,,*⟧>

In the example above, you may provide a *name* and you must provide a *recordname*. You may provide more than one *initialvalue* as long as you separate the values with commas. Note that you must type the angle brackets even if you do not provide any *initialvalue*.

¦

A vertical bar between items indicates that only one of the separated items can be used. You must make a choice between the items.

Example

D ⟦*address* ¦ *range*⟧

In the example above, you must enter the letter **D**. You may enter either an *address* or a *range* (but not both).

`Special typeface for examples`

Example text in this manual is shown in a special typeface so that it will look more like listings on the screen or produced with a printer.

Examples that represent source code follow these conventions:

- Lowercase letters for symbols, labels, instructions, and registers
- Uppercase letters for reserved words
- Uppercase letters for hexadecimal digits
- Lowercase letters for radix indicators
- Upper- and lowercase letters for comments

These are documentation conventions, not language requirements. Your source code can generally use any combination of upper- and lowercase letters, though your code will be clearer if you choose a convention and use it consistently.

Examples

```
count   DB      0
        mov     ax,bx
        ASSUME  cs:_text,ds:DGROUP
print   PROC    near
```

Chapter 2
MASM: A Macro Assembler

2.1 Introduction

The Microsoft Macro Assembler (MASM) assembles 8086, 80186, and 80286 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

Sections 2.2.1 and 2.2.2 explain how to start and use MASM to assemble your program source files. You can assemble source files with MASM using two different methods: by responding to a series of prompts, or with an MS-DOS command line.

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

2.2.1 Assembly Using 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 the following prompt:

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

2. Type the name of the file you wish to assemble and press the RETURN key. Include a drive and path name if the file is not in the current directory. If you do not give an extension, the assembler supplies the extension **.ASM**. The assembler requires a source file, so you cannot press just the RETURN key at this prompt as you can at other prompts.

Once you have pressed the RETURN key, **MASM** displays this prompt:

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

3. Note that *source* is the name of the file specified at the "Source filename" prompt. Type the name of the file to receive the relocatable object code and press the RETURN key. If you do not give a file-name extension, the assembler uses **.OBJ** by default. If you want to use the default file name (represented by *source*), do not type a file name. Just press the RETURN key.

   Once you have pressed the RETURN key, **MASM** displays this prompt:

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

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

   Once you have pressed the RETURN key, **MASM** displays this prompt:

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

5. If you want the assembler to create a cross-reference file, type the name for the file and press the RETURN key. If you do not supply a file-name extension, the assembler uses **.CRF** by default. If you do not want a cross-reference listing, do not type a file name. Just press the RETURN key.

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

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

You must use an appropriate path name for any file that is not in the current drive and directory.

At any prompt, you can type the rest of the file names in the command line format. For example, you can choose the default responses for all remaining prompts by typing a semicolon (;) after any prompt (as long as you have supplied a source-file name), or you can type commas (,) to indicate several files, as described in Section 2.2.2. When **MASM** encounters a semicolon, it immediately chooses the default responses and processes the remaining files without displaying any more prompts.

Examples

MASM

```
Source filename [.ASM]: file
Object filename [file.OBJ]: b:file
Source listing  [NUL.LST]: PRN /D
Cross-reference [NUL.CRF]: b:\cref\file
```

This example directs **MASM** to assemble the source file `file.asm` on the current drive and place the relocatable object code in `file.obj` on the current directory of Drive B. The device name and the **/D** option at the "Source listing" prompt direct **MASM** to send a listing (including a Pass 1 listing) to the line printer (the **/D** option is described in Section 2.3.1). **MASM** also sends cross-reference data to `file.crf` in the `\cref` directory of Drive B.

MASM

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

The example above 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-file name directs the assembler to select the default file names for the remaining prompts. This means the assembler creates no assembly listing or cross-reference listing.

2.2.2 Assembly Using a Command Line

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 following form:

MASM *sourcefile* ⟦,⟦*objectfile*⟧ ⟦,⟦*listingfile*⟧ ⟦,⟦*crossreferencefile*⟧⟧⟧⟧ ⟦*options*⟧ ⟦;⟧

The *sourcefile* must be the name of the source file to be assembled. If you do not supply a file-name extension, **MASM** supplies the extension **.ASM**.

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

The optional *objectfile* is the name of the file to receive the relocatable object code. If you do not supply a name, **MASM** uses the source-file name, replacing the extension with **.OBJ**.

The optional *listingfile* is the name of the file to receive the assembly listing. The assembly listing shows the assembled code for each source statement and the names and types of symbols defined in the program. If you do not supply a file-name extension, **MASM** supplies the extension **.LST**.

The optional *crossreferencefile* is the name of the file to receive the cross-reference output. The resulting cross-reference file can be processed with **CREF**, the Microsoft 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 file-name extension, **MASM** supplies **.CRF** by default.

You can use a semicolon (;) in the command line to select defaults for the remaining file names. A semicolon after the source-file name selects a default object-file name and suppresses creation of the assembly listing and cross-reference files. A semicolon after the object-file name suppresses just the listing and cross-reference files. A semicolon after the listing-file name suppresses only the cross-reference file.

All files created during the assembly will be written to the current drive and directory unless you specify a different drive for each file. You must separately specify the alternate drive and path for each file that you do not want to go on the current directory.

You can also specify a device name instead of a file name. For example, **NUL** for no file or **PRN** for the printer.

Note

Unless a semicolon (;) is used, all the commas in the command line are required. If you want the file name for a given file to be the default (the file name of the source file), place the commas that would otherwise separate the file name from the other names side by side (,,).

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

Examples

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

The example above is equivalent to:

```
MASM file,,,;
```

The source file `file.asm` is assembled. 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;
```

The example above directs **MASM** to assemble the source file `startup.asm`. The assembler then writes the relocatable object code to the default object file, `startup.obj`. **MASM** creates a listing file named `stest.lst`, but the semicolon keeps the assembler from creating a cross-reference file.

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

The example above is exactly the same as the previous example except that the assembler creates a cross-reference file `startup.crf`. This is because the semicolon follows a comma marking the place of the cross-reference file instead of following the file name of the list file.

```
MASM  B:\src\build;
```

The example above directs **MASM** to find and assemble the source file `build.asm` in the directory `\src` on Drive B. The semicolon causes the assembler to create an object file named `build.obj` in the current directory, but prevents **MASM** from creating an assembly listing or cross-reference file. Note that the object file is placed on the current drive, not the drive specified for the source file.

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:

Option	**Action**
/**A**	Writes segments in alphabetical order
/**S**	Writes segments in source-code order
/**B***number*	Sets buffer size
/**C**	Specifies a cross-reference file
/**L**	Specifies an assembly listing file
/**D**	Creates Pass 1 listing
/**D***symbol*	Defines assembler symbol
/**I***path*	Sets include file search path
/**ML**	Preserves case sensitivity in names
/**MX**	Preserves case sensitivity in public and external names
/**MU**	Converts names to uppercase
/**N**	Suppresses tables in listing file
/**P**	Checks for impure code
/**R**	Creates code for real floating-point instructions
/**E**	Creates code for emulated floating-point instructions
/**T**	Suppresses messages for successful assembly
/**V**	Displays extra statistics to screen
/**X**	Includes false conditionals in listings
/**Z**	Displays error lines on screen

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. Options can be specified with either a forward slash (/) or a dash (-), and with either upper- or lowercase letters. The options /**A**, /**a**, **-A**, and **-a** are equivalent.

Note

> You should not use source-file names containing dashes. Although the dash is a legal character for MS-DOS file names, the assembler will interpret a dash as the beginning of an assembler option. For example, the file name `file-c` will be interpreted by the assembler as `file` followed by the invalid option `-c`. An error message will result.

2.3.1 Writing Segments in Alphabetical Order

Syntax

/A

The **/A** option directs **MASM** to place the assembled segments in alphabetical order before copying them to the object file. If this option is omitted, **MASM** copies the segments in the order encountered in the source file.

Note

> Some previous versions of the macro assembler ordered segments alphabetically by default. Listings in books and magazines may be written with these early versions in mind. If you have trouble assembling and linking a listing taken from a book or magazine, try using the **/A** option.

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 segments with the class types `'DATA'`, `'CODE'`, and `'STACK'`, the assembled segments in the object file have the order `'CODE'`, `'DATA'`, and `'STACK'`. The significance of segment order and class type are discussed in more detail in Sections 3.4.2 and 3.4.3 in this manual, and in Section 3.4.3 of the *Microsoft Macro Assembler Reference Manual.*

2.3.2 Writing Segments in Source-Code Order

Syntax

/S

The **/S** option tells **MASM** to place the assembled segments in the object file in the same order in which they appear in the source file. This is the default order. The **/S** option is provided for compatibility with XENIX®.

2.3.3 Setting the File Buffer Size

Syntax

/**B***number*

The **/B** option directs the assembler to change the size of the file buffer used for the source file. The *number* is the number of 1024-byte (1K) memory blocks allocated for the buffer. You can set the buffer to any size from 1K to 63K (but not 64K). The default size of the buffer is 32K.

A buffer larger than your source file allows you to do the entire assembly in memory, greatly increasing assembly speed. However, you may not be able to use a large buffer if your computer does not have enough memory or if you have too many resident programs using up memory. If you get an error message indicating insufficient memory, you can decrease the buffer size and try again.

Examples

```
MASM file,,/B16;
```

The example above decreases the buffer size to 16K.

```
MASM file,,/B63;
```

The example above increases the buffer size to 63K.

2.3.4 Creating a Pass 1 Listing

Syntax

/D

The **/D** option tells **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 program phase errors. Phase errors occur when the assembler 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 the assembler to display error messages for both Pass 1 and Pass 2 of the assembly, even if no assembly listing is created. See Section 2.4.6 for more information about Pass 1 listings.

Example

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

This example directs the assembler to create a Pass 1 listing for the source file `file.asm`. The listing is placed in the file `file.lst`.

2.3.5 Defining Assembler Symbols

Syntax

/D*symbol*

The **/D***symbol* option directs **MASM** to define a symbol that can be used during the assembly as if it were defined in the source file. The specified symbol is defined as a null-text string. This is similar to using the **EQU** directive within the source file to define a string.

The **/D***symbol* option can be used to define symbols that can be evaluated by the **IFDEF** and **IFNDEF** conditional-assembly directives. These directives are explained in Section 7.2.3 of the *Microsoft Macro Assembler Reference Manual.*

Example

```
MASM file,,/Dwide;
```

This example defines the symbol `wide` and gives it a null value. The symbol could then be used in the following conditional-assembly block:

```
IFDEF wide
     PAGE 50,132
ENDIF
```

When the symbol is defined in the command line, the listing file is formatted for a 132-column printer. When the symbol is not defined in the command line, the listing file is given the default width of 80 (see the description of the **PAGE** directive in Section 9.8 of the *Microsoft Macro Assembler Reference Manual*).

2.3.6 Setting a Search Path for Include Files

Syntax

/**I***path*

The /**I** option is used to set search paths for include files. You can set up to 10 search paths by using the option for each path. The order of searching is the order in which the paths are listed in the command line. The **INCLUDE** directive and include files are discussed in Section 9.2 of the *Microsoft Macro Assembler Reference Manual.*

Example

```
MASM file,, /Ib:\io /I\macro ;
```

This command line might be used if the source file contains the following statement:

```
INCLUDE dos.mac
```

In this case, **MASM** would search for file `dos.mac` first in directory `\io` on Drive B, then in directory `\macro` on the current drive, and finally in the current directory.

You should not specify a path name with the **INCLUDE** directive if you plan to specify search paths from the command line. For example, if the source file contained the statement

```
INCLUDE a:\macro\dos.mac
```

MASM would search path `a:\macro` and would ignore any search paths specified in the command line.

2.3.7 Preserving Case-Sensitivity in Names

Syntax

/ML

The **/ML** option directs the assembler to preserve lowercase letters in label, variable, and symbol names. All names that have the same spelling, but use letters of different cases are considered different. For example, with the **/ML** option, `DATA` and `data` are different. Without the option, the assembler automatically converts all lowercase letters in a name to uppercase.

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

Example

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

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

2.3.8 Preserving Case-Sensitivity in Public and External Names

Syntax

/MX

The /MX option directs the assembler to preserve lowercase letters in public and external names. **MASM** converts all other names to uppercase.

Public and external names include any label, variable, or symbol names defined using the **EXTRN** directive or the **PUBLIC** directive. See Chapter 6 of the *Microsoft Macro Assembler Reference Manual* for more information on global directives. If the /MX option is specified, the assembler writes public and external names to the object file in exactly the form in which they appear in the source file. The names `DATA` and `Data` would be different if written to the object file with the /MX option.

The /MX option is used to ensure that the names of routines or variables copied to the object module have unique spelling regardless of whether they are spelled with upper- or lowercase letters. The option is used with any source file to be linked with object modules created by a case-sensitive compiler.

Example

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

The preceding example directs **MASM** to preserve lowercase letters in any public or external names defined in the source file `file.asm`.

2.3.9 Converting Names to Uppercase

Syntax

/MU

The /MU option causes the assembler to convert lowercase letters to uppercase in public and external names. This is the default. The /MU option is provided for compatibility with XENIX.

2.3.10 Suppressing the Tables in the Listing File

Syntax

/N

The /N option tells the assembler to omit all tables from the end of the listing file. If this option is not chosen, **MASM** will include tables of macros, structures, records, segments and groups, and symbols. The code portion of the listing file is not changed by the /N option.

Example

```
MASM file,,/N;
```

2.3.11 Checking for Impure Code

Syntax

/P

The /**P** option directs **MASM** to check for impure code in the 80286 protected mode. This option has no effect unless assembly is being controlled by the **.286p** directive. The **.286p** and other instruction-set directives are explained in Section 3.3 of the *Microsoft Macro Assembler Reference Manual.*

Code that moves data into memory with the **CS:** override instruction is acceptable in nonprotected 286 mode and in 8086 and 80186 mode. However, such code may cause problems in protected mode. When the /**P** mode is in effect, the assembler checks for these situations and generates error 100 if it encounters them.

Example

```
MASM file /P;
```

This example instructs **MASM** to check for impure code where instruction data are moved directly into memory through a **CS:** override instruction.

2.3.12 Creating Code for a Floating-Point Processor

Syntax

/R

The **/R** option directs the assembler to generate floating-point instruction code that can be executed by an 8087 or 80287 coprocessor. Programs created using the **/R** option can run only on machines having an 8087 or 80287 coprocessor.

Example

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

This example directs **MASM** to assemble the source file `file.asm` and create actual 8087 or 80287 instruction code for floating-point instructions.

2.3.13 Creating Code for a Floating-Point Emulator

Syntax

/E

The **/E** option directs the assembler to generate floating-point instruction code that emulates the 8087 or 80287 coprocessor. This option is for the convenience of programmers who already own a math-emulation library such as the ones provided with Microsoft C, Pascal, and FORTRAN. The Microsoft Macro Assembler package does not include a math-emulation library.

If you intend to execute a program that uses 8087 or 80287 instructions on machines that do not have an 8087 or 80287 coprocessor, you must use the **/E** option during assembly, and then link the resulting object file with a math-emulation library. The library contains routines that emulate 8087 and 80287 floating-point instructions.

Example

```
MASM file /E;
LINK file,,,math.lib
```

This example directs **MASM** to create emulation code for any floating-point instructions it finds in the program. Note that the object file is linked with a math-library file in the second command line. If you try to use the **/E** option without a math library, you will be able to assemble the file successfully, but you will get error messages when you try to link the object file.

2.3.14 Displaying Extra Assembly Statistics

Syntax

/V

The /V option directs the assembler to send additional statistics to the screen at the end of assembly. In addition to the normal data on errors and symbol space, **MASM** reports the number of lines and symbols processed. (The V in the option name is mnemonic for verbose.)

Example

```
MASM file/V;
```

2.3.15 Listing False Conditionals

Syntax

/X

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**. Conditional-assembly directives are explained in Section 7.2 of the *Microsoft Macro Assembler Reference Manual.*

The **.SFCOND**, **.LFCOND**, and **.TFCOND** directives modify the effect of the /X option. A **.SFCOND** in the source file suppresses false conditionals while a **.LFCOND** directive restores listing of false conditionals. Both these directives work regardless of whether the /X option is given on the command line. A **.TFCOND** directive in the source file reverses the normal meaning of the /X option. When the /X option has been given and the assembler encounters a **.TFCOND** directive in a source file, subsequent false conditionals are suppressed. The next **.TFCOND** directive restores the listing.

The following table illustrates the effect of the **.TFCOND**, **.SFCOND**, and **.LFCOND** directives on the /X option:

Table 2.1

/X Option and Directives

Source File Directive:	/X Option Action:
.SFCOND	Has no effect; false conditionals not listed
.LFCOND	Has no effect; false conditionals listed
.TFCOND	Toggles between listing & suppressing false conditionals
No directive	Lists false conditionals

The /X option does not affect the assembly listing unless you direct the assembler to create an assembly-listing file. See Section 9.10 in the *Microsoft Macro Assembler Reference Manual* for more information about directives that control listing of false conditionals.

Example

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

If the source file, `file.asm` contains two **.TFCOND** directives, the assembler will start listing false conditionals at the first directive and continue until it reaches the second. It will continue to toggle between listing and suppressing each time it encounters a new **.TFCOND** directive.

2.3.16 Displaying Error Lines on the Screen

Syntax

/Z

The **/Z** option directs **MASM** to display lines containing errors on the screen. Normally when the assembler encounters an error, it displays only an error message describing the problem. When you use the **/Z** option in the command line, the assembler displays the source line that produced the error in addition to the error message. **MASM** assembles faster without the **/Z** option, but you may find the convenience of seeing incorrect source lines worth the slight cost in processing speed.

Previous versions of **MASM** always showed both the source line and the error message.

Example

```
MASM file/Z;
```

2.3.17 Specifying a Cross-Reference File

Syntax

/C

The **/C** option directs **MASM** to create a cross-reference file even if one was not specified in the command line or in response to prompts. A cross-reference file specified with the **/C** option always has the base name of the source file plus the extension **.CRF**. You cannot specify a file name with this option. The **/C** option is provided for compatibility with XENIX.

2.3.18 Specifying a Listing File

Syntax

/L

The /L option directs MASM to create an assembly-listing file even if one was not specified in the command line or in response to prompts. An assembly-listing file specified with the /L option always has the base name of the source file plus the extension .LST. You cannot specify a file name with this option. The /L option is provided for compatibility with XENIX.

2.3.19 Suppressing Messages for Successful Assembly

Syntax

/T

The /T option suppresses all messages if the source file is assembled without any warning errors or severe errors. The copyright message and information about errors and symbol space appear only if at least one error is encountered. This option may be useful in batch files if the user does not want the output cluttered with unnecessary messages. (The T in the option name is mnemonic for terse.)

2.4 Reading the Assembly Listing

MASM creates an assembly listing of your source file whenever you give an assembly-listing file name on the MASM command line or in response to the MASM prompts. The assembly listing contains both the statements in the source-program file, and the object code generated for each statement. The listing also shows the names and values of all labels, variables, and symbols in your source file.

The assembler creates tables for macros, structures, records, segments, groups, and other symbols. These tables are placed at the end of the assembly listing (unless you suppress them with the /N option). MASM lists only the types of symbols encountered in the program. If your program has no macros, there will be no macro section in the symbol table.

The assembly listing also contains error messages if errors occurred during assembly. **MASM** places each message below the statement that caused the error. At the end of the listing, the assembler tells how many error and warning messages it issued.

Sections 2.4.1–2.4.6 explain the format of assembly listings and the meanings of special symbols used in listings.

2.4.1 Reading Code in the Listing

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

⟦*linenumber*⟧ *offset code statement*

The optional *linenumber* is the number of the line starting from the first statement in the assembly listing. Line numbers are produced only if you request a cross-reference file. Line numbers in the listing do not always correspond to the same lines in the source file.

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 for the statement. **MASM** gives the actual numeric value of the code in hexadecimal if possible. Otherwise, it indicates what action is necessary to compute the value. The *statement* is the source statement shown exactly as it appears in the source file, or as expanded by a macro.

If any errors occur during assembly, each error message and error number will be printed directly below the statement where the error occurred. Refer to Appendix A for a list of **MASM** errors. Error messages show the source-file name, the source-line number, the error number, and an error message as shown below:

```
     28                                               nov      ds,ax
work.ASM(22) : error 10: Syntax error
```

Note that the 22 in the error message is the line number in the source file. The 28 on the code line is the line number of the listing file, which may not be the same as the source line. Line numbers in the listing file are produced only if you request a cross-reference file.

The assembler uses the special characters shown in Table 2.2 to indicate addresses that need to be resolved by the linker or values that were generated in a special way:

Table 2.2

Special Characters in Listings

Character	Meaning
R	Relocatable address; linker must resolve
E	External address; linker must resolve
----	Segment/group address; linker must resolve
=	**EQU** or equal-sign (=) directive
nn:	Segment override in statement
nn/	**REP** or **LOCK** prefix instruction
nn[xx]	**DUP** expression; nn copies of the value xx
n	Macro expansion nesting level (+ if more than nine)
C	Line from **INCLUDE** file

Example

```
Microsoft MACRO Assembler  Version 4.00      9/25/85 13:58:46

                                             Page     1-1

      1                              quit    MACRO
      2                                      mov      ah,4Ch
      3                                      int      21h
      4                                      ENDM
      5
      6 = FFFF                       max     EQU      65535
      7
      8                                      EXTRN    work:NEAR
      9
     10 0000                         stack   SEGMENT para public 'STACK'
     11 0000   0100[                         DB       256 DUP(?)
     12             ??
     13                         ]
     14
     15 0100                         stack   ENDS
     16
     17 0000                         data    SEGMENT public 'DATA'
     18 0000   0064[                 buffer  DW       100 DUP(?)
     19             ????
```

```
      20                          ]
      21
      22 00C8                         data    ENDS
      23
      24 0000                         code    SEGMENT public 'CODE'
      25                                      ASSUME  cs:code, ds:data
      26
      27 0000   B8 ---- R             start:  mov     ax,data
      28                                      nov     ds,ax
test.ASM(22) : error 10: Syntax error
      29 0003   E8 0000 E                     call    work
      30                                      quit
      31 0006   B4 4C               1         mov     ah,4Ch
      32 0008   CD 21               1         int     21h
      33 000A                         code    ENDS
      34                                      END     start

Microsoft MACRO Assembler  Version 4.00       9/25/85 13:58:46

                                              Symbols-1

Macros:

          N a m e             Lines

QUIT . . . . . . . . . . .        2

Segments and Groups:

          N a m e             Size   Align  Combine     Class

CODE . . . . . . . . . . . .  000A   PARA   PUBLIC      'CODE'
DATA . . . . . . . . . . . .  00C8   PARA   PUBLIC      'DATA'
STACK  . . . . . . . . . . .  0100   PARA   PUBLIC      'STACK'

Symbols:

          N a m e             Type   Value  Attr

BUFFER . . . . . . . . . . .  L WORD        0000   DATA Length = 0064

MAX  . . . . . . . . . . . .  Number FFFF

START  . . . . . . . . . . .  L NEAR 0000   CODE

WORK . . . . . . . . . . . .  L NEAR 0000          External

     26 Source  Lines
     28 Total   Lines
     29 Symbols

  50002 Bytes symbol space free

      0 Warning Errors
      1 Severe  Errors
```

The line numbers referencing the sample source file indicate that a cross-reference file was requested when the file was assembled. Source and reference files for this sample listing are shown in Section 5.3.

2.4.2 Reading a Macro Table

The table at the end of a listing file shows the names and sizes of all macros defined in the source file. The list has two columns with the headings `Name` and `Lines`, as shown in the following example:

```
                 N a m e            Lines

BIOSCALL . . . . . . . . . .            2
DISPLAY  . . . . . . . . . .            3
DOSCALL  . . . . . . . . . .            2
KEYBOARD . . . . . . . . . .            4
LOCATE . . . . . . . . . . .            7
SCROLL . . . . . . . . . . .            6
```

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 conversion is suppressed with the **/ML** option). Names longer than 31 characters are truncated. The `Lines` column lists the number of lines in the macro.

2.4.3 Reading a Structure and Record Table

The table at the end of a listing file shows the names and dimensions of all structures and records in the source file.

The `Name` column lists the name of the structure or record, and this is followed on succeeding indented lines by the names of the fields within the structure or record. 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 conversion is suppressed with the **/ML** option). Names longer than 31 characters are truncated.

The following example shows the format for structures:

N a m e	Width Shift	# fields Width	Mask	Initial
STRUC1	001A	0003		
COUNT	0000			
VALUE	0001			
NAME	0015			

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 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 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.

The following example shows the format for records:

N a m e	Width Shift	# fields Width	Mask	Initial
REC0	000B	0002		
FL1	0003	0008	07F8	0400
FL2	0000	0003	0007	0002
REC1	000A	0003		
FL1	0006	0004	03C0	0000
FL2	0003	0003	0038	0000
FL3	0000	0003	0007	0000

For fields in a record, the `Shift` column lists the offset (in bits) from the low-order bit of the record to the low-order 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.

2.4.4 Reading a Segment and Group Table

The following example of a table at the end of a listing file shows the names, sizes, and attributes of all segments and groups in the source file:

```
                N a m e          Size   Align  Combine    Class

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'
FIRST  . . . . . . . . . . .     0037   WORD   PUBLIC     'CODE'
MAIN_STARTUP . . . . . . . .     007E   PARA   NONE       'MEMORY'
```

The table 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 byte size (in hexadecimal) of each segment. Since a group has no size, only the word `GROUP` is shown.

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

byte

word

para

page

at

If the segment is defined with no explicit align type, **MASM** lists the default align type 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

common

address (for **at** combine type)

If no explicit combine type is defined for the segment, the listing shows `NONE`, representing the private combine type. If the `Align` column contains `AT`, the `Combine` column contains that hexadecimal address of the beginning of the segment.

The `Class` column lists the class name of the segment. The name is spelled exactly as given in the source file except that lowercase letters are converted to uppercase (unless the /ML option is used). If no name is given, none is shown.

For a complete explanation of the align and combine types, and class names, see Section 3.4 of the *Microsoft Macro Assembler Reference Manual.*

2.4.5 Reading a Symbol Table

The following example of a table at the end of a listing file shows the names, types, values, and attributes of all symbols in the source file:

```
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
```

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 conversion is suppressed with the /ML option for all names or with the /MX option for public and external names). Names longer than 31 characters are truncated.

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

Type	Definition
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 the symbol is defined by an **EQU** directive or an equal-sign (=) directive, the `Type` column will show either `Number`, `Opcode`, `Alias`, or `Text`. If the symbol represents a variable, label, or procedure, the `Type` column will show the symbol's length if it is known. A length is given as one of the following:

Type	Length
BYTE	One byte (8-bits)
WORD	One word (16-bits)
DWORD	Doubleword (2 words)
QWORD	quadword (4 words)
TBYTE	Ten-bytes (5 words)
number	Length in bytes of a structure variable

If the symbol represents an absolute value defined with an **EQU** or equal-sign (=) directive, the `Value` column shows the symbol's value. The value may be another symbol, a string, or a constant numeric value (in hexadecimal), depending on whether the type is `Alias`, `Text`, or `Number`. If the type is `Opcode`, the `Value` column will be blank. If the symbol represents a variable, label, or procedure, the `Value` column shows the symbol's hexadecimal offset from the beginning of the segment in which it is defined.

The Attr column shows the attributes of the symbol. The attributes include the name of the segment (if any) in which the symbol is defined, the scope of the symbol, and the code length. A symbol's scope is given only if the symbol is defined using the **EXTRN** and **PUBLIC** directives. The scope can be External or Global. The code length (in hexadecimal) is given only for procedures. The Attr column is blank if the symbol has no attribute.

2.4.6 Reading a Pass 1 Listing

When you specify the **/D** option in the **MASM** command line, the assembler puts a Pass 1 listing in the assembly-listing file, making the listing file show the results of both assembler passes. The listing is intended to help locate the sources 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, the jle instruction to a forward reference produces an error message:

```
 0017  7E 00               jle     smlstk
PASS_CMP.ASM(20) : error 9 : Symbol not defined SMLSTK
 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 the assembler 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, the assembler would have displayed an error message.

In the following program fragment, a mistyped label creates a phase error:

```
 0000                code segment
 0000 E9 0000  U          jmp   go
PASS_TST.ASM(2)  : error 9: Symbol not defined GO
 0003                go      label   byte
 0003 B8 0001                mov     ax, 1
 0006                code    ends
```

In Pass 1, the label `go` is used in a forward reference and creates a `Symbol not defined` error. The assembler assumes that the symbol will be defined later and generates 3 bytes of code, reserving 2 bytes for the symbol's actual value.

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 2 bytes of code in Pass 2, 1 byte less than in Pass 1. The result is a phase error:

```
 0000                code segment
  0003 R                  jmp   go
PASS_TST.ASM(2)  : error 57: Illegal size for item
 0003                go   label byte
PASS_TST.ASM(3)  : error 6: Phase error between passes
 0003 B8 0001             mov   ax, 1
 0006                code ends
```

Most Pass 1 errors are resolved in Pass 2, so they are not counted as either warning or severe errors in the error count. However, there are five Pass 1 errors that cannot be resolved during Pass 2. They are counted in the error count and listed on the first page of the listing file even if no Pass 1 listing is requested. The following five Pass 1 errors will be included in the listing:

Code	**Message**
2	`Register already defined`
5	`Redefinition of symbol`
13	`Must be declared in pass 1`
17	`Forward reference is illegal`
85	`End of file, no END directive`

Chapter 3
LINK: A Linker

3.1 Introduction

The Microsoft 8086 Object Linker, (**LINK**), creates executable programs from object files generated by the Microsoft Macro Assembler (**MASM**) or by high-level-language compilers, such as C or Pascal. The linker 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 the linker 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 one megabyte of code and data.

Section 3.2 explains how to use the linker to create executable programs. Section 3.3 defines each of the options you can use in a **LINK** command line to control the linking process. Section 3.4 explains how **LINK** creates programs.

3.2 Starting and Using LINK

This section explains how to start and use the linker to create executable programs. You can use **LINK** in three different ways: by answering a series of prompts, by supplying an MS-DOS command line, or by using a response file. The three methods can also be mixed.

Once you start **LINK**, it will either process the files you supplied or prompt you for additional files. You can stop the linker at any time by pressing the CONTROL-C key combination.

3.2.1 Using Prompts to Specify LINK Files

When you type the command name **LINK** at the MS-DOS prompt, the linker will prompt you for the information it needs. 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 you do not supply file-name 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. The linker displays the following prompt:

   Run File [*filename*.EXE]:

3. Note that *filename* is the same as the first file name entered at the "Object Modules" prompt. 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-file name, just press the RETURN key. The file name 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 file-name extension, the linker uses **.MAP** by default. If you do not want a map file, do not type a file name. Just press the RETURN key.

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

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

5. Type the names of any library files 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 file-name extensions, the linker 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 file names.

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.

When entering file names, you must give a path name for any file that is not on the current drive and directory. You can use LINK options by typing them after the file name at any prompt. If the linker cannot find an object file, it displays a message and waits so that you can change disks if necessary.

At any prompt, you can type the rest of the file names in the command line format described in Section 3.2.2. For example, you can choose the default responses for all remaining prompts by typing a semicolon (;) after any prompt, or you can type commas (,) to indicate several files. (If you type a semicolon at the "Object Modules" prompt, be sure to supply at least one object-file name.) When the linker encounters a semicolon, it immediately chooses the default responses and processes the remaining files without displaying any more prompts.

Example

```
LINK

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

This example links the object modules `moda.obj`, `modb.obj`, `modc.obj`, and `startup.obj`. It searches the library file `math.lib` on Drive B of 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 Modules" prompt line causes **LINK** to pause while you change disks. The linker then creates the executable file (see Section 3.3.2).

3.2.2 Using a Command Line to Specify LINK Files

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 following general form:

LINK *objectfiles* [[,[[*executablefile*]] [[,[[*mapfile*]] [[,[[*libraryfile*]]]]]]]] [[*options*]] [[;]]

The *objectfiles* include 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 linker requires at least one object file. If you do not supply an extension, **LINK** provides the extension **.OBJ**.

The optional *executablefile* is a placeholder for the name you wish to give the executable file **LINK** will create. If you do not supply an *executablefile*, **LINK** creates a file name by using the file name of the first object file in the command line and appending the extension **.EXE**.

The optional *mapfile* is the name of the file to receive the map listing. If you do not supply an extension, the linker provides the extension **.MAP**. If you specify the **/MAP** or **/LINENUMBERS** option, a map file will be created even if no map file was specified in the command line.

The optional *libraryfiles* include the name or names of the libraries containing routines that you wish to link to create a program. If you do not supply an extension, **LINK** supplies the extension **.LIB**.

The *options* control the operation of **LINK**. You can use any of the options listed in Section 3.3. You can put *options* anywhere on the command line.

The commas (,) separating file names for the different types of files are required even if no file name is supplied. If you want the file name for a file to be the default (the same as the base name of the first object file), you can type the comma that would follow the file name without actually supplying a file name. You can use a semicolon (;) anywhere after the object file to terminate the command line. If you type the comma after the object file, **LINK** will supply the default name for the *executablefile* and suppress the *mapfile* and the *libraryfiles*.

If you do not supply all file names in the command line and do not end with a semicolon, the linker will prompt for additional files, using the prompts described in Section 3.2.1. If you give more than one object file or library file, you must separate the names with spaces or with plus signs (+).

If you do not specify a drive or directory for a file, **LINK** assumes the file will be on the current drive and directory. You cannot specify the drive or directory for the *objectfile* and expect **LINK** to supply the same drive and directory for other files. The location of each file must be given specifically.

Note

When linking modules produced with a high-level-language compiler that supports overlays, you must specify overlay modules by putting them in parentheses. Since **MASM** has no overlay manager, you can only specify overlays for object files linked with the run-time library of a language compiler that supports overlays. For example, you can use overlays with modules compiled with Microsoft FORTRAN, Version 3.2 and later, Microsoft Pascal, Version 3.2 and later, and Microsoft C, Version 3.0 and later. See your language compiler manual for details on specifying overlays.

Examples

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

The first example is equivalent to the following line:

```
LINK file,,,routine
```

It 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 also creates a file called `file.map` containing a list of the program's segments and groups.

```
LINK startup+file,b:file,\map\file;
```

The second example uses the two object files `startup.obj` and `file.obj` on the current drive to create an executable file named `file.exe` on Drive B. **LINK** creates a map file on the `\map` directory of the current drive, but does not search any libraries.

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

The final example links the object modules `moda.obj`, `modb.obj`, `modc.obj`, and `startup.obj`. The linker searches through the library file `math.lib` in the `\lib` directory on Drive B 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 the linker to pause while you change disks before creating the executable file (see Section 3.3.2).

3.2.3 Using a Response File to Specify LINK Files

You can create a program by listing, in a response file, the names of all the files to be processed, and by giving the name of the response file on the LINK command line. The simplest way to use a response file is with a command line having the following form:

LINK @*filename*

A response file can also be specified at any prompt, or at any position in a command line. The input from the response file will be treated exactly as though it had been entered at prompts or in a command line, except that carriage-return/line-feed combinations in the file are treated the same as the RETURN key in response to a prompt, or a comma in a command line.

When specifying a response file, the *filename* must be the name of the response file, and it must be preceded by an at sign (@). If the file is in another directory or on another disk drive, a path name must be provided.

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

objectfiles
⟦*executablefile*⟧
⟦*mapfile*⟧
⟦*libraryfiles*⟧

Elements that have already been provided at prompts or with a partial command line can be omitted.

Each group of file names must be placed on a separate line. 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 file name 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 file names for all files you have not yet named in the response file. The remainder of the response file is ignored.

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

Note

A response file should end with either a semicolon (;) or a carriage-return/line-feed combination. If you fail to provide a final carriage-return/line-feed in the file, the linker will display the last line of the response file and wait for you to press the RETURN key.

Example

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

The response file above tells the linker 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`. The linker also creates a map file `abc.map`, and searches the library `math.lib` in the `\lib` directory of Drive B.

The following procedure combines all three methods of supplying file names. Assume you have a response file called `library` that contains one line:

```
lib1+lib2+lib3+lib4
```

Now start **LINK** with a partial command line:

```
LINK object1 object2
```

LINK takes `object1.obj` and `object2.obj` as its object files, and prompts for the next file:

```
Run File [object1.EXE]: exec
List File [NUL.MAP]:
Libraries [.LIB]: @library
```

You enter `exec` so that the linker will name the executable file `exec.exe`. You press the RETURN key to indicate that no map file is desired, and you enter `@library` so that the linker will read in the response file containing the four library-file names.

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**. Environment variables are explained under the **SET** command in the *Microsoft MS-DOS User's Guide.*

A search path is 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" prompt. You can specify up to 16 search paths. You can also assign the search paths to the **LIB** environment variable, using the MS-DOS **SET** command. In the latter case, the search paths must be separated by semicolons (;).

If a drive or directory name is included in the file name for a library in the **LINK** command line, the linker searches there only. If no drive or directory is given, **LINK** searches for library files in the following order:

1. First the linker searches the current drive and directory.
2. If the library is not found and one or more search paths have been given in the command line, the linker searches the specified search paths in the order in which they were given.
3. If the library is still not found and a search path has been set with the **LIB** environment variable, the linker searches there.
4. If the library is still not found, **LINK** prints an error message.

Examples

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

In the first example, the linker 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 the current drive, the current directory on drive B, and finally, directory `\lib` on drive D.

```
SET LIB=C:\lib;U:\system\lib
LINK file,,file.map,math+common
```

In the second example, **LINK** will search the current directory, directory `\lib` on drive C, and directory `\system\lib` on drive U to find the libraries `math.lib` and `common.lib`.

3.2.5 The 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 shown in this example:

```
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, the linker gives you the address of the program entry point.

If you have specified the **/MAP** option in the **LINK** command line, the linker adds a public-symbol list to the map file. The symbols are presented twice: once in alphabetical order, then in the order of their load addresses. The list has the general form shown in the following example:

```
 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 addresses 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 (see Sections 3.3.10 and 3.3.11) and the program's code and data combined do not exceed 64K, 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 the 20-bit address of TEMPLATE 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 the linker creates this file, it displays the following message:

```
VM.TMP has been created.
Do not change diskette in drive letter
```

Note that *letter* will be the proper drive name. After this message appears, you must not remove the disk from the drive specified by *letter* until the link session ends. The **/PAUSE** option cannot be used if a temporary file is created. After **LINK** has created the executable file, it deletes the temporary file automatically.

Warning

Do not use the file name **VM.TMP** for your own files. When the linker 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:

Option	Action
/HELP	Shows options list
/PAUSE	Pauses during linking
/EXEPACK	Packs executable file
/MAP	Creates public symbol map
/LINENUMBERS	Copies line numbers to map file
/NOIGNORECASE	Preserves case sensitivity in names
/NODEFAULTLIBRARYSEARCH	Overrides default libraries
/STACK	Sets stack size
/CPARMAXALLOC	Sets maximum allocation space
/HIGH	Sets high load address
/DSALLOCATE	Allocates data group
/NOGROUPASSOCIATION	Sets group association override

/OVERLAYINTERRUPT	Sets overlay interrupt
/SEGMENTS	Sets maximum number of segments
/DOSSEG	Specifies MS-DOS segment ordering

You can abbreviate option names as long as your abbreviations contain enough letters to distinguish the specified option from other options. Minimum abbreviations are listed for each option.

Many of the **LINK** options set values in the MS-DOS program header. You will understand these options better if you understand how the header is organized. The program header is described in the *Microsoft MS-DOS Programmer's Reference Manual* and in some reference books on MS-DOS.

3.3.1 Viewing the Options List

Syntax

/HELP

The **/HELP** option causes **LINK** to write a list of the available options to the screen. This may be convenient if you need a reminder of the available options. You should not give a file name when using the **/HELP** option.

Minimum abbreviation: **/HE**

Example

```
LINK /HELP
```

3.3.2 Pausing to Change Disks

Syntax

/PAUSE

The **/PAUSE** option causes **LINK** to pause before writing the executable file to disk so that you can swap disks before the linker writes the executable (**.EXE**) file to disk.

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

```
About to generate .EXE file
Change diskette in drive letter and press <ENTER>
```

Note that *letter* is the proper drive name. This message appears after the linker has read data from the object files and library files, and after it has written data to the map file, if one was specified. **LINK** resumes processing when you press the RETURN key. After **LINK** writes the executable file to disk, the following message appears:

```
Please replace original diskette
in drive letter and press <ENTER>
```

Minimum abbreviation: **/P**

Note

Do not remove the disk used for the **VM.TMP** file, if one has been created. If the temporary disk message appears when you have specified the **/PAUSE** option, you should press CONTROL-C to terminate the **LINK** session. Rearrange your files so that the temporary file and the executable file can be written to the same disk, then try again.

Example

```
LINK file/PAUSE,file,,\lib\math
```

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

3.3.3 Packing Executable Files

Syntax

/EXEPACK

The /**EXEPACK** option directs **LINK** to remove sequences of repeated bytes (typically nulls) and optimize the load-time relocation table before creating the executable file. Executable files linked with the option may be smaller, and thus load faster than files linked without the option. However, the Microsoft Symbolic Debug Utility (**SYMDEB**) cannot be used with packed files.

The /**EXEPACK** option will not always save a significant amount of disk space (and may sometimes actually increase file size). Programs that have a large number of load-time relocations (about 500 or more) and long streams of repeated characters will usually be shorter if packed. If you're not sure if your program meets these conditions, try linking it both ways and compare the results.

Minimum abbreviation: /E

Example

```
LINK program /E ;
```

This example creates a packed version of file `program.exe`.

3.3.4 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 the linker. For a complete description of the listing-file format, see Section 3.2.5. The /**MAP** option is required if you want to used **SYMDEB** for symbolic debugging (see Section 4.2).

Note

If you do not specify a map file in a **LINK** command, you can use the **/MAP** option to force the linker to create a map file. **LINK** gives the forced map file the same file name as the first object file specified in the command and the default extension **.MAP**.

Minimum abbreviation: /M

Example

```
LINK file,,/MAP;
```

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

3.3.5 Copying Line Numbers to the Map File

Syntax

/LINENUMBERS

The **/LINENUMBERS** option directs the linker to copy the starting address of each program source line to a map file. The starting address is actually the address of the first instruction that corresponds to the source line. The **MAPSYM** program can be used to copy line-number data to a symbol file, which can then by used by **SYMDEB**.

The linker copies the line-number data 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 in some high-level-language compilers, including Microsoft FORTRAN and Pascal, versions 3.0 and later, and Microsoft C Version 2.0 and later.

MASM does not copy line-number information to the object file. If an object file has no line-number information, the linker will ignore the **/LINENUMBERS** option.

Note

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

Minimum abbreviation: **/LI**

Example

```
LINK file/LINENUMBERS,,em+slibfp
```

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

3.3.6 Preserving Lowercase

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. When you use the **/NOIGNORECASE** option, the linker treats `TWO`, `Two`, and `two` as different 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 the linker will do the same.

If you are linking modules created with **MASM** to modules created with a case-sensitive language such as C, make sure public symbols have the same sensitivity in both modules. For example, you could make all variables in C distinctive by spelling, regardless of case, and then link without the

/**NOIGNORECASE** option. Another alternative would be to use the /**ML** or **MX** option to make public variables in **MASM** case-sensitive. Then link with the /**NOIGNORECASE** option.

Minimum abbreviation: /**NOI**

Example

```
LINK file1+file2/NOI,,,em+mlibfp
```

This command causes the linker 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 `\Slibc.lib` located in the `\lib` directory. The C language expects upper- and lowercase letters to be treated as distinct.

3.3.7 Ignoring Default Libraries

Syntax

/NODEFAULTLIBRARYSEARCH

The /**NODEFAULTLIBRARYSEARCH** option directs the linker 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 is linked with the program. Using this option overrides these default libraries and lets you explicitly name the libraries you want by including them on the **LINK** command line.

Minimum abbreviation: /**NOD**

Example

```
LINK startup+file/NOD,,,em+slibfp+slibc
```

This example links the object files `startup.obj` and `file.obj` with routines from the libraries `em`, `slibfp`, and `slibc`. Any default libraries that may have been named in `startup.obj` or `file.obj` are ignored.

3.3.8 Setting the Stack Size

Syntax

/**STACK:***size*

The /**STACK** option sets the program stack to the number of bytes given by *size*. The linker 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, the linker uses the given *size* in place of any value it may have calculated.

The *size* can be any positive integer value in the range 1 to 65535. The value can be a decimal, octal, or hexadecimal number. Octal numbers must begin with a zero. Hexadecimal numbers must begin with a leading zero followed by a lowercase x. For example, `0x1B`.

The stack size can also be changed after linking with the **EXEMOD** utility. See Appendix C.

Minimum abbreviation: /**ST**

Examples

```
LINK file/STACK:512,,;
```

The first example sets the stack size to 512 bytes.

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

The second example sets the stack size to 255 (FFh) bytes.

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

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

3.3.9 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.

LINK normally sets the maximum number of paragraphs to 65535. Since this represents all addressable 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.

The *number* can be any integer value in the range 1 to 65535. It must be a decimal, octal, or hexadecimal number. Octal numbers must begin with a zero. Hexadecimal values must begin with a leading zero followed by a lowercase x. For example, `0x2B`.

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 needed. The minimum number of paragraphs needed by a program is never less than the number of paragraphs of code and data in the program.

You can also change the maximum allocation after linking with the **EXEMOD** utility. See Appendix C.

Note

> The **/CPARMAXALLOC** option can be used to link files before debugging so that the **SYMDEB** Shell command (!) can be used. See Section 4.6.26.

Minimum abbreviation: **/C**

Examples

```
LINK file/C:15,,;
```

The first example sets the maximum allocation to 15 paragraphs.

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

The second example sets the maximum allocation to 255 (FFh) paragraphs.

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

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

3.3.10 Setting a High Start Address

Syntax

/HIGH

The **/HIGH** option sets the program's starting address to the highest possible address in free memory. If the **/HIGH** option is not given, the program's starting address is set as low as possible in memory.

Minimum abbreviation: **/H**

Example

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

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

3.3.11 Allocating a Data Group

Syntax

/DSALLOCATE

The **/DSALLOCATE** option directs the linker 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 appear 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. The linker 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,,,em+mlibfp
```

This example directs the linker 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.

3.3.12 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

This option exists strictly for compatibility with older versions of FORTRAN and Pascal (Microsoft version 3.13 or earlier, or any IBM version prior to 2.0). The **/NOGROUPASSOCIATION** option should never be used except to link with object files produced by those compilers, or with the run-time libraries that accompany the old compilers.

Minimum abbreviation: **/NOG**

3.3.13 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 a leading zero followed by a lowercase x. For example, `0x3B`.

MASM does not have an overlay manager. Therefore this option can only be used if you are linking with a run-time module from a language compiler that does support overlays. Check your compiler documentation, as this option is not appropriate for use with some compilers.

Note

You should not use interrupt numbers that conflict with the standard MS-DOS interrupts.

Minimum abbreviation: **/O**

Examples

```
LINK file/O:255,,,87+slibfp
```

The first example sets the overlay interrupt number to 255.

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

The second example sets the overlay interrupt number to 255 (FFh).

```
LINK startup+file,/O:0377,,em+mlibfp
```

The final example sets the overlay interrupt number to 255 (377 octal).

3.3.14 Setting the Maximum Number of Segments

Syntax

/SEGMENTS:*number*

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

If **/SEGMENTS** is not given, the linker allocates enough memory space to process up to 128 segments. If your program has more than 128 segments, you will need to set the segment limit higher to increase the number of segments **LINK** can process. If you get the following **LINK** error message:

```
Segment limit set too high
```

you should set the segment limit lower.

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 a leading zero followed by a lowercase x. For example, `0x4B`.

Minimum abbreviation: **/SE**

Example

```
LINK file/SE:192,,;
```

The first example sets the segment limit to 192.

```
LINK moda+modb,run/SEGMENTS:0xff,ab,em+mlibfp;
```

The second example sets the segment limit to 255 (FFh).

3.3.15 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 other 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.

The normal segment order when the **/DOSSEG** option is not used is explained in Section 3.4.3.

Minimum abbreviation: **/DO**

Example

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

This command causes the linker 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 the linker copies segments to the executable file defines the order and manner in which the segments will be loaded into memory.

You can tell the linker how to link a program's segments by giving segment attributes with a **SEGMENT** directive or by using the **GROUP** directive to form segment groups. These directives define group associations, classes, and align 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 options.

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

3.4.1 Alignment of Segments

The linker uses a segment's align type to set the starting address for the segment. The align 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 256, respectively. The default align type is **para**.

When the linker encounters a segment, it checks the align type before copying the segment to the executable file. If the align type is **word, para**, or **page**, the linker checks the executable image to see if the last byte copied ends at an appropriate boundary. If not, **LINK** pads the image with extra null bytes.

3.4.2 Frame Number

The linker computes a starting address for each segment in a program. The starting address is based on a segment's align type 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 (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** align types, the offset may be nonzero. The offset is always zero for **para** and **page** align types.

The frame number of a segment can be obtained from a **LINK** file. 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 the linker encounters two or more segments having the same class name. Segments having identical class names belong to the same class type, and are copied to the executable file as contiguous blocks.

Segment loading order and methods of controlling loading order by assigning class types are discussed in more detail in Section 3.4.3 of the *Microsoft Macro Assembler Reference Manual.*

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**, **common**, **memory**, **at**, and **private**. Combine types are also described in Section 3.4.2 of the *Microsoft Macro Assembler Reference Manual.*

If a segment has combine type **public**, the linker 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.

The linker preserves each individual segment's align type. This means that even though the segments belong to a single, large segment, the code and data in the segments retain their original align type. If the combined segments exceed 64K, **LINK** displays an error message.

If a segment has combine type **stack**, the linker carries out the same combine operation as for **public** segments. The only difference 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 you use the **stack** type for stack segments, you do not need to give instructions that load the segment into the **SS** register.

If a segment has combine type **common**, the linker automatically combines 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 of the combined segments.

The linker treats segments with combine type **memory** exactly like segments with combine type **public**. **MASM** provides combine type **memory** for compatibility with linkers that support a separate combine type for **memory** segments.

A segment has combine type **private** 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 permit non-contiguous segments that do not belong to the same class to 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.

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 64K.

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, the linker may place segments that do not belong to the group in the same 64K of memory. Although **LINK** does not explicitly check that all segments in a group fit within 64K of memory, the linker is likely to encounter a "fixup-overflow" error if this requirement is not met.

Groups, and how to define them, are discussed in Section 3.6 of the *Microsoft Macro Assembler Reference Manual.*

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, the linker can "fix up" any unresolved references to labels and variables. To fix up unresolved references, the linker 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 four 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 64K block of memory.

A short reference occurs in **JMP** instructions that attempt to pass control to labeled instructions that are in the same segment or group. The target instruction must be no more than 128 bytes from the point of reference. The linker 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 if the target is more than 128 bytes distant (in either direction).

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

Chapter 4
SYMDEB: A Symbolic Debug Utility

4.1 Introduction

The Microsoft Symbolic Debug Utility (**SYMDEB**) is a debugging program that helps you test executable files. You can display and execute program code, set "breakpoints" that stop the execution of your program, examine and change values in memory, and debug programs that use the floating-point emulation conventions used by Microsoft languages.

SYMDEB lets you refer to data and instructions by name rather than by address. **SYMDEB** can access program locations through addresses, global symbols, or line-number references, making it easy to locate and debug specific sections of code.

You can debug C, Pascal, and FORTRAN programs at the source-file level as well as at the machine level. You can display the source statements of a program, the disassembled machine code of the program, or a combination of source statements and disassembled machine code. **SYMDEB** accepts source line numbers as arguments to commands for displaying and changing data, setting breakpoints, and tracing execution.

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

4.2 Setting Up for Symbolic Debugging

SYMDEB is a useful tool even without its symbolic-debugging features. If you wish to use it as a nonsymbolic debugger, no setup is necessary. Simply start **SYMDEB** without a symbol file, as described in Section 4.3. However, if you wish to take full advantage of **SYMDEB**'s symbolic features during program development, you must first set up a symbol file that can be used by **SYMDEB**.

The steps for setting up a symbol file vary depending on whether you are developing your program with the Microsoft Macro Assembler (**MASM**) or with a compatible high-level language such as Microsoft Pascal, Microsoft C, or Microsoft FORTRAN. This chapter concentrates on the techniques for debugging programs prepared with **MASM**, but it also briefly covers the **SYMDEB** features that apply only to high-level-language programs.

All symbols to be used during debugging must be declared public. This is done automatically by most high-level-language compilers. However, you must do it yourself when developing programs with **MASM**.

4.2.1 Setting Up for Symbolic Debugging when Using MASM

The following assemblers are compatible with **SYMDEB**, and can be used for symbolic debugging:

Microsoft Macro Assembler, Version 1.0 and later

IBM Personal Computer Macro Assembler, Version 1.0 and later

To prepare symbol files when developing programs with a compatible assembler, follow these steps:

1. Declare public any symbols that you may wish to use in **SYMDEB**. Symbols that you may want to declare include procedure names, variable names, and labels. Segment and group names should not be declared public. They are automatically included in the map file and can be used during debugging.

   You may want to insert symbols in your program to use as breakpoints in **SYMDEB**, even though these symbols are not actually used by your program. For example, you could put a label in the code segment at a key point, even though that label is never used by a control instruction such as **JMP** or **LOOP**.

   For example, you could include the following lines in your source file before assembly:

   ```
   public   prompt,namebuf,fname,buffer  ;Data variables
   public   entry,get_file,open_file,ok  ;Code labels
   ```

2. Assemble your source file with **MASM**. You should probably specify a list file in the **MASM** command line and then print a copy of it. This is not necessary, but debugging is usually easier if you can refer to a listing. For example, type:

   ```
   MASM test,,;
   ```

3. Link the object file to produce an executable version of the program. Include a map (**.MAP**) file and the **/MAP** option in the **LINK** command line. It is not enough to specify a map file. You must also use the **/MAP** option. If you do not, you will get an error message when you try to create a symbol file with the **MAPSYM** program. For example, type:

```
LINK test,,/MAP;
```

4. Use the **MAPSYM** program to create a symbol file, as described in Section 4.2.3. For example, type:

```
MAPSYM test
```

SYMDEB is now ready for symbolic debugging as described in Section 4.3.2.

4.2.2 Setting Up for Symbolic Debugging when Using a Language Compiler

The following compilers are compatible with **SYMDEB** and can be used for symbolic debugging:

Microsoft FORTRAN, Version 3.0 and later

Microsoft Pascal, Version 3.0 and later

Microsoft C, Version 2.0 and later

Microsoft Macro Assembler, Version 1.0 and later

Microsoft BASIC Compiler, Version 1.0 and later

Microsoft Business BASIC Compiler, Version 1.0 and later

IBM Personal Computer FORTRAN, Version 2.0 and later

IBM Personal Computer Pascal, Version 2.0 and later

IBM Personal Computer Macro Assembler, Version 1.0 and later

IBM Personal Computer BASIC Compiler, Version 1.0 and later

However, not all these compilers support the source-line display capabilities of **SYMDEB**. Compilers that can generate the needed source-line information for **MAPSYM** and **SYMDEB** include:

Microsoft FORTRAN, Version 3.0 and later

Microsoft Pascal, Version 3.0 and later

Microsoft C, Version 2.0 and later

IBM Personal Computer FORTRAN, Version 2.0 and later

IBM Personal Computer Pascal, Version 2.0 and later

If you have a compatible compiler, follow these steps to prepare a symbol file:

1. Compile your source file. If your compiler has an optimization feature, debugging will be easier if you use the option that disables optimization. If your compiler can write line-number information to the object file, you may need to use an option in the command line to enable line numbers.
2. Link the object file to produce an executable version of the program. Use the **/MAP** option in the **LINK** command line. If your compiler supports source-line display, you should also use the **/LINENUMBERS** option.
3. Use the **MAPSYM** program to produce a symbol file as described in Section 4.2.3.
4. Start **SYMDEB** for symbolic debugging as described in Section 4.3.2.
5. Use the **SYMDEB** Go command (**G**) to execute the program up to the first procedure or function. This takes you past the start-up routine from the standard library of the high-level language you are using. Normally you will not want to trace through this initial routine. You can usually start debugging at the start of your program.

   In C programs, the first function is always `_main` (C adds a leading underscore to procedure names such as `main`). In FORTRAN, the first procedure is `MAIN`. In Pascal the first procedure is the one that names the program (the first procedure in the source code).

Examples

```
MSC /Zd /Od test.c;
LINK test,,/MAP/LINE;
MAPSYM test
SYMDEB test.sym test.exe
-G _main
```

The first example shows how to prepare a program for symbolic debugging using Microsoft C, Version 3.0 or later. The **/Zd** option directs the compiler to write line-number information to the object file, and the **/Od** option turns off optimization.

```
PAS1 /L test.pas;
PAS2
PAS3
LINK test,,/MAP/LINE;
```

```
MAPSYM test
SYMDEB test.sym test.exe
-G test
```

The preceding example shows how to prepare a program for symbolic debugging using Microsoft Pascal, Version 3.3 or later. The **/L** option directs the compiler to write line-number information to the object file. After starting **SYMDEB**, you will usually want to "Go" to the first procedure in the source code (the one that names the program).

```
FOR1 test.for;
PAS2
PAS3
LINK test,,/MAP/LINE;
MAPSYM test
SYMDEB test.sym test.exe
-G MAIN
```

The final example shows how to prepare a program for symbolic debugging using Microsoft FORTRAN, Version 3.3 or later. The compiler automatically writes line-number information to the object file. After starting **SYMDEB**, you will usually want to "Go" to the `MAIN` procedure.

4.2.3 Creating a Symbol File with the MAPSYM Program

Symbol files containing data for symbolic debugging can be created with the Microsoft Symbol File Utility (**MAPSYM**). The program converts the contents of the program's symbol (**.MAP**) file into a form suitable for loading with **SYMDEB**. Symbol files created with **MAPSYM** can contain up to 10000 symbols per segment and as many segments as are allowed by machine memory.

The **MAPSYM** command line has the form:

MAPSYM [[**/L**|**-L**]] *mapfilename*

The *mapfilename* is the file name (and optionally, the path name) for a symbol (**.MAP**) file created during linking. If you do not specify a file name extension, **.MAP** will be assumed.

The symbol-map file can be created by specifying a map file and the **/MAP** option when linking. If your compiler writes line-number information to the object file, you should also use the **/LINENUMBERS** option.

The /L option is the only one available with **MAPSYM**. It directs **MAPSYM** to display information on the screen about the conversion. The information includes the names of groups defined in the program, the program start address, the number of segments, and the number of symbols per segment. The /L option can also be specified as **-L**, **/l**, or **-l**.

Example

```
MAPSYM /L file
```

MAPSYM takes data from `file.map` to create `file.sym` on the current drive and directory. Information about the conversion is sent to the screen.

Note

The symbol (**.SYM**) file is always created on the current drive and directory. You cannot specify a destination in the command line, and you should not give a drive or directory for the map file. If you wish to place the symbol and map files on one drive while the **MAPSYM** program is on another, you should call the **MAPSYM** program from the drive with the map file. For example, to create `test.sym` on Drive B when the **MAPSYM** program is on Drive A and `test.map` is on Drive B, type:

```
A>B:
B>A:MAPSYM test
```

4.3 Starting SYMDEB

To start **SYMDEB**, enter the **SYMDEB** command line at the MS-DOS command prompt. The **SYMDEB** command line has the following form:

SYMDEB ⟦*options*⟧ ⟦*symbolfiles*⟧ ⟦*executablefile*⟧ ⟦*arguments*⟧

The *options* are one or more of the options described in Section 4.4. The *symbolfiles* are the names of symbol files. The *executablefile* is the name of a binary or executable file to be loaded by **SYMDEB**. The *arguments* are parameters that you want to pass to the *executablefile.*

Once started, **SYMDEB** displays a start-up message. The message is followed by the **SYMDEB** command prompt (–). When you see the prompt you can enter **SYMDEB** commands.

4.3.1 Starting SYMDEB with Only an Executable File

You can direct **SYMDEB** to load an executable file (**.EXE**, **.HEX**, **.COM**, or **.BIN**) by giving the name of the file on the **SYMDEB** command line. You can do this if you do not need to use symbol files, or if you are examining a program for which you do not have source code.

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

Note

> If the file is an **.EXE** or **.HEX** file, the MS-DOS executable file header will be stripped off during loading. Therefore, the program size will not match the file size, as it will for **.COM** and **.BIN** files.

Example

```
SYMDEB snap.com
Microsoft Symbolic Debug Utility
Version 4.00
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-R
AX=0000  BX=0000  CX=2975  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=2110  ES=2110  SS=2110  CS=2110  IP=0100   NV UP EI PL NZ NA PO NC
2110:0100 E91F29        JMP    2A22
-
```

In the example above, **SYMDEB** is started with a **.COM** file. Notice the line `Processor is [8086]` in the start-up message. This indicates that the system running **SYMDEB** has the 8086 (or the similar 8088) processor. The message would show 80186 or 80286 if the system had one of those processors.

The Register command (**R**) has been entered after start-up to show the initial status of the registers. Notice that **CX** contains 2975 (10613 decimal), indicating that the length of the program is 10613 bytes. You can confirm this by leaving **SYMDEB** and checking the file length with the MS-DOS **DIR** command. File length will match for **.COM** files, but not for **.EXE** files.

4.3.2 Starting SYMDEB for Symbolic Debugging

When developing and debugging programs, you may want to load symbol information along with an executable file so that you can refer to data and instructions by name rather than by address. Start **SYMDEB** for symbolic operation by specifying one or more symbol files on the command line. Specifying a symbol file directs **SYMDEB** to load the named file and allows you use the symbols defined by that file in **SYMDEB** commands.

You may specify more than one symbol file. Multiple symbol files are typically used with programs that consist of several separate executable files (such as programs that call overlays, execute other programs, or use device drivers). You must make sure that all symbol files are specified before the executable file. Any files specified after the executable file are assumed to be program arguments.

If you load multiple symbol files, only one of them will be opened initially. If one of the symbol files has the same name as the executable file, it will be opened. Otherwise, the first symbol file specified in the command line will be opened. During the **SYMDEB** session, you may use the Open Map command (**XO**) to open a different symbol file. The previous symbol file will be closed, since only one can be open at a time. See Section 4.6.17 for more information on opening symbol files.

You need not specify an executable file when you load symbols. You might load symbols without an executable file to debug a resident program, or if you wished to load the executable file later in the session using the Name command (**N**) and Load command (**L**).

Note

Do not rename symbol files and then attempt to load them in the **SYMDEB** command line. Renamed symbol files will have the wrong address when loaded.

Example

```
SYMDEB count.sym count.exe
-R
AX=0000  BX=0000  CX=0900  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=2125  ES=2125  SS=21C5  CS=2135  IP=0000   NV UP EI PL NZ NA PO NC
2135:0000 B84021          MOV     AX,DATA
-
```

In the example above, **SYMDEB** copies symbolic information from `count.sym` into memory, prepares the program header, then loads `count.exe`.

The **R** command has been entered to show the initial status of the registers. Notice that the **CX** register contains `0900` (2304 decimal). This is the length of the executable file minus the MS-DOS file header, which was stripped off during loading. (The **SYMDEB** start-up message would normally appear, but is omitted from this and other examples in the rest of the chapter.)

```
SYMDEB test1.sym test.sym test.exe
-
```

In the example above, **SYMDEB** copies symbolic information from the files `test1.sym` and `test.sym` into memory, prepares the program header, then loads `test.exe`. The symbol file `test.sym` is opened instead of `test1.sym` because it has the same name as the executable file.

4.3.3 Passing Arguments to a Loaded Program

You can pass one or more arguments to a program by typing the arguments immediately after the executable-file name on the **SYMDEB** command line. **SYMDEB** will copy all arguments to the program header in exactly the form you type them.

Example

```
SYMDEB ptest.exe param1 param2 param3 param4
-D 5D 9f
23B6:0050                                            50 41 52            PAR
23B6:0060  41 4D 31 20 20 20 20 20-00 00 00 00 00 50 41 52  AM1     .....PAR
23B6:0070  41 4D 32 20 20 20 20 20-00 00 00 00 00 00 00 00  AM2     ........
23B6:0080  1C 20 70 61 72 61 6D 31-20 70 61 72 61 6D 32 20  . param1 param2
23B6:0090  70 61 72 61 6D 33 20 70-61 72 61 6D 34 0D 00 0D  param3 param4...
-
```

In the example, the Dump command (**D**) has been entered to show the status of the program header after loading. The first and second parameters are parsed as file names into the default file control blocks. These blocks start at bytes 5Dh and 6Dh of the program header. The length of the parameter list is in the byte at 80h. An exact copy of the parameter list starts at byte 81h of the header. The program header is described in more detail in Section 4.6.16.

4.3.4 Starting SYMDEB without a File

You can start **SYMDEB** without a file by typing **SYMDEB**. When you start **SYMDEB** without a file name, it creates a program header, but does not attempt to load a program. You can then either create a program with the Assemble command (**A**) or Enter command (**E**), or you can use the Name command (**N**) and Load command (**L**) 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 (**IP**) to 0100h, clears all flags, and sets the remaining registers to zero.

Example

```
SYMDEB
-R
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=23B2  ES=23B2  SS=23B2  CS=23B2  IP=0100   NV UP EI PL NZ NA PO NC
23B2:0100 8AE5           MOV     AH,CH
-
```

In the example, the Register command (**R**) is entered after the start-up message to indicate the initial status of the registers.

4.4 Using SYMDEB Options

The following options can be entered on the **SYMDEB** command line:

Option	**Effect**
/IBM	Enable IBM-compatible mode
/K	Enable break key
/N	Enable non-masked interrupt
/S	Enable screen flip
/"*commands*"	Designates start-up commands

Options should be entered before the executable file on the command line so that **SYMDEB** will not interpret them as parameters. The option designator can be either a slash (/) or a dash (–), and the option letter can be specified with either upper- or lowercase letters.

Note

Files containing a dash in the file name must be renamed before use with **SYMDEB**. Otherwise, **SYMDEB** will interpret the dash as an option designator.

4.4.1 Designating IBM-Compatible Mode

Syntax

/I ¦ **–I**

The **/I** or **/IBM** option directs **SYMDEB** to use features available on IBM-compatible computers. The **/I** option is not necessary if you have an IBM Personal Computer since **SYMDEB** automatically checks the hardware on start-up. If **SYMDEB** does not find that the hardware is an

IBM Personal Computer, it assumes that the hardware is a generic MS-DOS machine, unless the /I option is used. Without the option, **SYMDEB** cannot take advantage of special hardware features such as the 8259 Interrupt Controller, IBM-style video display, and other capabilities of the IBM basic input and output system (BIOS).

Example

```
SYMDEB /I file.sym file.exe
```

4.4.2 Enabling the Interactive Breakpoint Key

Syntax

/K ¦ **-K**

The **/K** option enables the scroll-lock (break) key on IBM and compatible computers as an interactive breakpoint key. If the key is enabled, you can usually stop program execution by pressing it. For example, you could use the breakpoint key to get out of an endless loop started with the Go command (**G**).

The interactive breakpoint key acts like a hardware-activated interrupt key (as described in Section 4.4.3), except that it is less reliable. The interactive breakpoint key does not work in certain situations, such as when interrupts are turned off. If the program is waiting for input, press CONTROL-C rather than the BREAK key to interrupt program execution.

Note

If you have an IBM Personal Computer AT, the system request (SYS REQ) key can be used as an interactive break key even if you do not use the **/K** option.

Example

```
SYMDEB /K file.sym file.exe
```

4.4.3 Enabling Non-Maskable Interrupts for Non-IBM Hardware

Syntax

/N ¦ –N

The /N option enables you to use non-maskable interrupt break systems on non-IBM computers. To use non-maskable interrupts, your system must be equipped with the proper hardware. For example, you can use the /N option with these products:

- IBM Professional Debug Utility
- Software Probe (Atron Corp.)

SYMDEB only requires the hardware provided with these products; no additional software is needed. If you are using one of these products with a non-IBM system, you must use the /N option to take advantage of the break capability. You do not need to use the option if you are using an IBM Personal Computer. Using a non-maskable interrupt break system is more reliable than the interactive break key because its operation is independent of the state of interrupts and other conditions.

4.4.4 Enabling Screen Swapping

Syntax

/S ¦ -S

The /S option allows you to flip back and forth between a screen showing the debugger and a screen showing the program being debugged. This feature is particularly useful for graphics and other programs that send changing data to the screen. However, using the **/S** option does use up an additional 32K of system memory.

This option works only with IBM computers and some compatible computers. To use it with a compatible computer, you must also use the **/I** option in the command line. The **/S** option cannot be used with graphics modes that use more than 32K of memory.

Example

```
SYMDEB /I/S file.sym file.exe
```

The example above assumes an IBM-compatible computer. If you have an IBM Personal Computer, you do not need the **/I** option.

4.4.5 Specifying Start-Up Commands

Syntax

/"*commands*" ¦ –"*commands*"

The start-up command option directs **SYMDEB** to execute the commands contained within double quotation marks on start-up. This feature can be used to start **SYMDEB** from a batch file or to execute a series of commands that you use at the beginning of every **SYMDEB** session. A semicolon (;) separates each command from other commands in the list.

Examples

```
SYMDEB /"d40;u;r" file.exe
```

In the first example, **SYMDEB** loads `file.exe`, dumps the program header starting at 40h, unassembles the first few instructions, and shows the start-up status of the registers.

```
SYMDEB /"s+;g _main;v" cprog.sym cprog.exe
```

In the second example, **SYMDEB** loads the symbol file `cprog.sym` and the executable file (written in C) `cprog.exe`. Next, it sets the display mode to show source lines, executes the program up to the start of the `_main` function (always the first function in C programs), and displays the first few source lines. If the program were written in Pascal or FORTRAN, you would use the Go command (**G**) in the quoted commands to execute up to the first procedure of the program.

4.5 Specifying Parameters for Commands

SYMDEB commands have always have the following general form:

commandname parameters

Note that *commandname* is a one- or two-character command name, and *parameters* are numbers, symbols, or expressions that represent values or addresses to be used by the command. Any combination of upper- and lowercase letters may be used in commands and parameters. In most cases the first *parameter* can be placed immediately after *commandname* with no space between them.

The number of parameters used with each command depends on the command. If a command takes two or more parameters, you must separate them with commas (,) or with spaces.

Examples

```
DS _avg L 10
U .22
F ds:100,110 ff,fe,01,00
```

Sections 4.5.1–4.5.8 describe the different kinds of command parameters in detail.

4.5.1 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, an underscore (_), a question mark (?), an at sign (@), or a dollar sign ($).

When using **SYMDEB** to debug high-level-language programs, you should familiarize yourself with any conventions your compiler uses for designating symbols. For example, the Microsoft C Compiler automatically adds a leading underscore to the beginning of every global name.

Symbols are only available for debugging when the symbol file that defines their names and values has been loaded.

Notes

> **SYMDEB** is case-insensitive; it treats corresponding upper- and lower-case letters as the same letter. Symbols whose spellings differ only in case are treated as the same symbol. If a symbol 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. Be careful to give symbols unique names that do not mimic or conflict with instructions, register names, or hexadecimal numbers.

Examples

```
_main
next_loop
DGROUP
startup
code_seg
```

The symbols above are valid. Avoid using symbols such as the following, because they will cause problems, either during assembly or with **SYMDEB**:

```
AX          ; Don't use register name
faa         ; Don't use hexadecimal number
ADD         ; Don't use instruction name
```

4.5.2 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* plus 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 specified, **H** (hexadecimal) is assumed. The radix can be specified with either an upper- or lowercase letter (lowercase is used as a convention in examples). The following table lists the digits that can be used with each radix:

Table 4.1

Radixes for SYMDEB

Radix	Type	Digits
Y	Binary	0 1
O or **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

Hexadecimal numbers have precedence over symbols. Thus `FAA` is always interpreted as a hexadecimal number. Be careful not to give such ambiguous names to symbols.

Examples

```
0111111y         77q     63t     03Fh    3F
01001010100101y  11245o  4773t   12A5h   12A5
```

4.5.3 Addresses

Syntax

segment:*offset*

An address is a combination of two 16-bit values, one representing a segment address, the other a segment offset. When combined, the values 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, the default segment address is the current contents of the **DS** register. However, for the Assemble (**A**), Go (**G**), Load (**L**), PTrace (**P**), Trace (**T**), Unassemble (**U**), and Write (**W**) commands, the default segment address is the contents of the **CS** register.

Addresses can be specified as a positive or negative offset of a symbol. For example, the byte 5 bytes beyond the symbol `print` can be specified as `print+5`.

Examples

```
CS:0100
04BA:IP
CS:_main
pixel-10
DGROUP:count
buffer_1
```

4.5.4 Address Range

Syntax

startaddress endaddress

A range is a pair of memory addresses that bound a sequence of contiguous memory locations. Note that the span of the range is from *startaddress* to *endaddress*, inclusive.

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

```
_main _main+20
CS:100 110
get_out-30 get_out
buffer1 buffer2
14D stop
```

4.5.5 Object Range

Syntax

startaddress **L** *count*

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 *startaddress* specifies the address of the first object in the list and **L** *count* specifies the number of objects in the list.

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

Examples

```
seg1:table L 10
```

If you specified the sample range above with the Dump Bytes command, **SYMDEB** would dump the first 10 bytes beginning at `seg1:table`. If you specified the same range with the Unassemble command, **SYMDEB** would unassemble the next 10 instructions starting at `seg1:table`.

4.5.6 Line Numbers

Syntax

.+*number*|−*number*
.[[*filename*:]]*number*
.*symbol*[[+*number*|−*number*]]

A line number is a combination of decimal numbers, file names, and symbols that specifies a unique line of text in a program source file. Line number designations always start with a dot(.). Line numbers can only be used with programs developed with compilers that copy line-number data to the object file. See Section 4.2.2. Programs developed with **MASM** or an incompatible compiler cannot use line numbers.

In the first form shown in the syntax above, 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 specified, the new line is closer to the end of the source file. If the minus sign (−) is specified, the new line is closer to the beginning. **SYMDEB** displays an error message if there is no current line number, or if no source line exists for the specified line number.

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

In the third form, the combination specifies a symbolic line number. The *symbol* can be any instruction or procedure label. If *number* is specified, the *number* is an offset (in lines) from the specified label or procedure name to the new line. If the plus sign (+) is specified, the new line is closer to the end of the source file. If the minus sign (−) is specified, the new line is closer to the beginning. **SYMDEB** displays an error message if the *symbol* does not exist, or if 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'
```

A symbol such as `_main` can also be used to specify a line number. The symbol `_main` is equivalent to `._main`. Note, however, that `_main+3` specifies an address that is 3 *bytes* from `_main`, but `._main+3` specifies a source line that is 3 *lines* from `_main`.

4.5.7 Strings

Syntax

\&'*characters*'
"*characters*"

A string represents a list of ASCII values. It can be any 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 as part of the string, it must be specified 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."
```

4.5.8 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 unary operators in Table 4.2, or binary operators in Table 4.3.

Unary address operators assume **DS** as the default segment for addresses. 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.

Table 4.2

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 specified address	
WO	Low-order word from specified address	
DW	Double word from specified address	
POI	Pointer from specified address (same as **DW**)	
PORT	1 byte from specified port	
WPORT	Word from specified port	Lowest

Table 4.3

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

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)
```

4.6 Using SYMDEB Commands

The following table lists all **SYMDEB** commands.

Table 4.4
SYMDEB Commands

Command	Command Name	Command	Command Name
?	Display Values, Display Help	**H**	Hex
!	Shell Escape	**I**	Input
.	Source Line Display	**K**	Stack Trace
< {	Redirect Input	**L**	Load
> }	Redirect Output	**M**	Move
= ~	Redirect Input and Output	**N**	Name
*	Comment	**O**	Output
A	Assemble	**P**	PTrace
BC	Breakpoint Clear	**Q**	Quit
BD	Breakpoint Disable(s)	**R**	Register
BE	Breakpoint Enable	**S**	Search, Set Source Mode
BL	Breakpoint List	**T**	Trace
BP	Breakpoint Set	**U**	Unassemble
C	Compare	**V**	View
D	Dump	**W**	Write
E	Enter	**X**	Examine Symbol Map
F	Fill	**XO**	Open Symbol Map
G	Go	**Z**	Set Symbol Value

When entering **SYMDEB** commands, you can use any of the special editing keys described in the *Microsoft MS-DOS User's Guide.* You can also press CONTROL-C to abort execution of a **SYMDEB** command, or press CONTROL-S to suspend execution of a **SYMDEB** command.

CONTROL-C and CONTROL-S can abort or suspend execution of the Go command (**G**) if the program being debugged is engaged in input or output. If the program is not engaged in input or output, the only way to stop execution is with the break key if the **/K** option was used, or with a hardware interrupt device if one is installed on your system. See Section 4.4.2 for more information on the **/K** option and Section 4.4.3 for information on hardware interrupt devices.

4.6.1 Assemble Command

Syntax

A⟦*address*⟧

The Assemble command (**A**) assembles 8086-family (8086, 8087, 8088, 80186, 80287, 80286-unprotected) instruction mnemonics and places the resulting instruction codes into memory at the specified *address.* The only 8086-family mnemonics that cannot be assembled are 80286 protected-mode mnemonics. If no *address* is specified, the assembly starts at the address specified by the current values of the **CS** and **IP** registers.

When you type the Assemble command, the specified address is displayed. **SYMDEB** then waits for you to enter a new instruction in the standard 8086-family instruction-mnemonic form. You can enter instructions in either upper- or lowercase, or both (the examples use lowercase for instructions and data, and uppercase for reserved words).

To assemble a new instruction, type the desired mnemonic and press the RETURN key. **SYMDEB** assembles the instruction into memory and displays the next available address. To conclude assembly and return to the **SYMDEB** prompt, press the RETURN key only.

If an instruction you enter contains a syntax error, **SYMDEB** displays the message `Error`, redisplays the current assembly address, and waits for you to enter a correct instruction.

The following rules govern entry of instruction mnemonics:

1. The far return mnemonic is **RETF**.
2. String manipulation mnemonics must explicitly state the string size. For example, use **MOVSW** to move word strings and **MOVSB** to move byte strings.
3. **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, as shown in the following examples:

   ```
   jmp     502
   jmp     NEAR 505
   jmp     FAR  50A
   ```

   The **NEAR** prefix can be abbreviated to **NE**, but the **FAR** prefix cannot be abbreviated.
4. **SYMDEB** cannot tell whether some operands refer to a word memory location or to a byte memory location. In these cases, the data type must be explicitly stated with the prefix **WORD PTR** or **BYTE PTR**. Acceptable abbreviations are **WO** and **BY**. Two examples are shown below:

   ```
   mov     WORD PTR [bp],1
   mov     BYTE PTR [si-1],symbol
   ```

5. **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. Two examples are shown below:

   ```
   mov     ax,21
   mov     ax,[21]
   ```

   The first statement moves 21h into **AX**. The second statement moves the data at offset 21h into **AX**.
6. The **DB** opcode assembles byte values directly into memory. The **DW** opcode assembles word values directly into memory, as shown in the following examples:

   ```
   DB      1,2,3,4,"This is an example."
   DB      'This is a double quote: "'
   DB      "This is a single quote: '"
   DW      1000,2000,3000,"Bach"
   ```

7. **SYMDEB** supports all forms of register-indirect commands, as shown in the following examples:

```
add     bx,34[bp+2].[si-1]
pop     [bp+di]
push    [si]
```

8. All opcode synonyms are also supported, as shown in the following examples:

```
loopz   100
loope   100
ja      200
jnbe    200
```

   If you examine instructions with the Unassemble command (**U**), **SYMDEB** may show a synonymous instruction or opcode, rather than the one you entered.

9. Do not assemble and execute 8087 or 80287 instructions if your system is not equipped with one of these math coprocessors. The **WAIT** instruction, for example, will cause your system to hang up if you try to execute it without the appropriate chip.

Examples

```
-A
42BE:0100 mov   ah,2
42BE:0102 mov   dl,7
42BE:0104 int   21
42BE:0106 mov   ah,4C
42BE:0108 int   21
42BE:010A
-
```

The first example assembles a short program that beeps and returns to MS-DOS. Section 4.6.33 shows how to save this program to disk as a file called `bell.com`.

```
-U test L 2
CODE:TEST:
39B0:0040 89C3          MOV         BX,AX
-A test
39B0:0040 mov     cx,ax
39B0:0042
-U test L 2
CODE:TEST:
39B0:0040 89C1          MOV         CX,AX
-
```

The second example modifies the instruction at address `test` so that it moves data into the **CX** register instead of the **BX** register. The Unassemble command (U) is used to show the instruction before and after the assembly.

4.6.2 Breakpoint Commands

SYMDEB allows you to set and use "sticky" breakpoints. The five following commands govern breakpoint manipulation:

Command	Command Name
BP	Breakpoint Set
BC	Breakpoint Clear
BD	Breakpoint Disable
BE	Breakpoint Enable
BL	Breakpoint List

These commands are discussed in logical, rather than alphabetical, order in Sections 4.6.2.1–4.6.2.5.

4.6.2.1 Breakpoint Set Command

Syntax

BP ⟦*number*⟧ *address* ⟦*passcount*⟧ ⟦"*commands*"⟧

The Breakpoint Set command (**BP**) creates a "sticky" breakpoint at *address*. When encountered during program execution, sticky breakpoints stop the program execution and cause **SYMDEB** to display the current values of all registers and flags in the Register command (**R**) format (see Section 4.6.22). Sticky breakpoints, unlike breakpoints created by the Go command (**G**), remain in the program until removed using the Breakpoint Clear command (**BC**), or temporarily disabled using the Breakpoint Disable command (**BD**).

SYMDEB allows up to 10 sticky breakpoints (0 through 9). The *number* specifies which breakpoint is to be created. Spaces between **BP** and *number* are not allowed. If no *number* is specified, 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 *passcount* specifies the number of times the breakpoint is to be ignored before being taken. It can be any 16-bit value. The *commands* are an optional list of commands to be executed each time the breakpoint is taken. Each **SYMDEB** command in the list can include parameters, and is separated from the next command by a semicolon (;).

Examples

```
-BP do_again
-
```

The first example creates a sticky breakpoint at do_again.

```
-BP .19 3
-
```

The second example creates a sticky breakpoint at line 19 of the source file (or if there is no executable statement at line 19, at the first executable statement after line 19). The breakpoint is ignored three times before being taken.

```
-BP8 add
-
```

The third example creates breakpoint 8 at address add.

```
-BP 100 10
-
```

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

```
-BP 3206:2A02 "rdi di+1;g"
-
```

The final example increments the contents of the **DI** register by one whenever address 3206:2A02 is reached. Since neither the Register command (**R**) nor the Go command (**G**) stops to request input, the program will appear to execute normally, although program speed will decrease while the command is being executed.

4.6.2.2 Breakpoint Clear Command

Syntax

BC *list*|*

The Breakpoint Clear command (**BC**) permanently removes one or more previously set breakpoints. If *list* is specified, the command removes the breakpoints named in the list. The *list* can be any combination of integer values from 0 to 9. If * is specified, the command removes all breakpoints.

Examples

```
-BC 0 4 8
-
```

The first example removes breakpoints 0, 4, and 8.

```
-BC *
-
```

The second example removes all breakpoints.

4.6.2.3 Breakpoint Disable Command

Syntax

BD *list*|*

The Breakpoint Disable command (**BD**) 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 (**BE**).

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

Examples

```
-BD 0 4 8
-
```

The first example disables breakpoints 0, 4, and 8.

```
-BD *
-
```

The second example disables all breakpoints.

4.6.2.4 Breakpoint Enable Command

Syntax

BE *list*|*

The Breakpoint Enable command (**BE**) restores one or more breakpoints temporarily disabled by a Breakpoint Disable command (**BD**).

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

Examples

```
-BE 0 4 8
-
```

The first example enables breakpoints 0, 4, and 8.

```
-BE *
-
```

The second example enables all disabled breakpoints.

4.6.2.5 Breakpoint List Command

Syntax

BL

The Breakpoint List command (**BL**) lists current information about all breakpoints created by the Breakpoint Set command (**BP**). The **BL** 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 you are in source-line mode (see Section 4.6.25), the line number for each breakpoint is also shown.

The enable status can be `e` for enabled, `d` for disabled, or `v` for virtual. A virtual breakpoint is a breakpoint set at a symbol whose **.EXE** file has not yet been loaded.

If no breakpoints are currently defined, nothing is displayed.

Example

```
-BL
0 e 11BC:0036 [IGROUP:_main+0B (0036)] main.c:8
4 d 11BC:0100 [IGROUP:__cropzeros+08 (0100)] 0010 (000A)
8 e 11BC:0002 [IGROUP:_add] add.c:2 "DW;G"
-
```

The example above is taken from a C program in order to illustrate line numbers. Breakpoint 0 is enabled at address `IGROUP:_main+0B` (segment 11BC, offset 36). This address is at line 8 of source file `main.c`.

Breakpoint 4 is disabled at address `IGROUP:__cropzeros+08`. Since the breakpoint is disabled, the source line is not shown. This breakpoint initially had a pass count of 16 (10h) and now has 10 (0Ah) remaining passes to be taken before the breakpoint.

Breakpoint 8 is enabled at address `IGROUP:_add`. This address is at line 2 of source file `add.c`. It has no initial pass count. Whenever breakpoint 8 is reached, the command list `DW;G` (Dump Word and Go) is executed.

4.6.3 Comment Command

Syntax

**comment*

The Comment command is an asterisk (*) followed by text. **SYMDEB** echoes the text of the comment to the screen (or other output device). This command is useful in combination with the redirection commands to save or print commented copies of a **SYMDEB** session.

Example

```
-RCX 80
-* Change the count in CX to 80
 Change the count in CX to 80
-
```

4.6.4 Compare Command

Syntax

C *range address*

The Compare command (**C**) 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 100,01FF 300
39BB:102 0A 00 39BB:302
39BB:108 0A 01 39BB:308
-
```

The first example compares the block of memory from 100h to 1FFh with the block of memory from 300h to 3FFh. It indicates that the second and eighth bytes are different in the two areas of memory.

```
-C test L 100   test+100
-
```

The second example compares the 256 (100h) bytes starting at symbol `test` with the 256 bytes starting at the address 256 bytes beyond `test`. **SYMDEB** produces no output, so the bytes are the same.

4.6.5 Display Command

Syntax

? *expression*

The Display command (?) displays the value of *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 string characters will be shown as dots if their value is less than 32 (20h) or greater than 126 (7Eh).

The *expression* can be any combination of numbers, symbols, addresses, and operators. For a list of operators, see Section 4.5.8.

Examples

```
-? 9*8
0048h   00000048   (72)   "H"
-
```

The first example displays the value of the expression `9*8`.

```
-? .19
39E0:0017h   00039E17   (237079)   "."
-
```

The second example displays the address in memory of line 19 in the source file. The Display command is a convenient way to find addresses for source code.

```
-? CS:_main
39E0:0002h   00039E02   (237058)   "."
-
```

The third example displays the value of the symbolic address `CS:_main`.

```
-? WO DGROUP:_environ
2E36h  00002E36  (11830)  ".6"
-
```

The final example displays the word at the symbolic address `DGROUP:_environ`.

4.6.6 Dump Commands

SYMDEB has several commands for dumping data from memory to the screen (or other output device). The dump commands are listed below:

Command	Command Name
D	Dump
DA	Dump ASCII
DB	Dump Bytes
DW	Dump Words
DD	Dump Doublewords
DS	Dump Short Reals
DL	Dump Long Reals
DT	Dump Ten-Byte Reals

Sections 4.6.6.1–4.6.6.8 discuss these commands in logical, rather than alphabetical, order.

4.6.6.1 Dump Command

Syntax

D [[*address*|*range*]]

The Dump command (**D**) displays the contents of memory at the specified *address* or in the specified *range* of addresses. The Dump command dumps data in the format of the most recently entered dump command (as described in the next seven sections). If no other dump command has been entered, the default dump format is the format of the Dump Bytes command (**DB**).

The Dump command displays one or more lines, depending on the address or range specified. Each line displays the address of the first item displayed. The command always displays at least one value. If a *range* is specified, **SYMDEB** displays all values in the range. If neither *address* nor *range* is specified, **SYMDEB** dumps memory starting at the byte after the last byte dumped by a previous dump command. If no previous dump command has been used, **SYMDEB** dumps data starting from the current location of the instruction pointer (**IP**). If no segment is specified in an initial dump command, **SYMDEB** uses the **DS** register value as the default segment.

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 the first example, the Dump command displays the ASCII string at the address immediately following the string displayed by the Dump ASCII command. The Dump command uses the ASCII format because the last dump command was **DA** (Dump ASCII).

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

In the second example, the Dump command displays the word at the address `ds:324`. The format is words because the last dump command was Dump Words (**DW**).

4.6.6.2 Dump ASCII Command

Syntax

DA [[*address*|*range*]]

The Dump ASCII command (**DA**) displays the ASCII characters at a specified *address* or in a specified *range* of addresses. The command displays one or more lines of characters, depending on the *address* or *range* specified. Up to 48 characters per line are displayed. Unprintable characters, such as carriage returns and line feeds, are displayed as dots (**.**). ASCII characters below 32 (20h) and above 126 (7Eh) are unprintable.

If an *address* is specified, the command continues to display ASCII characters until the first null byte is encountered, or until 128 bytes have been displayed. If a *range* is specified, the command continues to display ASCII characters until the end of the range. If neither *address* nor *range* is specified, the command displays all characters up to the first null 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 specified number of characters has been displayed.

Examples

```
-DA DS:100 110
04BA:0100 A string..Text..
-
```

The first example displays the ASCII values of the bytes from `DS:100` to `DS:110`. Unprintable characters are shown as dots.

```
-DA
04BA:0111 Some letters
-
```

The second 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 prompt
294A:0000 Enter file name: $.
-
```

The final example displays the characters at the symbolic address `prompt`.

4.6.6.3 Dump Bytes Command

Syntax

DB [*address*|*range*]

The Dump Bytes command (**DB**) displays the hexadecimal and ASCII values of the bytes at the specified *address* or in the specified *range* of addresses. The command displays one or more lines, depending on the address or range supplied.

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 dash (-). ASCII values are printed without separation. Unprintable ASCII values (lower than 20h or higher than 7Eh) are displayed as dots (.). 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
04BA:0100 41 20 73 74 72 69 6E 67-04 01 05 54 65 78 0D 0A  A string...Text..
04BA:0110 2E
-
```

The first example displays the byte values from `cs:100` to `110`. ASCII characters are shown on the right.

```
-DB
```

The second example displays 128 bytes starting 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. The dumped bytes are not shown in this example.

```
-DB buffer buffer+f
2145:0020                          -66 75 6E 63 74 69 6F 6E          function
2145:0030  0D 0A 20 20 20 20 20 20                           ..
-
```

The final example displays the first 16 (0Fh) bytes starting at the symbolic address `buffer`.

4.6.6.4 Dump Words Command

Syntax

DW [*address*|*range*]

The Dump Words command (**DW**) displays the hexadecimal values of the words (2-byte values) at *address* or in the specified *range* of addresses. The command displays one or more lines, depending on the address or range specified. 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
04BA:0100 2041 7473 6972 676E 0104 5405 7865 0A0D
04BA:0110 002E
-
```

The first example displays the word values from `cs:100` to `cs:110`. No more than eight values per line are displayed.

```
-DW
```

The second example displays 64 words starting 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. The dumped bytes are not shown in this example.

```
-DW buffer buffer+f
2145:0028  7566 636E 6974 6E6F 0A0D 2020 2020 2020
-
```

The final example displays the first eight words (0Fh bytes) starting at the symbolic address `buffer`.

4.6.6.5 Dump Doublewords Command

Syntax

DD [*address*|*range*]

The Dump Doublewords command (**DD**) displays the hexadecimal values of the doublewords (4-byte values) at *address* or in the specified *range* of addresses. The command displays one or more lines, depending on the address or range specified. Each line displays the address of the first doubleword in the line, followed by up to four hexadecimal doubleword values. The words of each doubleword are separated by a colon. The 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
04BA:0100 7473:2041 676E:6972 5405:0104 0A0D:7865
04BA:0110 0000:002E
-
```

The first example displays the doubleword values from `cs:100` to `cs:110`. No more than four doubleword values per line are displayed.

```
-DD
```

The second example displays 32 doublewords starting 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. The dumped bytes are not shown in this example.

```
-DD buffer buffer+f
2145:0028  636E:7566 6E6F:6974 2020:0A0D 2020:2020
-
```

The final example displays the first four doublewords (0Fh bytes) starting at the symbolic address `buffer`.

4.6.6.6 Dump Short-Reals Command

Syntax

DS ⟦*address*¦*range*⟧

The Dump Short-Reals command (**DS**) displays the hexadecimal and decimal values of the short (4-byte) floating-point numbers at *address* or in the specified *range* of addresses.

The command displays one or more lines, depending on the address or range specified. Each line displays the address of the floating-point number in the first column. Next, the hexadecimal values of the bytes in the number are shown, followed by the decimal value of the number. The hexadecimal values are separated by spaces.

The decimal value has the form:

+¦−**0.***decimaldigits*E+¦−*mantissa*

The sign of the number is followed by a **0** and a decimal point (**.**). Next come as many as 16 *decimaldigits* (although only 7 of these digits are significant). The decimal digits are followed by the letter **E**, which marks the start of the *mantissa*. Next comes the sign of the mantissa followed by the digits of the mantissa.

The command displays at least one value. If a *range* is specified, all values in the range are displayed.

Examples

```
-DS ds:100
04BA:0100 A3 68 21 A3   -0.8749985175576769E-17
-
```

The first example displays the short-real floating-point number at the address `ds:100`. Only one value per line is displayed.

```
-DS pi
210C:0140  DB OF 49 40  +0.3141592741012573E+1
-
```

The second example displays the short-real floating-point number at the symbolic address `pi`.

4.6.6.7 Dump Long-Reals Command

Syntax

DL [*address*|*range*]

The Dump Long-Reals command (**DL**) displays the hexadecimal and decimal values of the long (8-byte) floating-point numbers at the specified *address* or in the specified *range* of addresses.

The command displays one or more lines, depending on the address or range specified. Each line displays the address of the floating-point number in the first column. Next, the hexadecimal values of the bytes in the number are shown, followed by the decimal value of the number. The hexadecimal values are separated by spaces.

The decimal value has the form:

+|–**0.***decimaldigits***E**+|–*mantissa*

The sign of the number is followed by a **0** and a decimal point (**.**). Next come as many as 16 *decimaldigits*. The decimal digits are followed by the letter **E**, which marks the start of the *mantissa*. Next comes the sign of the mantissa, followed by the digits of the mantissa.

The command displays at least one value. If a *range* is specified, all values in the range are displayed.

Examples

```
-DL DS:100
04BA:0100 04 C6 06 10 1F 01 33 C0  -0.1900438022771233E+2
-
```

The first example displays the long-real floating-point number at the address `DS:100`. Only one value per line is displayed.

```
-DL pi
210C:0120  11 2D 44 54 FB 21 09 40  +0.314159265358979E+1
-
```

The second example displays the long-real floating-point number at the symbolic address `pi`.

4.6.6.8 Dump Ten-Byte Reals Command

Syntax

DT [[*address*|*range*]]

The Dump Ten-Byte Reals command (**DT**) displays the hexadecimal and decimal values of the 10-byte floating-point numbers at the specified *address* or in the specified *range* of addresses.

The command displays one or more lines, depending on the address or range specified. Each line displays the address of the floating-point number in the first column. Next, the hexadecimal values of the bytes in the number are shown, followed by the decimal value of the number. The hexadecimal values are separated by spaces.

The decimal value has the form:

+|–**0.***decimaldigits*E+|–*mantissa*

The sign of the number is followed by a **0** and a decimal point (**.**). Next come as many as 16 *decimaldigits*. The decimal digits are followed by the letter **E**, which marks the start of the *mantissa*. Next comes the sign of the mantissa followed by the digits of the mantissa.

The command displays at least one value. If a *range* is specified, all values in the range are displayed.

Examples

```
-DT DS:100
04BA:0100 66 21 A3 06 2B A3 04 2B A3 0E  +0.5145365070468582E-3804
-
```

The first example displays the 10-byte real floating-point number at the address `DS:100`. Only one number per line is displayed.

```
-DT pi
210C:0100  DE 87 68 21 A2 DA 0F C9 00 40  +0.314159265358979E+1
-
```

The second example displays the 10-byte floating-point number at the symbolic address `pi`.

4.6.7 Enter Commands

SYMDEB has several commands for entering data from the keyboard (or other input device) to memory. The enter commands are listed below:

Command	Command Name
E	Enter
EA	Enter ASCII
EB	Enter Bytes
EW	Enter Words
ED	Enter Doublewords
ES	Enter Short Reals
EL	Enter Long Reals
ET	Enter Ten-Byte Reals

The next sections discuss these commands in logical, rather than alphabetical, order.

4.6.7.1 Enter Command

Syntax

E *address* ⟦*list*⟧

The Enter command (**E**) enters one or more values into memory at *address.* The size of the value which may be entered depends on the most recently used Enter command. If no Enter command has been used, the Enter Bytes command (**EB**) is assumed.

If an error occurs while entering a value, the value remains unchanged. If you do not supply a *list* of values to be entered, **SYMDEB** prompts for a new value at *address* by displaying the address and its current value followed by a dot (.). You can then replace the value by typing the new value after the current value. The command ignores extra trailing digits or other characters.

To exit the Enter command, press the RETURN key. You can exit the command at any time.

The different variations of the Enter command are explained in the next seven sections.

4.6.7.2 Enter Bytes Command

Syntax

EB *address* ⟦*list*⟧

The Enter Bytes command (**EB**) enters one or more byte values into memory at *address*. If the optional *list* is specified, the command replaces the byte at the specified address and the bytes at each subsequent address until all values in the list have been used.

If you do not supply a *list*, **SYMDEB** prompts for a new value at *address* by displaying the 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.

- To replace the byte value, type the new value after the current value.
- 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.
- 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.
- To stop entering bytes and return to the **SYMDEB** prompt, press the RETURN key. You can exit the command at any time.

Examples

```
-EB CS:100 01 2B E5
-
```

The first example replaces the 3 bytes at CS:100, CS:101, and CS:102 with `01`, `2B`, and `E5`, respectively.

```
-EB CS:100
```

The second example causes **SYMDEB** to display the current value on the line following the command and wait for you to enter a new value. In the examples below an underscore represents the cursor:

```
-EB CS:100
2344:0100  F3._
```

You can then change the value F3 to the new value 5E by typing 5E as shown below

```
-EB CS:100
2344:0100  F3.5e_
```

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

```
-EB CS:100
2344:0100  F3.5e    10._
```

Then type the next value:

```
-EB CS:100
2344:0100  F3.5e    10.76_
```

Press the SPACE bar:

```
-EB CS:100
2344:0100  F3.5e    10.76    BO._
```

You could then return to the previous value to correct a mistake by typing a minus sign:

```
-EB CS:100
2344:0100  F3.5e    10.76    BO.-
2344:0100  76._
```

Type the correct value:

```
-EB CS:100
2344:0100  F3.5e    10.76    BO.-
2344:0100  76.77_
```

Press the RETURN key to stop entering bytes. After you press the RETURN key, the **SYMDEB** prompt reappears as shown below:

```
-EB CS:100
2344:0100  F3.5e    10.76    BO.-
2344:0100  76.77
-
```

4.6.7.3 Enter ASCII Command

Syntax

EA *address* ⟦*list*⟧

The Enter ASCII command (**EA**) works exactly the same as the Enter Bytes command (**EB**), described in the previous section.

Example

```
-EA data_seg:msg2 "Can't open file"
-
```

In the example above, the string `Can't open file` is entered starting at the symbolic address `data_seg:msg2`. You could use the Enter Bytes command to do the same thing, or you could enter nonstring values as shown in Section 4.6.7.2 using the Enter ASCII command.

4.6.7.4 Enter Words Command

Syntax

EW *address* ⟦*value*⟧

The Enter Words command (**EW**) enters a word value into memory. The optional *value* consists of a single word value.

If no *value* is specified, the command displays the word at *address* and prompts for a replacement. If a value is specified, the command replaces the word at the specified address, then displays the next word and prompts for a replacement.

The Enter Words command continues to display words and prompt for replacement values until you exit the command by pressing the RETURN key.

Example

```
-EW CS:400 4e3a
2344:0402  ED32.8ad8
2344:0404  1D3C.
-
```

In the example above, the word at CS:400 is replaced with 04E3A. **SYMDEB** displays the next word (ED32) and prompts for a replacement. The number 8AD8 is supplied as the next word, and the RETURN key is pressed to stop entering words.

4.6.7.5 Enter Doublewords Command

Syntax

ED *address* ⟦*value*⟧

The Enter Doublewords command (**ED**) enters a doubleword value into memory. The optional *value* consists of one doubleword value. Doublewords must be typed as two words separated by a colon (:).

If no value is specified, the command displays the doubleword at *address* and prompts for a replacement. If a value is specified, the command replaces the doubleword at the specified address, then displays the next doubleword and prompts for a replacement.

The Enter Doublewords command continues to display doublewords and prompt for replacement values until you exit the command by pressing the RETURN key.

Example

```
-ED CS:100 12EF:CD01
2344:0104  440E:1234.1234:5678
2344:0108  8ED9:1234.
-
```

In the example above, the doubleword at CS:100 is replaced with 12EF:CD01. **SYMDEB** displays the next doubleword (440E:1234) and prompts for a replacement. The number 1234:5678 is supplied as the next doubleword, and the RETURN key is pressed to stop entering doublewords.

4.6.7.6 Enter Short-Reals Command

Syntax

ES *address* ⟦*value*⟧

The Enter Short-Reals command (**ES**) enters a short-real value into memory. The optional *value* consists of one short-real value.

If no value is specified, the command displays the short-real value at the specified *address* and prompts for a replacement. If a value is specified, the command replaces the short-real value at the specified address, then displays the next short-real value and prompts for a replacement.

The Enter Short-Reals command continues to display short-real values and prompt for replacement values until you exit the command by pressing the RETURN key.

Example

```
-ES pi 3.1415926
-
```

The example above enters 3.1415926 at the symbolic address `pi`. The same number could also be entered as shown below:

```
-ES pi
210C:0130  -0.1256210825216E+16          +0.31415926e+1
210C:0134  -0.4309309980615894E-31
-
```

If you used the Dump Short-Reals command (**DS**) to examine the value just entered (as shown below), up to 16 digits would be displayed, but the last nine digits would not be significant:

```
-DS pi
210C:0130  DA OF 49 40  +0.3141592502593994E+1
-
```

4.6.7.7 Enter Long-Reals Command

Syntax

EL *address* ⟦*value*⟧

The Enter Long-Reals command (**EL**) enters a long-real value into memory. The optional *value* consists of one long-real value.

If no value is specified, the command displays the long-real value at the specified *address* and prompts for a replacement. If a value is specified, the command replaces the long-real value at the specified address, then displays the next long-real value and prompts for a replacement.

The Enter Long-Reals command continues to display long-real values and prompt for replacement values until you exit the command by pressing the RETURN key.

Example

```
-EL pi 3.141592653589793
-
```

The example above enters 3.141592653589793 at the symbolic address `pi`. The same number could also be entered as shown below:

```
-EL 170
210C:0170  +0.1343280735843091E+65299    +0.3141592653589793e+1
210C:0178  -0.1040230032441619E-71
-
```

4.6.7.8 Enter Ten-Byte Reals Command

Syntax

ET *address* [[*value*]]

The Enter Ten-Byte Reals command (**ET**) enters a 10-byte real value into memory. The optional *value* consists of a single 10-byte real value.

If no value is specified, the command displays the 10-byte real value at the specified *address* and prompts for a replacement. If a value is specified, the command replaces the 10-byte real value at the specified address, then displays the next 10-byte real value and prompts for a replacement.

The Enter Ten-Byte Reals command continues to display 10-byte real values and prompt for replacement values until you exit the command by pressing the RETURN key.

Example

```
-ET pi 3.141592653589793
-
```

The example above enters 3.141592653589793 at the symbolic address `pi`. The same number could also be entered as shown below:

```
-ET pi
210C:0150  +0.0204654128113587E+7898      +0.3141592653589793e+1
210C:015A  +0.5976239733286124E+3896
-
```

4.6.8 Examine Symbol Map Commands

Syntax

X [[*]]
X? [[*mapname*!]] [[*segmentname*:]] [[*symbolname*]]

The Examine Symbol Map commands (**X** or **X?**) display the names and addresses of the symbols in the current symbol maps. **SYMDEB** creates a symbol map for each symbol-file name specified in the **SYMDEB** command line. The Examine Symbol Map commands can then be used to examine the contents of the maps.

The **X** form of the Examine Symbol Map 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 currently loaded symbol maps.

The **X?** form of the Examine Symbol Map command displays the names and addresses of one or more symbols in the symbol map. If a *mapname!* is specified, the command displays information for that symbol map. The *mapname* must be the file name (without extension) of the corresponding symbol file. The file name must by followed by an exclamation point (!).

If *segmentname*: is specified, the command displays the name and load segment address for that segment. The segmentname must be the name of a segment named within the explicitly specified or currently open symbol map. The segmentname must be followed by a colon (:).

If a *symbolname* is specified, the command displays the segment address and segment offset for that symbol. The symbolname must be the name of a symbol in the specified segment.

To display information about more than one segment or symbol, enter a partial *segmentname* or *symbolname* ending with an asterisk (*). The asterisk acts as a wildcard character. **SYMDEB** displays information about all segments or symbols whose names start with the same characters with which *segmentname* or *symbolname* start. For example, `F*:` matches all segment names that start with `F`. Similarly, `_*` matches all symbols that start with an underscore (_).

In the examples, assume that **SYMDEB** was started with the following command line:

```
SYMDEB resident.sym count.sym count.exe
```

This command line instructs **SYMDEB** to load two symbol files and one executable file: `resident.sym`, `count.sym`, and `count.exe`. Only one symbol map can be open at a time, so **SYMDEB** opens the one whose name matches the name of the executable file (`count.sym`). If none of the symbol file names matched, **SYMDEB** would open the first symbol file in the command line.

Examples

```
-X
[2154 COUNT]
    2164 DATA
   [21E8 CODE]
-
```

The example above displays the name of the currently open symbol map and the names and load-segment addresses of the segments in that map. Brackets indicate that a symbol map or segment is open. An open segment will be searched first if you give a command that accesses a symbol. The example indicates that the segment `code` is open, so symbols in the code segment will be accessed slightly faster than symbols in the `data` segment.

```
-X*
 0000 COUNT
       0010 DATA
       01A3 CODE
[2154 Resident]
       2164 DATA
      [21E8 CODE]
-
```

In the second example above, all currently loaded symbol maps are displayed. Brackets indicate the open map and segment.

```
-X?resident!
 0000 RESIDENT
-
```

The third example displays the load-segment address of the symbol map file `resident`.

```
-X?resident!code:
CODE: (01A3)
-
```

The fourth example displays the start address of segment `code` in the map file `resident`.

```
-X?resident!data:c*
CODE: (01A3)
01E2 CYCLE      04D1 CLEAR
-
```

The fifth example displays the addresses of all symbols beginning with `c` in the `data` segment of symbol file `resident`.

```
-X?*
CODE: (21E8)
0016 GET_FILE  002C OPEN_FILE 0044 OK        0050 BUFF_READ 0069 DONE
0071 CONV_HEX  0075 ROTATE    008F QUIT      0095 WORD_C    00A4 NEXT_CHAR
00AA NEW_WORD  00AB OUT_WORD  00B6 IN_WORD
DATA: (2164)
0000 PROMPT    0011 NAMEBUF
0013 FNAME     0028 BUFFER    0828 ERR1      083C ERR2      0848 COUNT
-
```

The final example displays the addresses of all symbols in the currently open map file (`count`).

4.6.9 Fill Command

Syntax

F *range list*

The Fill command (**F**) fills the addresses in the specified *range* with the values specified in *list*. If the 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 the 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
-
```

The first example fills 255 (100h) bytes of memory starting at CS:100 with the value FFh.

```
-F DGROUP:table L 64  42 79 74 65 73
-
```

The second example fills the 100 (64h) bytes starting at `DGROUP:table` with the following byte values: 42h, 79h, 74h, 65h, and 73h. These five values are repeated until all 100 bytes are filled.

4.6.10 Go Command

Syntax

G [[=*startaddress*]] [[*breakaddresses*]]

The Go command (**G**) passes execution control to the program at the optional *startaddress*. Execution continues to the end of the program or until the optional *breakaddress* is encountered. The program also stops at any breakpoints set using the Breakpoint Set command (**BP**).

If no *startaddress* is specified, the command passes execution control to the address specified by the current values of the **CS** and **IP** registers. The equal sign (=) indicates that the value is a start address. Any values specified without the equal sign are assumed to be break addresses.

If a break address is specified, it must specify an instruction address (that is, the address must contain the first byte of an instruction opcode). Up to 10 addresses can be specified at one time. The addresses can be specified in any order. If you attempt to set more than 10 breakpoints, **SYMDEB** displays an error message. Only the first address encountered during execution will cause a break. All others are ignored. If you want execution to stop at more than one breakaddress, use the Breakpoint Set command.

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 (**R**).

Notes

The Go command (**G**) 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 at least 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** 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 command (**N**) and Load command (**L**) before attempting to run the program again.

SYMDEB displays the message `Program terminated normally` whenever execution reaches the program end. **SYMDEB** stops execution and displays the current values of registers and flags.

Examples

```
-G =_main  _add
```

In the first example, **SYMDEB** starts program execution at the instruction named by the symbolic address `_main`. Execution continues until the address `_add` is reached (or until the end of the program if `_add` is not encountered).

```
-G
```

The second example passes control to the instruction pointed to by the current values of the **CS** and **IP** registers. **SYMDEB** will continue execution until it reaches either the end of the program or a breakpoint defined with the Breakpoint Set command (**BP**).

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

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

4.6.11 Help Command

Syntax

?

The Help command (?) displays a list of all **SYMDEB** commands and operators with the syntax for each.

4.6.12 Hex Command

Syntax

H *value1 value2*

The Hex command (**H**) 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, use the Display command (**D**) (see Section 4.6.5).

Examples

```
-H 3 4
0007  FFFF
-
```

The first example displays the results of 3 + 4 (7) and 3 – 4 (FFFF).

```
-H afd 2ec
0DE9  0811
-
```

The second example displays the results of 0AFD + 02EC (0DE9) and 0AFD - 02EC (0811).

4.6.13 Input Command

Syntax

I *port*

The Input command (**I**) reads and displays a byte from the specified *port*. The input port can be any 16-bit port address.

Example

```
-I 2F8
E8
-
```

The preceding example reads input port number 2F8 and displays the result (E8h).

4.6.14 Load Command

Syntax

L ⟦*address* ⟦*drive record count*⟧⟧

The Load command (**L**) copies the contents of a named file or the contents of a specified number of logical disk records into memory. The contents are copied to the specified *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 file name must be supplied before the Load command can be used. You can give a name by using the Name command (**N**) (Section 4.6.16), or by passing it as a program argument when you start **SYMDEB** (Section 4.3.3). 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 of the default file control block that receives any file name specified with the Name command or any file name passed as a program argument.

If an *address* is specified, the command places the contents of the file or sectors at the memory locations starting at the specified address. Otherwise, it places the contents at the address specified 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 specified. 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), or 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 be read from the disk. It can be any 1- to 4-digit hexadecimal number.

Notes

If the named file has an **.EXE** extension, the Load command (**L**) adjusts the load address to the address specified in the **.EXE** file header. This means that the *address* parameter is always ignored for **.EXE** files.

Since the Load command strips any header information from an **.EXE** file before loading, the number of bytes actually loaded will differ from 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 specified, the file is loaded at its start address.

Examples

```
-N file.exe
-L
-
```

The first example loads the file named `file.exe` into memory at the address CS:100. The number of bytes loaded (the length of `file.exe` minus its program header) is copied to the **BX:CX** register pair.

```
-L DGROUP:table
-
```

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

```
-L workspace 2 34 3
-
```

The final example loads three logical records from Drive C (02), beginning with logical record number 34h, into memory at the symbolic address `workspace`.

4.6.15 Move Command

Syntax

M *range address*

The Move command (M) 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 when the source and destination blocks overlap. 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, the Move command copies data starting at the source block's lowest address whenever the source is at a higher address than the destination. If the source is at a lower address, the Move command copies data beginning at the source's highest address.

Examples

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

The first 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
-
```

The second example copies the 256 (100h) bytes at the symbolic address `DS:table` to the symbolic address `workspace`.

4.6.16 Name Command

Syntax

N ⟦*filename*⟧ ⟦*arguments*⟧

The Name command (**N**) sets the file name for a subsequent Load command (**L**) or Write command (**W**), or sets program arguments for subsequent execution of a loaded program.

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

If *arguments* are specified, 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 file names, the command creates file control blocks (FCBs) 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.

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

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

Address	**Contents**
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
-D 80 8f
2BB2:0080  0A 20 66 69 6C 65 31 2E-65 78 65 0D 20 63 3A 43   . file1.exe. c:C
-
```

The first example sets the file name `file1.exe` for use by subsequent Load and Write commands. The Dump command (**D**) is entered to show the result. The Name command copies the length of the name (0Ah or 10 decimal including the initial space) to byte 80 of the data segment and copies the file name to the bytes starting at 81.

```
-N file1.dat file2.dat /m /b
-D 50 9f
2BB2:0050  CD 21 CB 00 00 00 00 00-00 00 00 00 00 46 49 4C   M!K.........FIL
2BB2:0060  45 33 20 20 20 44 41 54-00 00 00 00 00 20 20 20   E1   DAT.....
2BB2:0070  20 20 20 20 20 20 20 20-00 00 00 00 00 00 00 00           ........
2BB2:0080  1A 20 66 69 6C 65 32 2E-64 61 74 20 66 69 6C 65   . file1.dat file
2BB2:0090  33 2E 64 61 74 20 2F 6D-20 2F 62 0D 4E 54 2E 65   2.dat /m /b.NT.e
-
```

The second example sets the program arguments for the program being debugged. The Dump command has been entered to show the results. The Name command creates a File Control Block (FCB) for file `file2.dat` at DS:5C. It also copies the entire command line (except the command letter N), to memory starting at DS:81. The characters following the last letter of the command line are simply data left over from previous commands.

4.6.17 Open Map Command

Syntax

XO ⟦*mapname*!⟧ ⟦*segmentname*⟧

The Open Map command (**XO**) sets the active symbol map and/or segment. If *mapname* is specified, the command sets the active symbol map to the specified map. The *mapname* must be the file name (without extension) of one of the symbol files specified in the **SYMDEB** command line. If *segmentname* is specified, the command sets the active segment to the named segment. The *segmentname* must be the name of a segment in the specified symbol map. All segments in an open map are accessible, but the open segment is searched first. A map file can be opened only if it was loaded by providing its name in the **SYMDEB** command line.

The examples below assume that **SYMDEB** was started with the following command line. The Examine Symbol-Map command is also entered to show the initial status:

```
SYMDEB resident.sym count.sym count.exe
-X*
 0000 RESIDENT
       0010 DATA
       01A3 CODE
[2154 COUNT]
       2164 DATA
      [21E8 CODE]
```

Examples

```
-XO resident!
-X*
[0000 RESIDENT]
      [0010 DATA]
       01A3 CODE
 2154 COUNT
       2164 DATA
       21E8 CODE
-
```

The first example opens the symbol map `resident`.

```
-XO count!data
-X*
 0000 RESIDENT
       0010 DATA
       01A3 CODE
[2154 COUNT]
      [2164 DATA]
       21E8 CODE
-
```

The second example opens the segment `data` in the symbol map `count`.

```
-XO code
-X*
 0000 RESIDENT
       0010 DATA
       01A3 CODE
[2154 COUNT]
       2164 DATA
      [21E8 CODE]
-
```

The final example activates the segment `code` in the current symbol map (`count`).

4.6.18 Output Command

Syntax

O *port byte*

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

Examples

```
-O 2f8    4f
-
```

The first example sends the byte value 4Fh to output port 2F8h.

```
-O  3  21
-
```

The second example sends the byte value 21h to output port 3.

4.6.19 PTrace Command

Syntax

P ⟦=*startaddress*⟧ ⟦*count*⟧

The PTrace command (**P**) executes the instruction at the specified *startaddress*, then displays the current values of all registers and flags. The display has the same format as the Register command (**R**) (see Section 4.6.22).

If the optional *startaddress* is specified, the command starts execution at the specified address. Otherwise, it starts execution at the instruction pointed to by the current **CS** and **IP** registers. The equal sign (=) is necessary to indicate a *startaddress.* If a number is specified without an equal sign, **SYMDEB** assumes that the number is a *count.*

If the optional *count* is specified, the command executes *count* number of instructions before stopping. The command displays the current values of the registers and flags for each instruction before executing the next.

In source-only mode (S+), PTrace operates directly on source lines. In this mode, PTrace steps over function or procedure calls. The source-only mode is only available for programs developed with high-level-language compilers. See Section 4.6.25 for more information about setting the source mode.

Note

The PTrace command is identical to the Trace command (**T**), except that it automatically executes and returns from any calls or software interrupts it encounters, leaving execution control at the next instruction after the called routine. The Trace command always stops after executing the call or interrupt, leaving execution control inside the called routine. One exception to this rule is that neither the Trace nor the PTrace command enters interrupt 21h, the MS-DOS function request interrupt.

Examples

```
-P =work
AX=0800  BX=0005  CX=0800  DX=002E  SP=00FE  BP=0000  SI=0017  DI=0000
DS=2BED  ES=2BD2  SS=2C72  CS=2BE2  IP=008C   NV UP EI PL NZ NA PE NC
2BE2:008C BE2E00          MOV    SI,002E
-
```

The first example executes the instruction at `work`, then displays the current values of the registers and flags, and the next instruction to be executed.

```
-T
AX=0800  BX=0005  CX=0800  DX=002E  SP=0100  BP=0000  SI=0017  DI=0000
DS=2BED  ES=2BD2  SS=2C72  CS=2BE2  IP=004D   NV UP EI PL NZ NA PE NC
2BE2:004D E83B00          CALL   WORD_C
-P
AX=0800  BX=0005  CX=0378  DX=002E  SP=0100  BP=0000  SI=084E  DI=0000
DS=2BED  ES=2BD2  SS=2C72  CS=2BE2  IP=0050   NV UP EI PL NZ NA PO NC
2BE2:0050 EBED            JMP    OK+05 (003F)
-
```

In the second example, the first instruction is executed with the Trace command, but the second is executed with the PTrace command so that the **CALL** instruction will be executed instead of traced.

4.6.20 Quit Command

Syntax

Q

The Quit command (**Q**) terminates **SYMDEB** execution and returns control to MS-DOS.

Example

```
-Q
```

This example terminates **SYMDEB**.

4.6.21 Redirection Commands

Syntax

< *devicename*
> *devicename*
= *devicename*
{ *devicename*
} *devicename*
~ *devicename*

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

The { command reads all input for the debugged program from the specified device. The } command writes all output from the debugged program to the specified device. The ~ command causes the debugged program to both read from, and write to, the specified device.

The *devicename* can be any MS-DOS device or file name. If **COM1** or **COM2** is specified, the port's baud rate and other modes must be properly set for the attached terminal. If redirection does not appear to work correctly, check your MS-DOS manual and hardware manuals to make sure the lines are set up correctly.

The Redirection commands are typically used to debug programs that require full use of the console screen. For example, you might redirect output from a graphics program to a color graphics monitor while reading the **SYMDEB** output on a monochrome monitor.

Note

If input is redirected to **COM1** or **COM2**, the CONTROL-S and CONTROL-C keystroke combinations are unavailable and will be ignored. Make sure the device you specify is available before using a redirection command.

Examples

```
->COM1
```

The first example redirects **SYMDEB** command output to the **COM1** device.

```
-=COM1
```

The second example redirects command input from, and output to, **COM1**.

```
->outfile.txt
```

The third example redirects command output to the file `outfile.txt`. The cursor disappears. Any keystrokes you type will not be echoed to the screen, but they will be sent to the file. Make sure you know exactly what commands you want to send to the file before you begin. To close the file, enter the command `>CON` or `Q`.

```
-<infile.txt
```

The final example redirects command input from file `infile.txt` to **SYMDEB**. If the file contains a series of **SYMDEB** commands (separated by carriage returns), **SYMDEB** will execute the commands to the end of the file. The last command in the file should be either `Q` or `<CON`. If you fail to place one of these commands at the end of the file, you will have to do a warm boot since there will be no way to tell **SYMDEB** to end the session.

4.6.22 Register Command

Syntax

R ⟦*registername*⟦⟦=⟧*value*⟧⟧

The Register command (**R**) displays the contents of the central processing unit (CPU) registers and allows the contents to be changed to new values.

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

The register display shows the next statement to be executed and attempts to evaluate it, if that is appropriate. If an operand of the instruction contains memory expressions or immediate data, **SYMDEB** will evaluate operands. If the instruction is an MS-DOS call, the function number will be shown. If the **CS** and **IP** registers are currently at a breakpoint or a memory location, the register display will indicate the symbol or breakpoint. Examples are shown at the end of this section.

The Trace command (**T**) and PTrace command (**P**) show registers in the same format as the Register command.

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

The register name can be any of the following names: **AX**, **BX**, **CX**, **DX**, **CS**, **DS**, **SS**, **ES**, **SP**, **BP**, **SI**, **DI**, **IP**, **PC**, or **F**.

IP and **PC** name the same register: the instruction pointer. **F** is a special name for the flags register. The other registers are discussed in Section 5.2.5 of the *Microsoft Macro Assembler Reference Manual.*

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 prompt consisting of a colon. 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 type an illegal register name, **SYMDEB** 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 flag values are shown below:

Table 4.5

Flag Values

FLAG	SET	CLEAR
Overflow	OV	NV
Direction	DN (decrement)	UP (increment)
Interrupt	EI (enabled)	DI (disabled)
Sign	NG (negative)	PL (positive)
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 dash (–). Enter new values after the dash 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, simply press the RETURN key.

If more than one value is entered for a flag, a `Double flag!` message will be displayed. If you enter names 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

The first example displays all register and flag values, as well as the instruction at the address pointed to by the **CS** and **IP** registers. In **S+** or **S&** mode, the display might look like this:

```
-R
AX=0008  BX=0A68  CX=0034  DX=0000  SP=0A64  BP=0A70  SI=00E6  DI=0A7A
DS=151B  ES=151B  SS=151B  CS=151B  IP=0036  NV UP EI PL NZ NA PE NC
8:       a = add(f, g);
11BC:0036 83EC08        SUB     SP,+08                          ;BR2
-
```

Notice the comment at the right of the last line showing that the current address is at breakpoint 2.

In **S–** mode, the display might look like this:

```
-R
AX=4A00  BX=4500  CX=0000  DX=CD00  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=2382  ES=2382  SS=2382  CS=2382  IP=0104   NV UP EI PL NZ NA PO NC
2382:0104 CD21          INT     21  ;Modify Allocated Memory
-
```

The instruction is shown last. Notice the comment indicating the MS-DOS function number about to be executed. The function number is taken from the **AH** register.

```
-R
AX=4A00  BX=4500  CX=0000  DX=CD00  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=2382  ES=2382  SS=2382  CS=2382  IP=0100   NV UP EI PL NZ NA PO NC
CODE:START:
2382:0100 B745          MOV     BH,45                            ;'E'
-
```

In the second example immediately above, notice the words CODE:START: indicating that the next instruction is at the symbol START in the CODE segment. The ;'E' to the right of the instruction indicates that 45 evaluates to the ASCII code for E. This may not always be relevant to the purpose of the instruction, but often it is useful.

```
-R
AX=4A00  BX=4500  CX=0000  DX=CD00  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=2382  ES=2382  SS=2382  CS=2382  IP=0102   NV UP EI PL NZ NA PO NC
2382:0102 8A34          MOV     DH,[SI]                               DS:0000=CD
-
```

In the third example immediately above, the memory operand [SI] in the instruction is evaluated on the right side of the screen as DS:0000=CD. This means that the byte pointed to by **SI** is at offset 0 in the **DS** segment, and that it contains the value CDh.

```
-RIP 100
-
```

The fourth example changes the **IP** register to the value 100h (256 decimal).

```
-R AX
```

The fifth example displays the current value of the **AX** register and prompts for a new value. The display will look like this (the underscore represents the **SYMDEB** cursor):

```
-R AX
AX 0E00
:_
```

You can now type any 16-bit value after the colon (:). For example, to change the **AX** value to 100h, enter `100` as shown below:

```
-R AX
AX 0E00
:100
-
```

You could also press the RETURN key if you decided not to change the register value.

```
-R F
```

The final example displays the current flag values and prompts for changes. The display should look like this (the underscore represents the **SYMDEB** cursor):

```
-R F
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. For example, to set the carry flag, enter `CY` as shown below:

```
-R F
NV UP DI NG NZ AC PE NC -CY
-
```

4.6.23 Screen Swap Command

Syntax

\

The Screen Swap command (\) allows you to switch from the debugging screen to the program screen. This command is convenient for programs that update the screen frequently, or for graphics programs in which the program output cannot be shown on the **SYMDEB** screen. After you enter a backslash (\), the program screen immediately replaces the **SYMDEB** screen. After you inspect the current status of the program screen, you can press any key to return to the **SYMDEB** screen.

This command is only available if you use the **/S** option when starting **SYMDEB** and your computer is an IBM Personal Computer or a close compatible. If your computer is an IBM compatible, you must also use the **/I** option.

4.6.24 Search Command

Syntax

S *range list*

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

The *list* can have any number of bytes. Each byte value must be separated from the others by a space or comma (,). If the list contains more than one byte, the Search command 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 buffer 1 1500 "error"
2BBA:040A
2BBA:05E3
2BBA:0604
-
```

The first example displays the address of each memory location containing the string `error`. The command searches the first 1500h bytes at the address specified by `buffer`. The string was found at the three addresses shown by **SYMDEB**.

```
-S DS:100 200 0A
3CBA:0132
3CBA:01C2
-
```

The second example displays the address of each memory location in the range DS:100 to DS:200 containing the byte value 0Ah. The value was found at the two addresses shown by **SYMDEB**.

4.6.25 Set Source Mode Command

Syntax

S–¦&¦+

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

Initially, **SYMDEB** displays intermingled source lines and disassembled code (the **S&** setting).

The Set Source Mode command is only meaningful if you are debugging executable files produced with high-level-language compilers. Since **MASM** cannot send line numbers to the object file, you cannot create a map file that **SYMDEB** can use to relate assembler instructions to source-code lines. All three source modes work as if the setting were **S–** when you debug programs created with **MASM** or an incompatible compiler.

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 specified, **SYMDEB** displays source lines only when the current instruction address specified by **CS:IP** matches a line number. The Set Source Mode command affects instructions displayed by the Unassemble command (**U**) (see Section 4.6.2). When the source mode is set to **S–**, the Unassemble command displays only disassembled instruction code. When the source mode is **S+** or **S&**, the Unassemble command intermingles disassembled instructions with program source lines.

The Set Source Mode command also affects the Register (**R**), Trace (**T**), and PTrace (**P**) commands. In **S+** mode, these commands process one source line at a time (which may correspond to more than one line of disassembled instructions). In **S–** mode disassembled instructions are shown, but not source lines. In **S&** mode disassembled instructions and line numbers are shown.

Source lines have the form:

linenumber:source

Source lines are always displayed before any disassembled instructions. 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

Whenever **SYMDEB** must access a source file for the first time, it searches the current working directory for a source file with the same base name as the symbol file. If the source file is not found, **SYMDEB** displays the following prompt:

`Source file name for` *mapname* `(cr for none)?`

Note that *mapname* is the file name of the symbol file. To display source lines, you must type the name of the corresponding source file. The file name must include the file-name 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 RETURN key when **SYMDEB** prompts for the file name. **SYMDEB** will suppress the actual source lines and display a map name and line number instead.

One case in which you must suppress display of source lines is with early versions of Pascal and FORTRAN (prior to 3.31). The run-time object files of these compilers contain line-number information. When **SYMDEB** tries to access these lines, it will prompt you for the source-file name. Press the RETURN key to ignore this request, since you will not have access to the run-time source files.

Examples

```
-S+
-
```

The first example sets **SYMDEB** to source-line display mode.

```
-S&
-
```

The second example sets **SYMDEB** to combined source-line and disassembly display mode. On subsequent commands, **SYMDEB** displays both the source line and disassembled instruction code.

4.6.26 Shell Escape Command

Syntax

! [[*command*]]

The Shell Escape command (**!**) allows you to execute **COMMAND.COM** and MS-DOS commands from within **SYMDEB**. The Shell command by itself executes **COMMAND.COM** with no arguments, saving the current debugging context. After you are finished executing DOS commands, type the MS-DOS command **EXIT** and you will return to **SYMDEB** at the point where you left off.

In addition, you can type an MS-DOS command or an executable program file name directly after the Shell Escape command. The command will execute automatically, and, when it is completed, return control to **SYMDEB**.

Note

In order to use the Shell Escape command, the executable file being debugged must release the memory it does not need. A program can do this by using MS-DOS function call 4Ah (Modify Allocate Memory). This gives MS-DOS space to load the new **COMMAND.COM**. The same thing can be accomplished by linking with the **/CPARMAXALLOC** option.

Programs developed with Version 3.0 or later of Microsoft C do this automatically if they have been executed up to function `_main`. Programs developed with Version 3.30 or later of Microsoft Pascal or Microsoft FORTRAN also release memory if they have been executed up to the first procedure. **SYMDEB**, when loaded by itself, also frees memory. However, programs developed with **MASM** or an incompatible compiler must contain code to adjust memory if the Shell Escape command is to be used.

SYMDEB will print the message `Not enough memory` if memory has not been released.

The **SYMDEB** statement connector (;) cannot be used after the Shell Escape command, since all text encountered after the command is passed to **COMMAND.COM** will be interpreted as an MS-DOS command line. **SYMDEB** uses the **COMSPEC** environment variable to locate a copy of **COMMAND.COM**.

Examples

```
-!dir b:*.asm
```

In the first example, the MS-DOS internal command `dir` is executed, its output is shown on the screen, and control is returned to **SYMDEB**.

```
-!chkdsk b:
```

In the second example, the MS-DOS external command `chkdsk` is executed, the status of the disk in Drive B is displayed, and control is returned to **SYMDEB**. The file name specified could be for any executable file, not just for MS-DOS external programs.

4.6.27 Source Line Command

Syntax

.

A single period (.) displays the current source code line. This command works regardless of the current source mode. The command has no effect if you are debugging a program created with **MASM** or an incompatible compiler.

Example

```
-.
for (i = 0; i <= SIZE; i++);
-
```

The example above shows the current source line of the current source file (from a C program, in this case).

4.6.28 Stack Trace Command

Syntax

K ⟦*number*⟧

The Stack Trace command (**K**) allows you to display the current stack frame. The first line of the display shows the name of the current procedure, arguments to the procedure, and the file name and line number of the call to the procedure. The succeeding lines (if any) trace the call. For example, the next line displays the name of, and arguments to, the procedure that called the current procedure, and so on.

SYMDEB only displays the arguments to a procedure if it is able to determine the number of arguments. By specifying the optional *number*, you can force **SYMDEB** to display *number* words of arguments. For example, if the number of arguments to a procedure varies and **SYMDEB** cannot determine the exact number of actual arguments, no arguments will be displayed unless you give some value as the *number* argument.

Note

The Stack Trace command only works on procedures that follow the calling conventions used by Microsoft high-level languages. If a program produced with **MASM** does not follow these conventions, the command will be ignored. An example of a procedure call that follows these conventions is shown in Section 3.10 of the *Microsoft Macro Assembler Reference Manual.* The procedure shown in Section 5.2.9 of the same manual does not follow the conventions and would not work with the Stack Trace command.

Example

```
-K
IGROUP:_fact(0003) from .fact.c:12
IGROUP:_fact(0004) from .fact.c:12
IGROUP:_fact(0005) from .fact.c:12
IGROUP:_fact(0006) from .fact.c:3
IGROUP:_main(?)
-
```

In the example above, the first line of output indicates that the current procedure `_fact` (actually a function, since the example is in C), has one argument with a current value of 3. The procedure was called from line 12 of source file `fact.c`. The other output lines indicate that `_fact` is recursive and has called itself three times. The procedure was originally called from line 3 of the source file.

The procedure `_main` was also called, but **SYMDEB** could not determine how many arguments it had. You can force **SYMDEB** to give you the value for the first argument of `_main`, as shown below:

```
-K 1
IGROUP:_fact(0003) from .fact.c:12
IGROUP:_fact(0004) from .fact.c:12
IGROUP:_fact(0005) from .fact.c:12
IGROUP:_fact(0006) from .fact.c:3
IGROUP:_main(0001)
-
```

The last output line now indicates that the first argument of `_main` has a value of 1. This information may not always be relevant, depending on nature of the code being examined.

4.6.29 Symbol Set Command

Syntax

Z *symbol value*

The Symbol Set command (**Z**) sets the address of the specified symbol to the specified value.

Note

One specific situation in which you must set a symbol to a specific value is with old versions of FORTRAN and Pascal (Microsoft versions prior to 3.3 or IBM versions prior to 2.0). After starting **SYMDEB** and going to the first procedure of the program, use the Symbol Set command to set the address of **DGROUP** to the current value of the **DS** register. This enables you to access symbolic variable names within **DGROUP**. The correct address is set automatically with later versions of FORTRAN and Pascal.

Examples

```
-Z close 4C
-
```

The first example sets the address of the symbol `close` to the value 4Ch.

```
SYMDEB fortprog.sym fortprog.exe
-G main
-Z DGROUP DS
-
```

The second example starts **SYMDEB** with an early-version FORTRAN program, goes to the first procedure (`main`), and sets the value of the variable DGROUP to the current value of the **DS** register. You could do the same with early versions of Pascal, except that the first procedure would be the procedure having the program name. After this sequence of commands, symbols in DGROUP will have the correct addresses and can be accessed normally.

4.6.30 Trace Command

Syntax

T ⟦=*startaddress*⟧ ⟦*count*⟧

The Trace command (**T**) executes the instruction at *startaddress*, then displays the current values of all registers and flags. The display has the same format as the Register command (**R**).

If the optional *startaddress* is specified, the command starts execution at the specified address. Otherwise, it starts execution at the instruction pointed to by the current **CS** and **IP** registers. The equal sign (=) indicates a *startaddress*. If a number is specified without an equal sign, **SYMDEB** assumes the number is a *count*.

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

Use the Trace command if you want to trace through calls and interrupts. If you want to execute interrupts or calls without tracing through them, you should use the PTrace command (**P**) instead. Both commands execute DOS function calls (interrupt 21h) without tracing through them.

In source-only mode (S+), the Trace command operates directly on source lines. In this mode, the Trace command executes function or procedure calls while the PTrace command steps over them. This applies only to programs developed with high-level languages. Tracing through source lines works better if you turn off optimization when you compile the program (see Section 4.2.2).

Notes

The Trace command uses the hardware trace mode of the 8086, 8088, 80186, or 80286 microprocessor. Consequently, you may also trace instructions stored in ROM (read-only memory).

Examples

```
-T 2
AX=0924  BX=0000  CX=0900  DX=0017  SP=0100  BP=0000  SI=0000  DI=0000
DS=39E7  ES=39CC  SS=3A6C  CS=39DC  IP=000F   NV UP EI NG NZ AC PE CY
39DC:000F B40A          MOV    AH,0A
AX=0A24  BX=0000  CX=0900  DX=0017  SP=0100  BP=0000  SI=0000  DI=0000
DS=39E7  ES=39CC  SS=3A6C  CS=39DC  IP=0011   NV UP EI NG NZ AC PE CY
39DC:0011 CD21          INT    21  ;Buffered Keyboard Input
-
```

The first example executes the next two executable source lines, and displays them.

```
-T _open
AX=0A24  BX=0000  CX=0900  DX=0019  SP=0100  BP=0000  SI=0000  DI=0000
DS=39E7  ES=39CC  SS=3A6C  CS=39DC  IP=0025   NV UP EI NG NZ AC PE CY
39DC:0025 32C0          XOR    AL,AL
-
```

The second example executes the instruction at _open, then displays the current values of the registers and flags. It also displays the next instruction to be executed. If you are in source-only mode (**S+**), this example executes the instruction at _open, then displays the next source line.

```
-T
AX=0A00  BX=0000  CX=0900  DX=0019  SP=0100  BP=0000  SI=0000  DI=0000
DS=39E7  ES=39CC  SS=3A6C  CS=39DC  IP=0027   NV UP EI PL ZR NA PE NC
39DC:0027 B43D          MOV    AH,3D                        ;'='
-
```

The third example executes the instruction pointed to by the current **CS** and **IP** register values.

```
-T =013
AX=0A00  BX=0000  CX=0900  DX=0019  SP=0100  BP=0000  SI=0019  DI=0000
DS=39E7  ES=39CC  SS=3A6C  CS=39DC  IP=0015   NV UP EI PL ZR NA PE NC
39DC:0015 8A5C01        MOV    BL,[SI+01]                        DS:001A=00
-
```

The fourth example executes the instruction at 013h in the current **CS** segment.

```
-S+
-T 7
3:              printf("%d0,fact(6));
7:              int i;
9:              if (i == 1)
12:                     return(i * fact(i-1));
7:              int i;
9:              if (i == 1)
12:                     return(i * fact(i-1));
-
```

The final example sets the source-line mode to source only and traces through seven source lines. In source-only mode, no registers are shown, only source lines.

4.6.31 Unassemble Command

Syntax

U [[*range*]]

The Unassemble command (**U**) 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 (**S**), and on whether the program was developed with a high-level language. The different display modes all work as if the source-mode setting was **S-** when you debug programs developed with **MASM** or an incompatible compiler.

When you use the either the **S+** or **S&** mode on programs with a compatible compiler, **SYMDEB** displays source lines mixed with disassembled instructions. One source line is shown for each corresponding group of assembly-language statements. Source lines are read from the source file. Assembly-language statements are translated from memory bytes. The **S+** and **S&** modes work the same with the Unassemble command (they are different for the Trace command (**T**) and the PTrace command (**P**).

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 be in the map. If no line-number information exists for a specified portion of a program, **SYMDEB** will not display source text.

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

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 as a comment (following a semicolon) on the same display line.

80286 protected-mode mnemonics cannot be displayed.

Examples

```
-S+
-U .19
19:           i := 1;
2492:00CC B81300          MOV    AX,0013
2492:00CF 50              PUSH   AX
2492:00D0 9A82001126      CALL   DEBEQQ_CODE:LNTEQQ
2492:00D5 C7066A000100    MOV    Word Ptr [006A],0001
20:           notprime := false;
2492:00DB B81400          MOV    AX,0014
2492:00DE 50              PUSH   AX
-
```

The first example displays line 19 in the source code, followed by the disassembled instruction code for the statement at line 19 and part of the instructions for line 20. The source code in this example is in Pascal.

```
-S&
-U .18 L 10
18:       103 CONTINUE
294E:007C A1B200          MOV    AX, [00B2]
294E:007F 40              INC    AX
294E:0080 A3B200          MOV    [00B2],AX
294E:0083 3D0A00          CMP    AX,000A
294E:0086 7EA5            JLE    MAIN+2C (002D)
19:           CALL BUBBLE(R,10)
294E:0088 B86000          MOV    AX,0060
294E:008B 1E              PUSH   DS
294E:008C 50              PUSH   AX
294E:008D B88C0B          MOV    AX,0B8C
294E:0090 1E              PUSH   DS
294E:0091 50              PUSH   AX
294E:0092 9A35014E29      CALL   MAIN_:BUBBLE
20:           WRITE (*,002)
294E:0097 33C0            XOR    AX,AX
-
```

The second example displays 10 lines of disassembled instruction code and program-source lines beginning at the address line 18. The source code is in FORTRAN in this example.

```
-U CS:02AD
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
-
```

The third example displays eight lines of disassembled instruction code and program source code beginning at CS:02AD. Eight lines is the default if no *range* is specified. The source code is in C in this example.

```
-U conv_hex
CODE:CONV_HEX:
29D2:0071 B104          MOV    CL,04
29D2:0073 B504          MOV    CH,04
CODE:ROTATE:
29D2:0075 D3C3          ROL    BX,CL
29D2:0077 8AD3          MOV    DL,BL
29D2:0079 80E20F        AND    DL,0F
29D2:007C 80C230        ADD    DL,30                       ;'0'
-U
29D2:007F 80FA3A        CMP    DL,3A                       ;':'
29D2:0082 7C03          JL     ROTATE+12 (0087)
29D2:0084 80C207        ADD    DL,07
29D2:0087 B402          MOV    AH,02
29D2:0089 CD21          INT    21
29D2:008B FECD          DEC    CH
29D2:008D 75E6          JNZ    ROTATE
CODE:QUIT:
-
```

The fourth example shows the effect of the Unassemble command when **SYMDEB** is used on a sample program produced by **MASM**. The command disassembles eight lines of code beginning at the symbolic address conv_hex, then unassembles the next eight lines. No source-mode command is entered since the display will be the same regardless of the current mode.

```
-S-
-U _main L 0A
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]
-
```

The final example displays 10 (0Ah) lines of disassembled code starting at the address `_main`. The program in this example is written in C, but since no source lines are shown, the format of the symbols is the only indication of the source.

4.6.32 View Command

Syntax

V *address*

The View command (**V**) displays source lines beginning at the specified address. The symbol file must contain line-number information for source lines to be displayed. This means that the View command has no effect on programs developed with **MASM** or an incompatible compiler.

With compatible compilers, this command always shows source lines, regardless of the current source mode (**S–**, **S&**, or **S+**).

Example

```
-V _func
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++)
-
```

The example above displays eight source lines beginning at the address specified by `_func`. The example shows C code, but FORTRAN or Pascal code would be displayed in the same way.

4.6.33 Write Command

Syntax

W ⟦*address* ⟦*drive record count*⟧⟧

The Write command (W) writes the contents of a specified memory location to a named file, or to a specified logical record on disk.

To write to a file, the file name must be previously set with a Name command (N), and the **BX:CX** register pair must be set to the number of bytes to be written. If no *address* is specified, the command copies bytes starting from the address CS:100, where CS is the current value of the **CS** register. If *address* is specified, the command copies bytes starting at that address.

To write to a logical record on disk, the *address*, *drive*, *record*, and *count* must be specified. The *drive* must name the drive to be written to. It can be any number in the range 0 to 3, representing Drive A (0), B (1), C (2), or D (3). The *record* specifies 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 be written 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.

If the file you are debugging is a **.COM** or **.BIN** file, you can make changes to the program with **SYMDEB** and then write the program to a file. When you load the file, the file length, starting address, and file name will be set correctly for writing. However, if you use the Go (**G**), Ptrace (**P**), or Trace (**T**) commands during debugging, or if you change the **BX:CX** register values, you must reset each of these conditions before writing the file to disk.

You cannot use the Write command to write **.EXE** or **.HEX** files to disk. However, it is possible to modify these files with **SYMDEB**. The steps are outlined below. This is an advanced technique that may require some experimentation.

1. Start **SYMDEB** with the executable file and note the hexadecimal values of the first few instructions of the program.
2. Quit **SYMDEB** and rename the file so that its extension is not **.EXE** or **.HEX**. For example, change `file.exe` to `file.e`.
3. Start **SYMDEB** with the renamed executable file. **SYMDEB** will not strip off the MS-DOS file header as it normally does with **.EXE** and **.HEX** files. Therefore, the first instructions will be an attempt by **SYMDEB** to make sense of the data in the file header. They will not be the initial instructions of the program. (Don't load symbol files, since all symbolic data will be incorrect.)
4. Use the Search command (**S**) and the value of the first instructions to find the start of the program. This may take some trial and error. The starting address will vary, depending on the order of segments and other factors.
5. Once you have found the start of the program, you can find the instructions that need to be modified and make the appropriate changes.
6. Set the parameters for the Write command and write the whole file, including the file header, to disk. Make sure you include the file header in the program length entered to the **BX:CX** register pair.
7. Quit **SYMDEB** and rename the file back to its original name.

Examples

```
-N b:bell.com
-R BX 00
-R CX 0A
-W 100
-
```

The first example writes 10 (0Ah) bytes to the file named `bell.com` on Drive B. The bytes to be written start at address 100. The program `bell.com` is shown in section 4.6.1.

```
-W workspace 2 34 3
-
```

The example above writes three logical records to Drive C, starting at record number 34h. The bytes to be written start at the address `workspace`.

4.7 Sample SYMDEB Session

This sample session gives examples of commonly used **SYMDEB** commands. The assembly-language program used in the session is called `count.exe`. It prompts for a file name, opens the specified file, counts the words in the file, and prints the total on the screen. The source code for the program is shown on the next few pages. In order to keep the code as short as possible, the program has minimal error checking and prints the total in hexadecimal. This source file is included on your distribution disk.

Note the following points about the source file:

- The first line, after the macros, in the source file declares public each of the variable names used to store program data.
- The next two lines declare public some of the labels used in the program code. Only labels at key points that might be accessed by **SYMDEB** are declared.
- Several labels declared in the code are not used by any statement in the code. For example, `get_file`, `open_file`, and `conv_hex` are not used by any jump or loop instructions. They are placed at important points in the code so that **SYMDEB** can access those addresses by name.

  When developing your own programs, you may want to temporarily place symbols at problem areas. Declare these labels public for testing, and then remove them when the program is debugged.
- All numbers in the source code are specified in hexadecimal. This makes it easier to compare the code to **SYMDEB** displays, which always show hexadecimal numbers.
- The source code contains a bug that will be identified and corrected during the sample session.

```
dosint      MACRO  function             ;; Call the DOS interrupt
            mov    ah,function          ;; Put function number in AH
            int    21h
            ENDM

error       MACRO  errnum               ;; Display error and exit
            mov    dx,OFFSET err&errnum;; Load address of error message
            dosint 09h                  ;; Display string function
            mov    al,errnum            ;; Exit with return code of errnum
            dosint 4Ch                  ;; Quit
            ENDM

PUBLIC      prompt,namebuf,fname,buffer,err1,err2,count,new_flag
PUBLIC      get_file,open_file,ok,buff_read,done,conv_hex,rotate
PUBLIC      quit,word_c,next_char,new_word,old_word,out_word,get_out
```

```
stack       SEGMENT word stack 'STACK'
            DB     100h DUP(?)
stack       ENDS

data        SEGMENT word public 'DATA'
prompt      DB     'Enter file name: $'
namebuf     DB     15h,?                 ; Maximum length of file name
fname       DB     15h DUP(?)            ;   is 15h (21d)
buffer      DB     800h DUP(?)           ; Buffer size is 800h (2048d)
err1        DB     'Can''t access file',0Dh,0Ah,'$'
err2        DB     'I/O error',0Dh,0Ah,'$'
count       DW     0                     ; Initialize word count to 0
new_flag    DB     1                     ; Initialize new word to true (1)
data        ENDS

code        SEGMENT byte public 'CODE'
            ASSUME cs:code,ds:data

start:      mov    ax,data
            mov    ds,ax                 ; Load data segment address

            mov    dx,OFFSET prompt      ; Load address of prompt string
            dosint 09h                   ; Display it
get_file:   mov    dx,OFFSET namebuf     ; Load address for file name buffer
            dosint 0Ah                   ; Get file name string
            mov    si,dx                 ; Set SI to start of file name buffer
            mov    bl,BYTE PTR [si+1]    ; Put the number of bytes read in BL
            mov    BYTE PTR [si+bx+2],0; Put 0 at end to make ASCIIZ string
                                         ;   (0 overrides CR from prompt)
            mov    dl,0Ah                ; Load linefeed character
            dosint 02h                   ; Print it

open_file:  mov    dx,OFFSET fname       ; Load offset of ASCIIZ string
            xor    al,al                 ; Set code 0 - open for reading
            dosint 3Dh                   ; Try to open the file
            jnc    ok                    ; If opened, then process file
access:     error  1                     ;   else error macro

ok:         mov    bx,ax                 ; Move file handle to BX
            mov    dx,OFFSET buffer      ; Give address to dump file contents
io_loop:    mov    cx,800h               ; Set buffer size
            dosint 3Fh                   ; Read a buffer of data from file
buff_read:  jc     io_err                ; If there's a read error, then quit
            cmp    ax,0                  ;   else see if we read anything
            je     done                  ; If not, we're done
            call   word_c                ;   else count what we read
            jmp    SHORT io_loop         ; Do it again
io_err:     error  2                     ; Error macro

done:       dosint 3Eh                   ; Close (file handle already in BX)

            mov    bx,count              ; Put count in BX for processing
conv_hex:   mov    cl,4                  ; Load number of bits to rotate
            mov    ch,4                  ; Load count for digits
rotate:     rol    bx,cl                 ; Rotate left digit to right
            mov    dl,bl                 ; Move to DL for processing
            and    dl,0Fh                ; Mask off left digit
            add    dl,30h                ; Convert to ASCII digit
            cmp    dh,3Ah                ; Is it greater than 9?
            jl     show                  ; If not, display character
            add    dl,07h                ;   else convert hex letter
show:       dosint 02h                   ; Display character function
            dec    ch                    ; Decrement the digit count
```

```
            jnz     rotate              ; If count isn't zero, do it again

quit:       xor     al,al               ;   else set 0 for return code
            dosint  4Ch                 ; Return to DOS function

word_c      PROC    NEAR                ; Procedure to count words in buffer
            push    bx                  ; Save BX - it has file handle
            mov     si,OFFSET buffer-1  ; Load address one byte before buffer
            mov     bx,0                ; Set BX to 0 for word count
            mov     cx,ax               ; Put number of characters read in CX
            mov     ah,new_flag         ; Set new word flag (AH)

next_char:  inc     si                  ; Bump index (adjust on first pass)
            mov     al,[si]             ; Get next character
            cmp     al,20h              ; Compare to space
            jle     out_word            ; If less, we're not in a word
            cmp     ah,1                ;   else is new word flag TRUE?
            je      new_word            ; If flag is TRUE, it's a new word
            jmp     old_word            ;   else it's an old word

new_word:   inc     bx                  ; Bump word count
            xor     ah,ah               ; Set new word flag to FALSE (0)
old_word:   loop    next_char           ; Get next character
            jmp     get_out             ; Fall through at end of buffer
out_word:   mov     ah,1                ; Set new word flag to true (1)
            loop    next_char           ; Get next character

get_out:    add     count,bx            ; Add buffer count to variable
            mov     new_flag,ah         ; Save current flag status
            pop     bx                  ; Restore file handle
            ret
word_c      ENDP

code        ENDS
            END     start
```

4.7.1 Assembling and Loading

The steps for assembling and loading `count.exe` are shown below. The example assumes that all files are on the same drive.

1. Assemble the program. You may want to print a listing file for comparison, as shown below:

   `MASM count,,;`

2. Link the object file using the **/MAP** option:

   `LINK count,,/MAP;`

3. Create a symbol file:

   `MAPSYM count`

4. Start **SYMDEB** with the symbol file, the executable file, and any options you wish to use:

   `SYMDEB /S/K/"R;X?*" count.sym count.exe`

In the example, the /S option is used so that the program screen will be separate from the **SYMDEB** screen. The /K option is used so that we can escape if we accidentally get into an endless loop. The start-up command option is used to start with a register display and a list of symbols.

The example assumes you have an IBM Personal Computer. If you have an IBM-compatible computer, you should add the /I option so that the /S and /K options will be functional. If your computer is not an IBM or compatible, you can leave out the /S and /K options, since they will have no effect.

4.7.2 Examining a Program with SYMDEB

In the following session, hexadecimal numbers are used except where noted. When you start **SYMDEB** with the command line shown in the previous section, the following display appears:

```
Microsoft Symbolic Debug Utility
Version 4.00
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
AX=0000  BX=0000  CX=0A09  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=292A  ES=292A  SS=293A  CS=29CE  IP=000C   NV UP EI PL NZ NA PO NC
29CE:000C B84A29          MOV    AX,DATA

CODE: (29CE)
0018 GET_FILE  002E OPEN_FILE 0046 OK        0052 BUFF_READ 006B DONE
0073 CONV_HEX  0077 ROTATE    0091 QUIT      0097 WORD_C    00A4 NEXT_CHAR
00B3 NEW_WORD  00B6 OLD_WORD  00BB OUT_WORD  00BF GET_OUT
DATA: (294A)
0000 PROMPT
0012 NAMEBUF   0014 FNAME     0029 BUFFER    0829 ERR1      083D ERR2
0849 COUNT     084B NEW_FLAG
-
```

The first lines after the start-up message show the register status. These lines are produced with the first command (R) specified with the start-up command option. Notice that the stack pointer (**SP** register) is at 100h, the number of bytes assigned to the stack.

The second command (X?*) specified with the start-up command option displays all the symbols loaded from the symbol file.

The first few instructions load the segment and display a prompt. We'll skip them and start by going directly to the instructions that get a file name for processing:

```
-G get_file
AX=0924  BX=0000  CX=0A09  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=294A  ES=292A  SS=293A  CS=29CE  IP=0018   NV UP EI PL NZ NA PO NC
CODE:GET_FILE:
29CE:0018 BA1200         MOV     DX,0012
-
```

According to the symbol display shown when **SYMDEB** was started, the symbol `get_file` is at address 18h. The register display confirms that after going to `get_file`, the instruction pointer (**IP**) is at address 18h.

Note

If you did not start **SYMDEB** with the **/S** option, the prompt `Enter file name:` will appear at this point. This session includes information about the double-screen display available with IBM and compatible computers. If your computer doesn't have this capability, all the prompts and displays described for the program screen will actually appear on the **SYMDEB** screen.

Now take a look at the next few instructions using the Unassemble command (**U**):

```
-U
29CE:001B B40A           MOV     AH,0A
29CE:001D CD21           INT     21
29CE:001F 8BF2           MOV     SI,DX
29CE:0021 8A5C01         MOV     BL,[SI+01]
29CE:0024 C6400200       MOV     Byte Ptr [BX+SI+02],00
29CE:0028 B20A           MOV     DL,0A
29CE:002A B402           MOV     AH,02
29CE:002C CD21           INT     21
-
```

Notice that an Unassemble command with no argument starts at the next instruction after the current address (1Bh, in this case). Step through the next few instructions with the Trace (**T**) and PTrace (**P**) commands:

```
-T
AX=0924  BX=0000  CX=0A09  DX=0012  SP=0100  BP=0000  SI=0000  DI=0000
DS=294A  ES=292A  SS=293A  CS=29CE  IP=001B   NV UP EI PL NZ NA PO NC
29CE:001B B40A           MOV     AH,0A
-T
AX=0A24  BX=0000  CX=0A09  DX=0012  SP=0100  BP=0000  SI=0000  DI=0000
DS=294A  ES=292A  SS=293A  CS=29CE  IP=001D   NV UP EI PL NZ NA PO NC
29CE:001D CD21           INT     21  ;Buffered Keyboard Input
```

```
-P
AX=0A00  BX=0000  CX=0A09  DX=0012  SP=0100  BP=0000  SI=0000  DI=0000
DS=294A  ES=292A  SS=293A  CS=29CE  IP=001F   NV UP EI PL NZ NA PO NC
29CE:001F 8BF2          MOV     SI,DX
-
```

Notice how the registers change with each instruction. The PTrace instruction is not strictly necessary for skipping over interrupt 21h, but it is a good idea to get in the habit of using it, since **SYMDEB** will trace through any interrupt except 21h. Tracing interrupts is sometimes useful, but usually you will want to execute them.

After you execute MS-DOS function 0Ah, **SYMDEB** waits for you to enter a file name. If you started **SYMDEB** with the **/S** option, the program screen will temporarily replace the **SYMDEB** screen at this point. In this session, `count.exe` is used to count the words in `count.asm`. Enter `count.asm` at the file-name prompt.

The results can be examined with the Dump command (**D**):

```
-D namebuf fname-1
294A:0010        15 09                                          ..
-D fname buffer-1
294A:0010              63 6F 75 6E-74 2E 61 73 6D 0D 00 00      count.asm...
294A:0020  00 00 00 00 00 00 00 00-00                       .........
-
```

The dump is in the Dump Bytes (**DB**) format (the default when you start **SYMDEB**). The first byte of `namebuf` contains the maximum number of bytes available for the file name as set in the source code (15h). The second byte contains the actual number of characters entered (09h). The dump of `fname` confirms that the variable is indeed 15h bytes long and that 09h ASCII bytes and a carriage return (0Dh) were entered. You can check the *Microsoft MS-DOS Programmer's Reference Manual* or some other MS-DOS reference book to confirm that this is the proper format for strings entered with MS-DOS function 0Ah.

The next few instructions change `fname` to the ASCIIZ format (a String terminated by a null) used by the file functions of MS-DOS Version 2.0 and later:

```
-T
AX=0A00  BX=0000  CX=0A09  DX=0012  SP=0100  BP=0000  SI=0012  DI=0000
DS=294A  ES=292A  SS=293A  CS=29CE  IP=0021   NV UP EI PL NZ NA PO NC
29CE:0021 8A5C01        MOV     BL,[SI+01]                    DS:0013=09
```

```
-T
AX=0A00  BX=0009  CX=0A09  DX=0012  SP=0100  BP=0000  SI=0012  DI=0000
DS=294A  ES=292A  SS=293A  CS=29CE  IP=0024   NV UP EI PL NZ NA PO NC
29CE:0024 C6400200       MOV    Byte Ptr [BX+SI+02],00            DS:001D=0D
-T
AX=0A00  BX=0009  CX=0A09  DX=0012  SP=0100  BP=0000  SI=0012  DI=0000
DS=294A  ES=292A  SS=293A  CS=29CE  IP=0028   NV UP EI PL NZ NA PO NC
29CE:0028 B20A           MOV    DL,0A
-
```

Notice how memory locations in operands are expanded on the far right of the screen. For example, the operand `[BX+SI+02]` evaluates to `DS:001D=0D`, which means that memory offset 1Dh (09+12+02) of the data segment contains 0Dh (line feed). The instruction

```
MOV     Byte Ptr [BX+SI+02],00
```

replaces the line feed with a zero as illustrated by the dump below. Compare the tenth byte of this dump with the same byte in the earlier dump of `fname`.

```
-D fname buffer-1
294A:0010                 63 6F 75 6E-74 2E 61 73 6D 00 00 00      count.asm...
294A:0020  00 00 00 00 00 00 00 00-00                            .........
-
```

If you started **SYMDEB** with the **/S** option, you can enter a backslash (\) to see the current status of the program screen. If you do this, notice that the cursor is at the start of the first line. This is because a carriage return was provided without a line feed. The next two instructions solve this problem by printing a line feed.

Now execute the next few instructions and examine the status of the registers after opening a file and reading a buffer full of data:

```
-G buff_read
AX=0800  BX=0005  CX=0800  DX=0029  SP=0100  BP=0000  SI=0012  DI=0000
DS=294A  ES=292A  SS=293A  CS=29CE  IP=0052   NV UP EI PL ZR NA PE NC
CODE:BUFF_READ:
29CE:0052 720A           JB     BUFF_READ+0C (005E)
```

At this point **CX** still contains the size of the file input buffer, **BX** contains the file handle (05h, in the example), and **DX** contains the offset of the input buffer. Interrupt 3Fh has just been used to read the first 800h (2048 decimal) bytes of text from the file to the buffer. The following ASCII dump shows the contents of the buffer:

```
-DA buffer L 100
294A:0029  dosint      MACRO  function            ;; Call t
294A:0059  he DOS interrupt ..            mov    ah,functio
294A:0089  n          ;; Put function number in AH ..
```

```
.
.
.
294A:07A9  i+1]  ; Put the number of bytes read in BL..
294A:07D9          mov    BYTE PTR [si+bx+2],0; Put 0 at en
294A:0809  d to make ASCIIZ string ..
-
```

When you enter the **DA** command, several screens full of data scroll past. Notice the double dots scattered throughout the text. These are carriage-return/line-feed combinations, as you can confirm if you dump bytes instead of ASCII characters. If you typed the source code yourself, you may see dots representing tab characters instead of series of spaces (depending on how your editor handles tabs).

Next, set some breakpoints to examine different parts of the program:

```
-BP next_char "DA ds:si+1 L 1;R"
-BP new_word
-BP buff_read "DW count count+1;R"
-
```

These breakpoints are chosen because they represent three levels within the program. Two of them have quoted commands that will be executed each time the breakpoint is reached. To execute to the first break, enter the Go command (**G**):

```
-G
294A:0029  d
AX=0100  BX=0000  CX=0800  DX=0029  SP=00FC  BP=0000  SI=0028  DI=0000
DS=294A  ES=292A  SS=293A  CS=29CE  IP=00A4   NV UP EI PL NZ NA PE NC
CODE:NEXT_CHAR:
29CE:00A4 46            INC    SI                             ;BR0
-
```

The program stops each time it reads in a new character. The quoted command DA `ds:si+1 L 1` displays the character that is about to be read and the quoted command R displays the registers. Enter the Go command again. This takes you to the second breakpoint:

```
-G
AX=0164  BX=0000  CX=0800  DX=0029  SP=00FC  BP=0000  SI=0029  DI=0000
DS=294A  ES=292A  SS=293A  CS=29CE  IP=00B3   NV UP EI PL ZR NA PE NC
CODE:NEW_WORD:
29CE:00B3 43            INC    BX                             ;BR1
-
```

If you enter the Go command several times, you will stop at the first breakpoint for each new character and at the second breakpoint every time you start a new word. Notice how **BX**, which contains the word count, is incremented every time you reach the second breakpoint (BR1). Reading in every character is a slow process. You can speed things up by disabling the first breakpoint (BR0):

```
-BD 0
-
```

Now when you enter the Go command a few times, you move through the buffer faster, stopping only when you reach a new word. You can speed things up more by disabling the second breakpoint. The example display also shows a breakpoint list:

```
-BD 1
-BL
0 d 29CE:00A4 [CODE:NEXT_CHAR]  "DA DS:SI+1 L 1;R"
1 d 29CE:00B3 [CODE:NEW_WORD]
2 e 29CE:0052 [CODE:BUFF_READ]  "DW COUNT COUNT+1;R"
-G
294A:0849  00E1
AX=0800  BX=0005  CX=0800  DX=0029  SP=0100  BP=0000  SI=0828  DI=0000
DS=294A  ES=292A  SS=293A  CS=29CE  IP=0052   NV UP EI PL NZ NA PE NC
CODE:BUFF_READ:
29CE:0052 720A          JB      BUFF_READ+0C  (005E)          ;BR2
-
```

From the breakpoint list, you can see that breakpoints 0 and 1 are still in memory. You can turn them back on with the Breakpoint Enable command (**BE**) any time you want.

When you enter the Go command, execution now stops after reading a whole buffer. The quoted command DW count count+1 shows the variable where the current word total is stored. The word count is E1h (225 decimal) after reading the first buffer.

The sample file contains only a few buffers of text, so after you enter the Go command several times, the program will terminate without finding the breakpoint. You will see the following message:

```
-G

Program terminated normally (0)
-
```

When the program terminates, use the Quit command (**Q**) to return to DOS. If you started **SYMDEB** with the **/S** option, the program screen should look like this:

```
Enter file name: count.asm
02;8
```

The total shown (`02;8`) is not a valid hexadecimal number. (If you typed `count.asm` yourself with different comments or spacing, you might not see this problem, but it will become obvious if you try counting the words in other text files.) The bug is probably in the routine that converts binary numbers to hexadecimal. To find and correct it, restart **SYMDEB**. (Don't try to run the program without quitting **SYMDEB** and restarting.) Then enter the following command:

```
-G conv_hex
AX=0004  BX-02B8  CX=0800  DX=0029  SP=0100  BP=0000  SI=0775  DI=0000
DS=294A  ES=292A  SS=293A  CS=29CE  IP=0073   NV UP EI PL ZR NA PE NC
CODE:CONV_HEX:
29CE:0073 B104          MOV    CL,04
-
```

This shows the status of the registers the first time through the conversion loop. Notice that **BX** contains the total word count taken from the variable `count`. This is the number we want to print. To examine processing of the digit that prints incorrectly, set a breakpoint with a passcount of three:

```
-BP rotate 3
-G
AX=0232  BX=B802  CX=0204  DX=0032  SP=0100  BP=0000  SI=0775  DI=0000
DS=294A  ES=292A  SS=293A  CS=29CE  IP=0077   NV UP EI PL NZ NA PO CY
CODE:ROTATE:
29CE:0077 D3C3          ROL    BX,CL                          ;BRO
-
```

Notice that the register containing the loop count (CH) contains 2. The loop has already been executed twice and this is the third pass. Trace through the next four instructions:

```
-T 4
AX=0232  BX=802B  CX=0204  DX=0032  SP=0100  BP=0000  SI=0775  DI=0000
DS=294A  ES=292A  SS=293A  CS=29CE  IP=0079   NV UP EI PL NZ NA PO CY
29CE:0079 8AD3          MOV    DL,BL
AX=0232  BX=802B  CX=0204  DX=002B  SP=0100  BP=0000  SI=0775  DI=0000
DS=294A  ES=292A  SS=293A  CS=29CE  IP=007B   NV UP EI PL NZ NA PO CY
29CE:007B 80E20F        AND    DL,0F
AX=0232  BX=802B  CX=0204  DX=000B  SP=0100  BP=0000  SI=0775  DI=0000
DS=294A  ES=292A  SS=293A  CS=29CE  IP=007E   NV UP EI PL NZ NA PO NC
29CE:007E 80C230        ADD    DL,30                          ;'0'
AX=0232  BX=802B  CX=0204  DX=003B  SP=0100  BP=0000  SI=0775  DI=0000
DS=294A  ES=292A  SS=293A  CS=29CE  IP=0081   NV UP EI PL NZ NA PO NC
29CE:0081 80FE3A        CMP    DH,3A                          ;':'
-
```

The first instructions seem all right. The number in **BX** is rotated and its lower byte moved to **BL**. The second digit is masked off and 30h is added to convert to an ASCII digit. But then 3Ah (the ASCII code for the character one above the digit 9) is compared to **DH** (which contains zero). The number we want to compare is in **DL**, not **DH**. That's probably the bug. Use the Assemble command (**A**) to fix it:

```
-A
29CE:0081 cmp dl,3A
29CE:0084
-
```

You don't need to supply an address since the Assemble command assumes the current **IP** address if none is specified. Enter the correct instruction on the first line, then press the RETURN key on the next line to indicate you don't want to assemble any more instructions. Now trace through the next three instructions:

```
-T 3
AX=0232  BX=802B  CX=0204  DX=003B  SP=0100  BP=0000  SI=0775  DI=0000
DS=294A  ES=292A  SS=293A  CS=29CE  IP=0084   NV UP EI PL NZ NA PO NC
29CE:0084 7C03          JL      ROTATE+12 (0089)
AX=0232  BX=802B  CX=0204  DX=003B  SP=0100  BP=0000  SI=0775  DI=0000
DS=294A  ES=292A  SS=293A  CS=29CE  IP=0086   NV UP EI PL NZ NA PO NC
29CE:0086 80C207        ADD     DL,07
AX=0232  BX=802B  CX=0204  DX=0042  SP=0100  BP=0000  SI=0775  DI=0000
DS=294A  ES=292A  SS=293A  CS=29CE  IP=0089   NV UP EI PL NZ AC PE NC
29CE:0089 B402          MOV     AH,02
-
```

The digit is now adjusted from a semicolon (ASCII 3Bh) to a C (ASCII 42h). If the instruction hadn't been changed, the program would have jumped over the adjustment instruction. Use the Go command (**G**) twice to run the rest of the program. It should print the word count correctly now.

You can now fix the bug in the source code and reassemble. This type of minor bug is the kind that is often difficult to spot from reading source code. **SYMDEB** lets you see what is happening inside the processor so that you can examine operations and locate bugs easily.

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 an assembly-language program. 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 for use as a debugging aid to 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 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 file name when you invoke the assembler. You create the cross-reference listing by invoking **CREF** and supplying the name of the cross-reference file.

Sections 5.2 and 5.3 explain how to create a cross-reference file for **CREF** and how to use **CREF** to create a cross-reference listing.

5.2.1 Creating a Cross-Reference File

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

To create a cross-reference file using a prompt, enter MASM, then supply the file name in response to the fourth command prompt. For example, to create a cross-reference file `test.crf` for the program `test.asm`, type

```
MASM

Source filename [.ASM]: test
Object filename [test.OBJ]: test
Source listing  [NUL.LST]: test
Cross-Reference [NUL.CRF]: test
```

If you do not type a file name after the "Cross-Reference" prompt, the assembler will not create a cross-reference file. If you do not supply an extension, **MASM** uses the extension **.CRF**. This is the extension expected by **CREF** and is recommended for all cross-reference files.

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 (`test.crf`) for the source file (`test.asm`), type:

```
MASM  test,test,test,test
```

This command also creates object and listing files for the program while the program is being assembled. **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. See Section 2.2.1 for more information.

5.2.2 Creating a Cross-Reference Listing Using Prompts

You can direct **CREF** to prompt you for file names when it starts by typing just the **CREF** command name. **CREF** displays a series of prompts asking for the file names. To start **CREF** with prompts, follow these steps:

1. From the MS-DOS prompt, type

   ```
   CREF
   ```

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

   ```
   Cross-Reference [.CRF]:
   ```

2. Type the name of the cross-reference file that you wish to convert to a cross-reference listing, then press the RETURN key. You need not supply a file-name extension if your cross-reference file already has the extension **.CRF**. If your cross-reference file does not have this extension, you must supply the correct extension at this time.

   Once you supply a file name, **CREF** displays the following prompt:

   Listing [*filename*.REF]:

   Note that *filename* is the default file name for the cross-reference listing.

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

Once you have supplied the file names, **CREF** reads the cross-reference file and creates the new listing. It also displays the number of symbols in the cross-reference file.

Example

```
CREF
Microsoft Cross Reference Utility  Version 3.50
(C)Copyright Microsoft Corp 1981, 1983, 1984, 1985

Cross reference [.CRF]: test
Listing [test.REF]:

8 Symbols
```

In the example above, **CREF** creates reads `test.crf` and processes it to produce `test.ref`. Eight symbols were cross-referenced.

5.2.3 Creating a Cross-Reference Listing Using a Command Line

You can create a cross-reference listing by typing **CREF** followed by the names of the files you want to process. The command line has the form:

CREF *crossreferencefile* ⟦,*crossreferencelisting*⟧ ⟦;⟧

The *crossreferencefile* is the name of the cross-reference file created by **MASM**, and the *crossreferencelisting* is the name of the readable ASCII file you wish to create.

If you do not supply file-name extensions when you name the files, **CREF** will automatically provide **.CRF** for the cross-reference file and **.REF** for the cross-reference listing. If you do not want these extensions, you must supply your own.

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

You can specify a directory or disk drive for either of the files. You can also name output devices such as CON (display console) and PRN (printer).

Examples

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

The first example uses the cross-reference file `test.crf` to create a cross-reference listing `test.ref`. It is equivalent to

```
CREF   test,test
```

or

```
CREF   test;
```

The following example directs the cross-reference listing to the screen. No file is created.

```
CREF   test,con
```

5.3 Cross-Reference Listing Format

The cross-reference listing contains the name of each symbol defined in your program. Each name is followed by a list of line numbers representing the line or lines in the program listing file in which a symbol is defined or used. Line numbers in which a symbol is defined are 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 the source file. See Section 9.6 in the *Microsoft Macro Assembler Reference Manual.*

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

```
quit        MACRO               ; Return to DOS
            mov     ah,4Ch      ;;DOS exit function
            int     21h
            ENDM

max         EQU     65535

            EXTRN   work:NEAR
```

```
stack       SEGMENT para public 'STACK'
            DB      256 DUP(?)
stack       ENDS

data        SEGMENT public 'DATA'
buffer      DW      100 DUP(?)
data        ENDS

code        SEGMENT public 'CODE'
            ASSUME  cs:code,ds:data

start:      mov     ax,data     ; Load address
            mov     ds,ax
            call    work        ; Call procedure
            quit                ; Call macro
code        ENDS
            END     start
```

To assemble the program and create a cross-reference file, type:

```
MASM test,test,test,test
```

The listing file `test.lst` produced by this assembly will look like the following listing (the tables at the end of the listing are not shown):

```
Microsoft MACRO Assembler  Version 4.00      9/25/85 13:58:46

                                             Page     1-1

      1                               quit    MACRO
      2                                       mov     ah,4Ch
      3                                       int     21h
      4                                       ENDM
      5
      6 = FFFF                        max     EQU     65535
      7
      8                                       EXTRN   work:NEAR
      9
     10 0000                          stack   SEGMENT para public 'STACK'
     11 0000   0100[                          DB      256 DUP(?)
     12               ??
     13                           ]
     14
     15 0100                          stack   ENDS
     16
     17 0000                          data    SEGMENT public 'DATA'
     18 0000   0064[                  buffer  DW      100 DUP(?)
     19               ????
     20                           ]
     21
     22 00C8                          data    ENDS
     23
     24 0000                          code    SEGMENT public 'CODE'
     25                                       ASSUME  cs:code,ds:data
     26
     27 0000   B8 ---- R              start:  mov     ax,data
     28                                       mov     ds,ax
     29 0003   E8 0000 E                      call    work
     30                                       quit
```

```
   31 0006  B4 4C              1          mov     ah,4ch
   32 0008  CD 21              1          int     21h
   33 000A                       code     ENDS
   34                                     END     start
```

To create a cross-reference listing of the file `test.crf`, type:

```
CREF test,test
```

The resulting cross-reference listing in the file `test.ref` will have the following format:

```
Microsoft Cross-Reference  Version 4.00   Wed Sep 25 12:12:40 1985

  Symbol Cross-Reference     (# is definition)             Cref-1

BUFFER . . . . . . . . . . .   18     18#

CODE . . . . . . . . . . . .   24     24#    24     25     33

DATA . . . . . . . . . . . .   17     17#    17     22     25     27

MAX  . . . . . . . . . . . .    6      6#

QUIT . . . . . . . . . . . .   30

STACK  . . . . . . . . . . .   10     10#    10     15
START  . . . . . . . . . . .   27     27#    34

WORK . . . . . . . . . . . .    8      8#    29

 8 Symbols
```

Compare the line numbers in the cross-reference listing to the listing file. Don't try to count lines in the source file, since line numbers there usually won't match line numbers in the listing and cross-reference listing files.

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 to include routines and variables used, but not defined, in the source code of 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 modules containing routines or variables used by the program are extracted and linked.

LIB can perform the following four tasks with library files:

- Create a new library
- Check an existing library for consistency
- Print a library-reference listing
- Maintain libraries

The last task, maintaining libraries, is the most common. The command symbols in Table 6.1 are used in library maintenance. They are discussed in detail in Section 6.6.

Table 6.1

LIB Commands

Symbol	Meaning
+	Add
–	Delete
–+	Replace
*	Copy
–*	Move

Each of the four kinds of **LIB** tasks can be done with prompts, a command line, a response file, or a combination of the three methods.

This chapter first describes in a general way the three methods of starting and using **LIB**. It then describes in detail each of the four tasks you can perform with **LIB**. **LIB** commands are discussed in connection with the fourth task, maintaining library files.

6.2 Starting and Using LIB

You can give the names of files for **LIB** to work on, and the commands specifying what you want **LIB** to do, in three ways: by answering a series of prompts, by entering a command line, or by supplying a response file. You can stop **LIB** at any time by pressing CONTROL-C.

6.2.1 Starting LIB with Prompts

You can let **LIB** prompt you for the information it needs by typing **LIB** 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 file-name extension, **LIB** supplies the extension **.LIB**. If you wish to create a new library, type the new name and press the RETURN key.

   LIB now looks for the specified library file. If it finds the file, **LIB** displays the next prompt. If it does not find the file, **LIB** displays the prompt:

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

   Type `y` to create the library file or type `n` to return to the MS-DOS command level.

If you want to change the default page size, you can specify the option:

/**PAGESIZE:***number*

after entering the library name. The *number* is the desired page size. See Section 6.2.4.

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

```
Operations:
```

3. Type the command or commands you wish to perform 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** will then prompt for further 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 library-reference listing file and press the RETURN key. Make sure the file name has the extension you want. **LIB** will not provide a default extension. If you do not want a library-reference listing file, do not type a name—just press the RETURN key.

   If you did not give commands to modify the library, **LIB** creates the list file and returns to DOS at this point. If you did give commands at the "Operations" prompt, **LIB** displays the following prompt:

   ```
   Output library:
   ```

5. Type the name of the output file you wish to create and press the RETURN key. If you do not supply a file-name extension, **LIB** supplies the extension **.LIB**. You can press the RETURN key without giving a file name if you want **LIB** to use the name of the old library file. In this case, **LIB** saves a backup copy of the current library by replacing its **.LIB** extension with the extension **.BAK**.

LIB now carries out the commands you have requested.

You can direct **LIB** to select the default responses to all remaining prompts by typing a semicolon (;) at any prompt line after the "Library name" prompt. At any prompt, you can fill in the rest of the file names and commands in the command-line format (see Section 6.2.2). You must supply a path name for any file that is not on the current drive and directory.

Example

```
LIB

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

This example creates a new library called `math1.lib` from the contents of the old library `math.lib`. **LIB** also adds the modules in the object files `sin.obj`, `cos.obj`, `atan.obj`, and `exp.obj` to the new library. A library-reference listing file called `math` (with no extension) is created.

6.2.2 Starting LIB with a Command Line

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

LIB *oldlibrary* ⟦**/PAGESIZE:***number*⟧ ⟦*commands*⟧⟦**,**⟦*listfile*⟧ ⟦**,**⟦*newlibrary*⟧⟧⟧ ⟦**;**⟧

The *oldlibrary* names the library file to be worked on. If you do not supply a file-name extension, **LIB** supplies **.LIB**.

The **/PAGESIZE** option defines the page size of the library. The default page size is 16 bytes. This option is discussed in detail in Section 6.2.4.

The *commands* are **LIB** commands from among those listed in Section 6.6. They specify what tasks are to be performed on the given library. If you do not specify any commands, **LIB** will create a library cross-reference listing without doing any operations.

The optional *listfile* is the name of the library-reference listing file. If no file name is given, **LIB** does not create a *listfile*.

The optional *newlibrary* is the name of the new library file to which you wish to copy the modified library. If no file name is given, **LIB** uses the *oldlibrary* file name and renames the *oldlibrary* file by giving it the extension **.BAK**.

If one of the files specified in the command line is in another directory or on a different drive, you must supply an appropriate path name.

If you give a *listfile*, you must separate it from the last command with a comma (,). If you give a *newlibrary*, you must separate it from the *listfile* with a comma (,) or from the last command with two commas (,,).

You can use a semicolon (;) after any entry except the *oldlibrary* to direct **LIB** to use the default responses for the remaining entries. If used, the semicolon should be the last character on the command line.

Examples

```
LIB lang +heap;
```

The first example instructs **LIB** to add the module `heap` to the library `lang.lib`. The semicolon at the end of the command line tells **LIB** to use the default responses for the library-reference listing and the new library file. This means that no reference listing is created and that the changes are written back to the original library file. The old library file is renamed to `lang.bak`.

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

The second 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 added. **LIB** also creates a listing file for the library named `lang.lst`.

6.2.3 Starting LIB with a Response File

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

LIB @ *responsefile*

A response file can also be specified at any prompt, or at any position in a command line. The input from the response file will be treated exactly as if it had been entered at prompts or in a command line. However, note that carriage-return/line-feed combinations in the response file are treated the same as the RETURN key entered in response to a prompt, or a comma used in a command line.

When starting **LIB**, the *responsefile* must be the name of the response file, and it must be preceded by an at sign (@). If the file is in another directory or on another disk drive, a path name must be provided.

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

library ⟦/PAGESIZE:*number*⟧
⟦*commands*⟧
⟦*listfile*⟧
⟦*output-file*⟧

Elements that have already been provided at prompts or with a partial command line can be left out.

Each file name must appear on a separate line. Any number of commands may 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.

You can place a semicolon (;) on any line in the response file. When **LIB** encounters the semicolon, it automatically supplies default file names 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, **LIB** displays each response from your response file on the screen in the form of prompts. If the response file does not contain names for required files, **LIB** prompts for the missing names and waits for you to enter responses.

Note

A response file should end with a semicolon (;) or a carriage-return/line-feed combination. If you fail to provide a final carriage-return/line-feed in the file, **LINK** will display the last line of the response file and wait for you to press the RETURN key.

Example

```
plib
+cursor +heap +stack
cross.lst
```

This response file causes **LIB** to work on the library `plib.lib`. The commands in the second line instruct **LIB** to add the modules `cursor`, `heap`, and `stack` to the new library file. A library-reference listing called `cross.lst` is created. Since no name is specified for the output library, the new library file will have the same name as the old. The old version will be renamed to `plib.bak`.

6.2.4 Setting the Library-Page Size

You can set the library-page size by adding a page-size option after the library-file name in the **LIB** command line or after the new library-file name at the "Library name" prompt. The option has the form:

/**PAGESIZE:***number*

The *number* specifies the new page size. It must be an integer value representing a power of 2 between the values 16 and 32768. The option name can be abbreviated to /**P:***number*.

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), calculated from the beginning of the file. The default page size is 16 bytes 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 *number*/2 bytes of storage space is wasted (where *number* is the page size). In most cases, a small page size is advantageous; you should use a small page size unless you need to put a very large number of modules in a library.

Examples

```
LIB

Library name: math /PAGESIZE:256
Operations: +tangent
List file: mathtan.lst
Output library: mathtan
```

This example creates a new library file named `mathtan.lib` from the old library file `math.lib`. The page size is set to 256 bytes. The module `tangent` is added to the new library file and a library-reference listing called `mathtan.lst` is created.

The example below shows how the same library would be created with a command line:

```
LIB math/P:256,+tangent,mathtan.lst,mathtan
```

6.3 Creating a New Library

You can create a new library by 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 in response to the "Library name" prompt, **LIB** searches for that file. If the specified library file does not exist, **LIB** displays the following prompt:

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

Type `y` to create the file or `n` to abort the library session.

If no file exists for a library name given in a command line, **LIB** creates the library, processes the commands, and fills in the rest of the command line. If you give the new library name in a command line without additional commands or files, **LIB** changes to prompt mode and asks if you want to create the new library.

Examples

```
LIB

Library name: display /PAGESIZE:64
Library does not exist.  Create? y
Operations: +cursor +scroll +position
List file:
```

In the example above, a library called `display.lib` is created from the object files `cursor.obj`, `scroll.obj`, and `position.obj`. The new library is created with a page size of 64 bytes.

You could create the same library with the following command line:

```
LIB display /P:64 +cursor +scroll +position;
```

6.4 Checking a Library's Consistency

You can check to make sure a library's contents are consistent and usable by running **LIB** without commands. Type a semicolon (;) at the "Operations" prompt or after the file name at the "Library name" prompt. You can also type a command line with the name of the library you wish to check followed by a semicolon. **LIB** then makes sure all entries in the library can be accessed. If any problems are discovered, **LIB** displays an error message. Otherwise, it displays nothing.

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.

Examples

```
LIB

Library name: math;
```

This example checks to make sure all modules in `math.lib` are valid and usable. You can do the same thing with the following command line:

```
LIB math;
```

6.5 Creating a Library-Reference Listing

You direct **LIB** to create a library-reference listing whenever you give a file name at the "List file" prompt or in the *listfile* position of a **LIB** command line. A library-reference listing consists of 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
```

You can get a listing of an existing file by pressing the RETURN key at the "Operations" prompt and entering a file name at the "List file" prompt. The same thing can be done in a command line by typing a comma (,) after the library name and then typing the name of the file containing the library-reference listing.

Examples

```
LIB

Library name: math
Operations:
List file: math
```

The example above creates a library-reference listing file called `math` (with no extension). The following command line does the same thing except that the library-reference listing is shown on the screen instead of being sent to a file:

```
LIB math,con
```

6.6 Maintaining Libraries

The **LIB** commands specify the maintenance tasks to be carried out on a given library. The commands are used to add, delete, and replace modules in a given library. They can also copy and move modules to new libraries.

Commands can be given on the **LIB** command line, in response to the **LIB** "Operations" prompt, or in a response file.

Make sure you have sufficient disk space to do the commands you specify. **LIB** may need additional space for a listing file and for a new library file. **LIB** will save the old version of a library file with the extension **.BAK** if you specify that the modified library file should have the same name as the original. You may get an error message if there is not enough space on the disk for both the new library file and the backup library file.

6.6.1 Adding a Module to a Library

Syntax

+*objectfile*

The Add command (+) adds the object module in the specified *objectfile* to the current library. The *objectfile* must be the file name of an object file. If you do not specify a file-name extension, **LIB** supplies **.OBJ** by default. If the file is in another directory or on a different disk, you must supply an appropriate path name. There must be no spaces between the plus sign (+) and the name.

LIB searches for the file you have named, and adds the object file's contents to the current library. **LIB** then strips the drive name, path name, and the file-name extension (if any) from the object-file name and places the resulting name in the library's table of contents. **LIB** always appends object modules to the end of the library file.

Examples

```
LIB math +sin.obj;
```

The first example adds the module in the file `sin.obj` to the library `math.lib`. No list file is created.

```
LIB \lib\math +cos, math;
```

The second example adds the module in the file `cos.obj` to the library `math.lib` in the `\lib` directory. A list file `math` (with no extension) is created.

```
LIB math +A:\src\atan;
```

The final 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. No list file is created.

6.6.2 Deleting Library Modules

Syntax

−*modulename*

The Delete command (−) deletes the object module identified by the placeholder *modulename* from the current library. The module name must be spelled exactly as it appears in the library's table of contents. Case is not significant when specifying module names.

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;
```

The first example deletes the module `sin` from the library `math.lib`. No list file is created.

```
LIB \lib\math -cos, math;
```

The second example deletes the module `cos` from the library `math.lib` in the `\lib` directory. The list file `math` (with no extension) is created.

```
LIB math +A:\src\atan -atan;
```

The final example deletes module `atan.obj` from 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 even though the Add command comes first on the command line.

6.6.3 Replacing Library Modules

Syntax

−+*modulename*

The Replace command (−+) replaces the module identified by *modulename* with the module in an object file having the same name. The *modulename* must have exactly the same spelling as the name in the library's table of contents (case is not significant). **LIB** first deletes this module, then searches the current working directory for a file having the same file name and the file-name extension **.OBJ**.

If the file is found, **LIB** adds it to the library file. 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 finds the file `cos.obj` in the current directory and adds the contents to the library file. No listing is created.

6.6.4 Copying Library Modules

Syntax

**modulename*

The Copy command (*) extracts from the library a copy of the module identified by *modulename*, and copies it to an object file having the same name. The *modulename* must have exactly the same spelling as the name in the library's table of contents (case is not significant). If the module is not in the library file, **LIB** displays an error message.

When **LIB** copies the module to an object file, it creates a file whose file name is the same as that of the module, but whose file-name 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. The module `cos` remains unchanged in the library file.

6.6.5 Moving Library Modules

Syntax

–**modulename*

The Move command (–*) moves the module identified by *modulename* from the current library to an object file having the same name as the module. The *modulename* must be spelled exactly as it appears in the library's table of contents (case is not significant). If the module is not in the library file, **LIB** displays an error message.

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`. No list file is created.

6.6.6 Combining Libraries

Syntax

+*libraryname*

The Add command (+) can also be used to add the contents of another library to the current library. The *libraryname* must be the name of the library file you wish to add. You must give the file-name extension of the file. Otherwise, **LIB** assumes the file is an object file.

LIB appends 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 Maintenance Utility (MAKE) automates the process of maintaining 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 has 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.

7.2 Using MAKE

To use MAKE, you must create a MAKE description file that defines the tasks you wish to accomplish and specifies the files on which these tasks depend. Once the description file exists, you invoke MAKE and supply the file name as a parameter. 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.

7.2.1 Creating a MAKE Description File

You can create a MAKE description file with a text editor. A MAKE description file consists of one or more target/dependent descriptions. Each description has the following general form:

targetfile : *dependentfiles*
    command1
    ⟦*command2*⟧
    .
    .
    .

The *targetfile* is the name of a file that may need updating, *dependentfile* is the name of a file on which the target file depends, and the commands are the names of executable files or MS-DOS internal commands.

The *targetfile* and *dependentfile* must be valid file names. A path name must be provided for any file that is not on the same drive and directory as the description file.

Any number of dependent files can be given, but only one target name is allowed. Dependent-file names must be separated by 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 valid MS-DOS command line consisting of an executable-file name or an MS-DOS internal command. Any number of commands can be given, but each must begin on a new line and must be preceded by a TAB, or by at least one space. The commands are carried out only if one or more of the dependent files has been modified since the target file was created.

One way to remember the **MAKE** format is to think of it as an "if-then" statement in the following format:

If a *dependentfile* is older than the *targetfile*, or
If a *dependentfile* does not exist,
    Then do *commands.*

You can give any number of target/dependent 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.

The pound character (#) is a comment character. All characters after the comment character on the same line are ignored. When comments appear in a *command* lines section, the comment character (#) must be the first character on the line (no leading white space). On any other lines, the comment character can appear anywhere.

Note

The order in which you place the target/dependent 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        #make a symbol file for debugging
#use the -l option to print information
       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`.

Note that a comment appears on the same line as the target description for `print.sym`. However, in the command lines section, the comment appears on a separate line, since the comment character (#) must be the first character on the line.

7.2.2 Starting MAKE

MAKE must be started with a command line. You cannot use prompts. The **MAKE** command line has the form:

MAKE ⟦*options*⟧ ⟦*macrodefinitions*⟧ *filename*

The *options* are one or more of the options described in section 7.2.3. The *macrodefinitions* are one or more macro definitions as described in Section 7.2.4. The *filename* is the name of a **MAKE** description file. A **MAKE** description file, by convention, has the same file name (but with no extension) as the program it describes. Although any file name 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 dependent file, it displays the command or commands from the target/dependent description, then executes the commands. If **MAKE** cannot find a specified file, it displays a message informing you that the file was not found. If the missing file is a target file, **MAKE** continues execution, since the missing file will in many cases be created by subsequent commands.

If the missing file is a dependent or command file, **MAKE** stops execution of the description file. **MAKE** also stops execution and displays the exit code if the command returns an error.

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.

7.2.3 Using MAKE Options

The options available with the **MAKE** command modify its behavior as described below.

Option	Action
/D	This option causes **MAKE** to display the last modification date of each file as the file is scanned.
/I	This option causes **MAKE** to ignore exit codes (also called return or "errorlevel" codes) returned by programs called by the **MAKE** description file. **MAKE** will continue execution of the next lines of the description file despite the errors.
/N	When this option is given, **MAKE** displays commands that would be executed by a description file, but does not actually execute the commands.
/S	This option causes **MAKE** to execute in "silent" mode. That is, lines are not displayed as they are executed.

Examples

```
MAKE /N test
```

The first example directs **MAKE** to display commands from the **MAKE** description file named `test` without executing them.

```
MAKE /D test
```

The second example directs **MAKE** to execute the instructions from `test`, displaying the last modification time of each file as it is scanned.

7.2.4 Using Macro Definitions

Macro definitions let you associate a symbolic name with a particular value. By using macro definitions, you can change values used in the description file without having to edit every line that uses a particular value.

The form of a macro definition is:

name=*value*

The form for using a previously defined macro definition is:

$(*name*)

Occurrences of the pattern $ (*name*) in the description file are replaced with the specified *value*. The *name* is converted to uppercase; `flags` and `FLAGS` are equivalent. If you define a macro name but leave the *value* blank, the *value* will be a null string.

Macro definitions can be placed in the **MAKE** description file or given on the **MAKE** command line. A *name* is also considered defined if it has a definition in the current environment. For example, if the environment variable **PATH** is defined in the current environment, occurrences of `$ (PATH)` in the description file will be replaced with the **PATH** value.

In the **MAKE** description file, each macro definition must appear on a separate line. Any white space (tab and space characters) between *name* and the equal-sign (=) or between the equal-sign and *value* is ignored. Any other white space is considered part of *value*. To include white space in a macro definition on the command line, enclose the entire definition in double quotation marks (").

If the same name is defined in more than one place, the following order of precedence applies:

1. Command line definition
2. Description file definition
3. Environment definition

Example

```
base=abc
buf=/B63

$(base).obj:        $(base).asm
        MASM $(base) $(buf),$(base),$(base),$(base)

$(base).exe:        $(base).obj \lib\math.lib
        LINK $(base),$(base),$(base) /map,\lib\math
```

The sample **MAKE** description file above shows macro definitions for the names `base` and `buf`. **MAKE** replaces each occurrence of `$(base)` with `abc`. If the description file is called `assemble`, you can give the following command:

```
MAKE base=def assemble
```

This command line enables you to override the definition of `base` in the description file, causing `def` to be assembled and linked instead of `abc`.

If you want to override the 63K buffer size specified by the macro `buf` in the **MAKE** description file and instead use the **MASM** default buffer size of 32K, you could start **MAKE** with the following command line:

```
MAKE buf= assemble
```

Since the value for `buf` is blank, it will be treated as a null string. However, since the null string was given from the command line, which has higher precedence than the definition in the description file, `buf` will be expanded to a null string and no option will be passed in the **MASM** command line.

7.2.5 Nesting Macro Definitions

Macro definitions can be nested. In other words, a macro definition can include another macro definition. For example, you might have the following macro definition in the **MAKE** description file `picture`:

```
LIBS=$(DLIB)\math.lib $(DLIB)\graphics.lib
```

You could then start **MAKE** with the following command line:

```
MAKE DLIB=d:\lib
```

In this case, every occurrence of the macro LIBS would be expanded to:

```
d:\lib\math.lib d:\lib\graphics.lib
```

Be careful to avoid infinitely recursive macros such as the following:

```
A = $(B)
B = $(C)
C = $(A)
```

7.2.6 Using Special Macros

MAKE recognizes three special macro names and will automatically substitute a value for each. The special names and their values are:

Name	Value Substituted
$*	Base name portion of the target (without the extension)
$@	Complete target name
$**	Complete list of dependencies

These macro names can be used in description files, as shown in the following example:

Example

```
test.exe: mod1.obj mod2.obj mod3.obj
         link $**, $@;
         mapsym $*
```

The example above is equivalent to the following:

```
test:exe: mod1.obj mod2.obj mod3.obj
         link mod1.obj mod2.obj mod3.obj, test.exe;
         mapsym test
```

7.2.7 Inference Rules

MAKE allows you to create inference rules that specify commands for target/dependent descriptions even when there is no explicit command in the **MAKE** description file. An inference rule is a way of telling **MAKE** how to produce a file with one type of extension from a file with the same base name and a second type of extension.

For example, if you define a rule for producing **.OBJ** files from **.ASM** files, the actual commands do not have to be repeated in the description file for each target/dependent description. Inference rules take the following form:

.dependentextension.targetextension **:**
    command1
    [[*command2*]]
    .
    .
    .

For lines that do not have explicit commands, **MAKE** looks for a rule that matches both the target's extension and the dependent's extension. If it finds such a rule, **MAKE** performs the commands given by the rule.

MAKE looks first for dependency rules in the current description file, but if it does not find an appropriate rule, it will search for the tools-initialization file, `tools.ini`. **MAKE** looks for `tools.ini` in the current drive and directory (or in any directories specified with the MS-DOS **PATH** command).

If **MAKE** finds `tools.ini`, it looks through the file for a line beginning with the tag `[make]`, which must come at the beginning of the line. Inference rules following this line will be applied if appropriate.

Example

```
.asm.obj:
        MASM $*.asm,,;

test1.obj: test1.asm

test2.obj: test2.asm
        MASM test2.asm;
```

In the sample description file above, an inference rule is defined in the first line. The file name in the rule is specified with the special macro name `$*`

so that the rule will apply to any base name. When **MAKE** encounters the dependency for files `test1.obj` and `test1.asm`, it looks first for commands on the next line. When it does not find any, **MAKE** checks for a rule that may apply and finds the rule defined in the first lines of the description file. **MAKE** applies the rule, replacing the `$*` macro with `test1` when it executes the command:

```
MASM test1.asm,,;
```

When **MAKE** reaches the second dependency for the `test2` files, it does not search for a dependency rule, since a command is explicitly stated for this target/dependent description.

7.3 Maintaining a Program: An Example

MAKE is especially useful for programs 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 the Microsoft Symbolic Debug Utility (**SYMDEB**).

The following target/dependent descriptions copied to the **MAKE** description file `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/dependent 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 have the following form:

```
MAKE test
```

MAKE carries out the following steps:

1. **MAKE** compares the modification date of `test.asm` with `test.obj`. If `test.obj` is out of date (or does not exist), **MAKE** executes the following 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 following command:

   ```
   CREF test,test
   ```

3. **MAKE** compares `test.exe` with the dates of `test.obj` and the library file `math.lib`. If `test.exe` is out of date with respect to either file, **MAKE** executes the following command:

   ```
   LINK test,test,test/map,\lib\math.lib
   ```

4. **MAKE** compares the dates of `test.sym` and `test.map`. If `test.sym` is out of date, **MAKE** executes the following command:

   ```
   MAPSYM /L test.map
   ```

When `test.asm` is first created, **MAKE** will execute all commands, since none of the target files exists. 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 **MAPSYM**, since `test.map` is created by **LINK**.

Appendixes

Appendix A
Error Messages

A.1 Introduction

This appendix lists and explains the error messages that can be generated by the programs in the Microsoft Macro Assembler package.

A.2 MASM Error Messages

This section lists and explains the messages displayed by the Microsoft Macro Assembler, **MASM**. The assembler displays a message whenever it encounters an error during processing. It displays a warning message whenever it encounters an instance of questionable statement syntax.

An end-of-assembly message is displayed at the end of processing, even if no errors occurred. The message tells how many bytes of symbol space are free and gives a count of the error and warning messages it displayed during the assembly. If the /V option is used, the number of source lines, the total number of lines (including macro expansions), and the number of symbols are also shown.

```
 1108 Source  Lines
 1286 Total   Lines
  215 Symbols

44814 Bytes symbol space free

    0 Warning Errors
    0 Severe  Errors
```

The first three lines of the message are only shown on the screen if the /V option is used. The entire message is copied to the end of the source listing, whether the /V option is used or not.

MASM error messages are listed in numerical order in this section with a short explanation where necessary. References to sections of the *Microsoft Macro Assembler User's Guide* (*User's Guide*) and sections of the *Microsoft Macro Assembler Reference Manual* (*Reference Manual*) are included where appropriate.

Code	Message
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 the line are received.
2	`Register already defined` This message indicates an internal error. If you get this message, notify Microsoft Corporation using the Software Problem Report at the end of the *Reference Manual.*

3	`Unknown symbol type`
	
	MASM does not recognize the size type specified in a label or external declaration. For example,
	
	`here   LABEL  bite`
	
	Rewrite with a valid type such as **BYTE**, **WORD**, **NEAR**, etc.

4	`Redefinition of symbol` If a symbol is defined in two places, this error occurs in Pass 1 on the second declaration of the symbol. See errors 5 and 26.
5	`Symbol is multi-defined` If a symbol is defined in two places, this error occurs in Pass 2 on each declaration of the symbol. See errors 4 and 26.
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 Pass 1 listing to aid in resolving phase errors between passes. See Sections 2.3.4 and 2.4.6.

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 being active.

9 `Symbol not defined`

A symbol is used without being defined. One potential source of this error is shown in Section 2.4.6.

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 else 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 the symbol `DEBUG` is not previously defined. See Section 7.2.1 in the *Reference Manual*.

14 `Symbol type usage illegal`

Illegal use of a **PUBLIC** symbol. See Section 6.2 of the *Reference Manual*.

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. For example, the following lines produce an error:

```
        DB      count DUP(?)
count   EQU     10
```

The statements would be legal if the lines were reversed.

18 `Must be register`

Register expected as operand, but you furnished a symbol.

19 `Wrong type of register`

Directive or instruction expected one type of register, but another was specified. For example, **INC CS**; you cannot increment the code segment.

20 `Must be segment or group`

Expecting segment or group, but something else was specified.

21 `Symbol has no segment`

Trying to use a variable with **SEG**, but the variable has no known segment.

22 `Must be symbol type`

Must have type **WORD**, **DW**, **QW**, **BYTE**, or similar designation, but received something else.

23 `Already defined locally`

Tried to define a symbol as **EXTRN** that had already been defined locally.

24 `Segment parameters are changed`

List of arguments to **SEGMENT** was not identical to the list the first time this segment was used.

25 `Not proper align/combine type`

SEGMENT parameters are incorrect. Check the align and combine types to make sure you have entered valid types from among those discussed in Section 3.4 of the *Reference Manual.*

26 `Reference to mult defined`

The instruction references a symbol that has been multi-defined. See errors 4 and 5.

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 division by 0 or a number larger than 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 ax,bh` is illegal; either both operands must be word or both must be byte. See Section 5.5 of the *Reference Manual.*

32 `Illegal use of external`

Use of an external in some illegal manner. For example,

```
DB      count DUP(?)
```

is illegal if `count` is declared external. See Section 6.3 of the *Reference Manual.*

33 `Must be record field name`

Expected 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. Often this error can be remedied by using the **PTR** operator to specify a size type.

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, *:symbol* is illegal; use *seg:symbol.*

39 `One operand must be const`

This is an illegal use of the addition operator. See Section 5.3.1 of the *Reference Manual.*

40 `Operands must be same or 1 abs`

Illegal use of the subtraction operator. See Section 5.3.1 in the *Reference Manual.*

41 `Normal type operand expected`

Received **STRUC**, **BYTE**, **WORD**, or some other invalid operand 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. See Section 7.2.5 in the *Reference Manual* for one example of how this can happen.

43 `Operand must have segment`

Illegal use of **SEG** directive. See Section 5.3.12 in the *Reference Manual* for valid use of the **SEG** operator.

44 `Must be associated with data`

Use of code-related item where data-related item was expected. For example:

```
here:  mov    ax,LENGTH ds:here
```

This line attempts to address an item through **DS** when the item is actually addressable to **CS**.

45 `Must be associated with code`

Use of data-related item where code-related item was expected. For example

```
       jmp    test
```

if the symbol `test` was declared in the data segment.

46 `Already have base register`

More than one base register was used in an operand. For example:

```
       mov    ax,[bx+bp]
```

47 `Already have index register`

More than one index register was used in an operand. For example:

```
       mov    ax,[si+di]
```

48 `Must be index or base register`

Instruction requires a base or index register and some other register was specified in square brackets ([]). For example:

```
       mov    ax,[bx+ax]
```

49 `Illegal use of register`

Use of a register with an instruction where no valid register is possible.

50 `Value is out of range`

Value is too large for expected use. For example,

```
mov    al,5000
```

is illegal; you must use a byte value for a byte register.

51 `Operand not in IP segment`

An operand cannot be accessed because it is not in the current **IP** segment.

52 `Improper operand type`

Use of an operand in a way that prevents opcode generation.

53 `Relative jump out of range`

Conditional jumps must be within the range -128 to +127 bytes of the current instruction, and the specific jump is beyond this range. You can usually correct the problem by reversing the condition of the conditional jump and using an unconditional jump (**JMP**) to the out-of-range label.

54 `Index displ. must be constant`

Illegal use of index displacement.

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 data were supplied as an operand for an instruction that cannot use immediate data. For example, the following statement is illegal:

```
mov    ds,data
```

You must move the segment address into a general register and then move it from that register to **DS**.

57 `Illegal size for item`

Size of referenced item is illegal. For example, shift of a doubleword. One example of an illegal size error is shown in Section 2.4.6. The error also frequently occurs when you try to assemble source code written for assemblers that have less strict type checking than the Microsoft Macro Assembler (such as early versions of the IBM assembler). Usually you can solve the problem by changing the size of the item with the **PTR** operator. See Section 5.5 of the *Reference Manual.*

58 `Byte register is illegal`

Use of one of the byte registers in context where it is illegal. For example, `PUSH AL` is illegal; use `PUSH AX`.

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 requires **AX** or **AL** as its right operand.

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`

Attempt 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 code segment defined with a different **ASSUME:CS**.

65 `Label can't have seg. override`

Illegal use of segment override. See Section 5.3.7 of the *Reference Manual* for examples of valid use of the segment override operator.

66 `Must have opcode after prefix`

Use of a **REPE**, **REPNE**, **REPZ**, or **REPNZ** instruction 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`

No **ASSUME** directive 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`

The **EVEN** directive was used, even though the segment was declared to be a byte segment. See Section 3.9 of the *Reference Manual.*

72 `Illegal value for DUP count`

The **DUP** count must be a constant that evaluates to a positive integer greater than zero.

73 `Symbol already external`

Attempt to define a symbol as local that is already external.

74 `DUP is too large for linker`

Nesting of **DUP** operators was such that too large a record was created for the linker. See Section 4.3.6 of the *Reference Manual.*

75 `Usage of ? (indeterminate) bad`

Improper use of the undefined operand (**?**). For example, `?+5` is illegal.

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 (**DB**, **DW**, etc.).

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, you tried to use a string such as `HELLO` for **DW** field when you should use **DB** for strings.

82 `Register can't be forward ref`

An attempt was made to forward reference a segment.

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 `End of file, no END directive`

You forgot an end statement or there is a nesting error.

86 `Data emitted with no segment`

Code that is not located within a segment attempted to generate data. An example is shown below:

```
code    SEGMENT
        .
        .
        .
code    ENDS
        push    ax
test    DW      ?
        END
```

Either of the two statements near the end of the sample would generate the error. Any statement that generates code or allocates data must be in a segment.

87 `Forced error - pass1`

You forced an error with the **.ERR1** directive.

88 `Forced error - pass2`

You forced an error with the **.ERR2** directive.

89 `Forced error`

You forced an error with the **.ERR** directive.

90 `Forced error - expression equals 0`

You forced an error with the **.ERRE** directive.

91 `Forced error - expression not equal 0`

You forced an error with the **.ERRNZ** directive.

92 `Forced error - symbol not defined`

You forced an error with the **.ERRNDEF** directive.

93 `Forced error - symbol defined`

You forced an error with the **.ERRDEF** directive.

94 `Forced error - string blank`

You forced an error with the **.ERRB** directive.

95 `Forced error - string not blank`

You forced an error with the **.ERRNB** directive.

96 `Forced error - strings identical`

You forced an error with the **.ERRIDN** directive.

97 `Forced error - strings different`

You forced an error with the **.ERRDIF** directive.

98 `Override value is wrong length`

The override value for a structure field is too large to fit in the field. An example is shown below:

```
x       STRUC
x1      DB      "A"
x       ENDS

y       x       <"AB">
```

The override value is a string consisting of two bytes, while the structure declaration only provided room for one.

99 `Line to long expanding` *symbol*

A symbol defined by an **EQU** or equal-sign (=) directive is so long that expanding it will cause the assembler's internal buffers to overflow. This message may indicate a recursive text macro.

100 `Impure memory reference`

The code contains an attempt to store data into the code segment when the **.826p** directive and the **/P** option are in effect. An example of storing code to the code segment is shown below:

```
code    SEGMENT
        ASSUME cs:code
c_word  DW      ?
        .
        .
        .
        mov     cs:c_word,data
code    ENDS
```

The **/P** option checks for such statements, which are acceptable in nonprotected mode, but can cause problems in protected mode.

101 `Missing data; zero assumed`

An operand is missing from a statement. For example:

```
mov     ax,
```

The code is assembled as if it were:

```
mov     ax,0
```

This is a warning error, and the object file is not deleted as it is with severe errors.

In addition to the numbered error messages listed above, **MASM** may generate the following unnumbered error messages:

`Out of Memory`

All available memory has been used, either because the source file is too long, or because there are too many symbols defined in the symbol table. There are several things you can do to resolve this problem. First, try assembling with only an object file. If this works, you can reassemble specifying a null object file to get a listing or cross-reference file. You can also rewrite the source file to take up less symbol space. Techniques for reducing symbol space include: minimizing use of macros, structures, and the **EQU** and equal-sign (=) directives; using short symbol names; using tab characters in macros rather than series of spaces; using macro comments (;;) rather than normal comments (;); purging macro definitions after the last use.

`Internal Error`

Note the conditions when the error occurs and contact Microsoft Corporation using the Software Problem Report at the end of the *Reference Manual.*

A.3 LINK Error Messages

This section lists the error messages that can occur when linking programs with the Microsoft 8086 Object Linker, **LINK**. The messages are in alphabetical order.

`Ambiguous switch error:` "*option*"

User did not enter a unique option name after the option indicator (/). For example, the command

```
LINK /N main;
```

will generate this error, since **LINK** can't tell which of the three options beginning with the letter "N" you intended to use. See Section 3.3 for more information on **LINK** options.

`Array element size mismatch`

A far communal array has been declared with two or more different array-element sizes (for example, declared once as an array of characters and once as an array of real numbers). This error cannot occur with object files produced by **MASM**. It only occurs with Microsoft C and any other compiler that supports far communal arrays.

`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** option is not in the range 1 to 65535. See Section 3.3.9.

`Cannot find library:` *filename*`.lib. Enter new file spec:`

The linker cannot find *filename*`.lib`. The user should respond to the prompt with a new file name, a new path specification, or both.

`Cannot open list file`

The disk or the root directory is full. Delete or move files to make space.

`Cannot open response file`

LINK cannot find the response file specified by the user. This usually indicates a typing mistake.

`Cannot nest response files`

User named a response file within a response file.

`Cannot open run file`

The disk or the root directory is full. Delete or move files to make space.

`Cannot open temporary file`

The disk or the root directory is full. Delete or move files to make space.

`Cannot reopen list file`

User did not actually replace the original disk when asked to. Restart the linker.

`Common area longer than 65536 bytes`

User's program has more than 64K of communal variables. This error cannot appear with object files generated by **MASM**. It can only occur with programs produced by Microsoft C or other compilers that support communal variables.

`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. Notify Microsoft Corporation using the Software Problem Report at the end of the *Reference Manual.* **LEDATA** is an MS-DOS term. It is explained in the *MS-DOS Programmer's Reference Manual* and some other MS-DOS reference books.

`Dup record too large`

LIDATA record (in an object module) contains more than 512 bytes of data. Most likely, an assembly module contains a structure definition that is very complex, or a series of deeply nested **DUP** operators. For example:

```
array DB 10 DUP(11 DUP (12 DUP (13 DUP (...))))
```

Simplify and reassemble. **LIDATA** is an MS-DOS term. It is explained in the *MS-DOS Programmer's Reference Manual* and in some other MS-DOS reference books.

filename `is not a valid library`

The file specified as a library file is invalid. **LINK** will abort.

`Fixup overflow near` *number* `in segment` *name* `in` *filename* `offset` *number*

Some possible causes are: 1) A group is larger than 64K; 2) the user's program contains an inter-segment short jump or inter-segment short call; 3) the user has a data item whose name conflicts with that of a subroutine in a library included in the link; or 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 versions of MS-DOS or PC-DOS prior to 2.0. Reboot your system with a valid version, 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 as a value for the **/OVERLAYINTERRUPT** option. Try again with a number in the range 0 to 255. See Section 3.3.13.

`Invalid numeric switch specification`

An incorrect value was entered for one of the linker switches (options). For example, a character string was entered for an option that requires a numeric value.

`Invalid object module`

One of the object modules is invalid. Try recompiling. If the error persists, contact Microsoft Corporation using the Software Problem Report form at the end of the *Reference Manual.*

`NEAR/HUGE conflict`

Conflicting near and huge definitions for a communal variable. This error cannot appear with object files generated by **MASM**. It can only occur with programs produced by Microsoft C or other compilers that support communal variables.

`Nested left parentheses`

User has made a typing mistake while specifying the contents of an overlay on the command line. See your compiler manual for instructions on specifying overlays for **LINK**. **MASM** does not have an overlay manager, so this problem can only occur if you are linking with a library from a high-level language that supports overlays.

`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** file 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 disk, 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 incorrect name and relink. See your compiler manual for instructions on specifying overlays for **LINK**. **MASM** does not have an overlay manager, so this problem can only occur if you are linking with a library from a high-level language that supports overlays.

`Relocation table overflow`

More than 32768 long calls, 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 off the debugging option.

`Segment limit set too high`

The limit on the number of segments allowed was set too high (over 1024) using the **/SEGMENTS** option. See Section 3.3.14.

`Segment limit too high`

There is insufficient memory for the linker to allocate tables to describe the number of segments requested (the default of 128 or the value specified with the **/SEGMENTS** option). Try linking again using the **/SEGMENTS** option to select a smaller number of segments (for example, 64 if the default was used previously), or free some memory by eliminating resident programs or shells.

`Segment size exceeds 64K`

User has a small-model program with more than 64K of code, or user has a middle-model program with more than 64K of data. Try compiling and linking middle- or large-model.

`Stack size exceeds 65536 bytes`

The size specified for the stack using the **/STACK** option is more than 65536 bytes. See Section 3.3.8.

`Symbol table overflow`

The user's program has more than 256K of symbolic information (publics, externals, 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 CONTROL-C.

`Too many external symbols in one module`

User's object module specified more than the limit of 1023 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 nine group definitions (**GRPDEF**s) in a single module. Reduce the number of **GRPDEF**s or split up the module. The term **GRPDEF** is explained in the *MS-DOS Programmer's Reference Manual* and in some other reference books on MS-DOS.

`Too many libraries`

User tried to link with more than 16 libraries. Combine libraries, or use modules that require fewer libraries.

`Too many overlays`

User's program defines more than 63 overlays. Reduce the number of overlays.

`Too many segments`

The user's program has more than the maximum number of segments as specified by the default of 128 or by the **SEGMENTS** option. Relink using the **/SEGMENTS** option with an appropriate number of segments. See Section 3.3.14.

`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 too many **TYPDEF** records. These records describe communal variables. This error cannot appear with object files generated by **MASM**. It can only occur with programs produced by Microsoft C or other compilers that support communal variables. **TYPDEF** is an MS-DOS term. It is explained in the *MS-DOS Programmer's Reference Manual* and in some other reference books on MS-DOS.

`Unexpected end-of-file on library`

The disk containing the library has probably been removed. Replace the disk with the library and try again.

`Unexpected end-of-file on scratch file`

Disk with **VM.TMP** was removed. See Section 3.2.6.

`Unmatched left parenthesis`

User has made a typing mistake while specifying the contents of an overlay on the command line. See your compiler manual for instructions on specifying overlays for **LINK**. **MASM** does not have an overlay manager, so this problem can only occur if you are linking with a library from a high-level language that supports overlays.

`Unmatched right parenthesis`

User has made a typing mistake while specifying the contents of an overlay on the command line. See your compiler manual for instructions on specifying overlays for **LINK**. **MASM** does not have an overlay manager, so this problem can only occur if you are linking with a library from a high-level language that supports overlays.

`Unrecognized switch error:` *option*

User entered an unrecognized character after the option indicator (/). For example:

```
LINK /ABCDEF main;
```

`Unresolved externals`

A symbol was declared external in one module, but it was not declared public in the module in which it was defined. A symbol must be defined and declared public (using the **PUBLIC** directive) in one and only one module before it can be used as an external symbol (using the **EXTRN** directive) by other modules.

`VM.TMP is an illegal file name and has been ignored`

User has specified **VM.TMP** as an object file name. Rename file and link again.

`Warning: no stack segment`

User's program contains no stack segment specified with **stack** combine type. Normally, every program should have a stack segment with the combine type specified as **stack**. You can ignore this message if you have a specific reason for not defining a stack or for defining one without the **stack** combine type.

`Warning: too many public symbols`

The **/MAP** option was used to request a sorted listing of public symbols in the map file, but there are too many symbols to sort. The linker will produce an unsorted listing of the public symbols.

A.4 SYMDEB Error Messages

The Microsoft Symbolic Debug Utility, **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 appears on the screen when you enter too many arguments for the Dump command (**D**).

```
D 0 1 2
      ^ Error
```

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

`Bad breakpoint number!`

You typed an invalid breakpoint number (the number must be in the range 0 to 9).

`Bad Flag!`

You attempted to alter a flag, but the characters typed were not among the acceptable pairs of flag values. See the Register command (**R**) in Section 4.3.5 for the list of acceptable flag entries.

`Breakpoint error!`

You typed **BP** without giving an address, or there are no more free breakpoints (all 10 have been set).

`Can't debug packed files!`

Files which have been packed with the **/EXEPACK** option of the linker, or with the **EXEPACK** program, cannot be debugged. See Section 3.3.3 for more information on the **/EXEPACK** option, or Section 8.1 for information on the **EXEPACK** utility.

`COMMAND.COM not found!`

You typed the Shell Escape command (!), but the shell cannot be created because **COMMAND.COM** was not found.

`No program to debug!`

You tried to redirect program I/O (input/output) when there was no program to debug.

`Not enough memory!`

You typed the Shell Escape character (!), but there is not enough free memory to execute **COMMAND.COM**. See Section 4.6.26.

`Too many breakpoints!`

You specified more than 10 breakpoints as parameters to the Go command (**G**). Retype the Go command with 10 or fewer breakpoints.

`Bad register!`

You typed the Register command (**R**) with an invalid register name. See the Register command (Section 4.6.22) 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 (**R**) in Section 4.6.22.

`Breakpoint list or '*' expected!`

You typed a Breakpoint Clear (**BC**), Breakpoint Disable (**BD**), or Breakpoint Enable (**BE**) command without giving a list of breakpoints to act on.

`Error reading .SYM file!`

The symbol file you requested in the **SYMDEB** command line cannot be read. The file may be empty, or a disk error may have occurred.

A.5 MAPSYM Error Messages

The Microsoft Symbol File Generator, **MAPSYM**, terminates operation and displays one of the following messages whenever it encounters an error:

`Can't create:` *mapname*

Can't create a symbol map for the file specified by *mapname*.

`Can't open MAP file:` *mapfile*

Usually indicates that the map file specified by *mapname* does not exist.

`mapsym: out of memory`

MAPSYM cannot find enough system memory to process the symbol map. Get rid of resident programs or add memory.

`mapsym: segment table (`*number*`) exceeded`

More than 1024 segments used in the map file. The *number* indicates the number of segments requested.

`No public symbols`
`Re-link with /M switch!`

You did not use the /M option when linking. This option must be specified in order to include public symbols in the map file.

`Unexpected eof reading:` *mapfile*

The specified *mapfile* is not in a valid format. This could mean that the file is corrupted. Try linking again to create a new map file.

`usage: MAPSYM [/l] maplist`

You entered the command line incorrectly. Re-enter the command with the syntax shown. The single brackets ([]) in the error message indicate that your choice of the item within them is optional.

`Write fail on:` *symbolfile*

The specified *symbolfile* cannot be written. The disk is full or some other file error occurred.

A.6 CREF Error Messages

The Microsoft Cross-Reference Utility, **CREF**, terminates operation and displays one of the following messages when it encounters an error:

`can't open cross-reference file for reading`

The **.CRF** file is not found. Make sure the file is on the specified disk and that the name is spelled correctly in the command line.

`can't open listing file for writing`

May indicate that the disk is full or write protected, that a file with the specified name already exists, or the specified device is not available.

`cref has no switches`

You specified an option in the command line with the slash (/) or dash (-) character, but **CREF** has no options.

`extra file name ignore`

You specified more than two files in the file name. **CREF** will create the reference file using only the first two files given.

`line invalid, start again`

No **.CRF** file was provided in the command line or at the prompt. **CREF** will display this message followed by a prompt asking for a **.CRF** file.

`out of heap space`

CREF cannot find enough memory to process the files. Try again with no resident programs or shells, or add more memory.

`premature eof`

You specified a file that is not a valid **.CRF** file, or the file is damaged.

`read error on stdio`

This error only occurs if the program receives a CONTROL-Z from the keyboard or from a redirected file.

A.7 LIB Error Messages

The following error messages may be displayed by the Microsoft Library Manager, **LIB**:

`cannot create extract file` *filename*

The disk or root directory is full, or the extract file specified by *filename* already exists with read-only protection. Make space on the disk or change the protection of the extract file.

`cannot create new library`

The disk or root directory is full, or the library file already exists with read-only protection. Make space on the disk or change the protection of the library file.

`cannot open response file`

The given response file was not found.

`cannot open VM.TMP`

The disk or root directory is full. Delete or move files to make space.

`cannot read from VM`

Note the circumstances of the failure and notify Microsoft Corporation using the Software Problem Report form at the end of the *Reference Manual*.

`cannot rename old library`

LIB could not rename the old library to have a **.BAK** extension because the **.BAK** version already existed with read-only protection. Change the protection on the old **.BAK** version.

`cannot reopen library`

The old library could not be reopened after it was renamed to have a **.BAK** extension.

`cannot write to VM`

Note the circumstances of the failure and notify Microsoft Corporation using the Software Problem Report form at the end of the *Reference Manual.*

`comma or newline expected`

A comma or carriage return was expected in the command line, but did not appear. This may indicate an inappropriately placed comma, as in the line:

```
LIB math.lib,-mod1+mod2;
```

The line should have been entered as:

```
LIB math.lib -mod1+mod2;
```

`error writing to cross reference file`

The disk or root directory is full. Delete or move files to make space.

`error writing to new library`

The disk or root directory is full. Delete or move files to make space.

`Free: not allocated`

Note the circumstances of the failure and notify Microsoft Corporation using the Software Problem Report form at the end of the *Reference Manual.*

`insufficient memory`

LIB does not have enough memory to run. Remove any shells or resident programs and try again, or add more memory.

`internal failure`

Note the circumstances of the failure and notify Microsoft Corporation using the Software Problem Report form at the end of the *Reference Manual.*

`invalid library`

The library does not conform to the format expected by **LIB**.

`Invalid object module` *name* `near` *location*
`in file` *libraryname*

The module specified by *name* is not a valid object module.

`Mark: not allocated`

Note the circumstances of the failure and notify Microsoft Corporation using the Software Problem Report form at the end of the *Reference Manual.*

`missing terminator`

The response to an `Output library:` prompt was not terminated by a carriage return.

`no more virtual memory`

Note the circumstances of the failure and notify Microsoft Corporation using the Software Problem Report form at the end of the *Reference Manual.*

`page size too small`

Page size specified with the **/PAGESIZE** option must be 16 or greater.

`too many symbols`

The maximum number of symbols allowed in a library file is 4609.

`syntax error`

The given command did not follow correct **LIB** syntax as specified in Chapter 6.

`syntax error (bad input)`

The given command did not follow correct **LIB** syntax as specified in Chapter 6.

`syntax error (bad file spec)`

A command operator such as a minus sign (-) was given without a following module name.

`syntax error (switch name expected)`

A forward slash (/) was given without the **PAGESIZE** option.

`syntax error (switch val expected)`

The **/PAGESIZE** option was given without a following value.

`unexpected EOF on command input`

An end-of-file character was received prematurely in response to a prompt.

`unknown switch`

An unknown option was given. The **/PAGESIZE** option is the only one currently recognized by **LIB**.

`write to extract file failed`

The disk or root directory is full. Delete or move files to make space.

`write to library file failed`

The disk or root directory is full. Delete or move files to make space.

A.8 MAKE Error Messages

Most error messages displayed by the Microsoft Program Maintenance Utility, **MAKE**, have the following form:

filename linenumber **:** *message*

The *filename* is the **MAKE** description file. The *linenumber* is the line where the error occurred. If an error occurs after **MAKE** has finished reading through the file, the *linenumber* will be listed as 1 even though this will not be the correct line number. The *message* is one of the error messages listed below:

`Exec not available on DOS 1.x`

MAKE requires MS-DOS or PC-DOS Version 2.0 or later.

`expansion too big`

A line with macros expands to longer than 512 bytes. Try rewriting the make file to use two short lines instead of one long one.

`line too long`

A line in the make file is longer than 128 characters. Try rewriting the make file to use two short lines instead of one long one.

`make:` *command* `-` *errorcode*

One of the programs or commands called in the make file was not able to execute correctly. **MAKE** terminates and displays the command followed by the code of the error that caused it to fail. Error codes are described in Appendix B of this *User's Guide.*

`make: colon missing in` *filename*

A line that should be a target-dependent line lacks a colon indicating the separation between target and dependent. **MAKE** expects any line following a blank line to be a target-dependent line.

`make: dependent '`*filename*`' does not exist, target '`*filename*`' not built`

MAKE could not continue because a required dependent file did not exist. Make sure all named files are present and that they are spelled correctly in the **MAKE** description file.

`make: infinitely recursive macro`

A circular chain of macros was defined. For example:

```
A=$(B)
B=$(C)
C=$(A)
```

`make: multiple source`

An inference ruler has been defined more than once.

`make: out of memory`

MAKE has run out of memory for processing the make file. Try to reduce the size of the make file by reorganizing or splitting it.

`make: out of space`

MAKE has run out of memory for processing the make file. Try to reduce the size of the make file by reorganizing or splitting it.

`make: syntax error`

The make file has a line beginning with an equal sign (=).

`make: target does not exist '`*filename*`'`

This usually does not indicate an error. It warns the user that the target file did not exist. **MAKE** executes any commands given in the target/dependent description since in many cases the target file will be created by a later command in the **MAKE** description file.

`Stack overflow`

Recursive macros have used up all available memory. Reduce the number or levels of nested macros.

`usage: make [/n] [/d] [/i] [/s] [name=value ...] file`

MAKE has not been invoked correctly. Try entering the command line again with the syntax shown in the message.

A.9 EXEPACK Error Messages

The Microsoft EXE File Compression Utility, **EXEPACK**, generates the following error messages:

`exepack: can't change load-high program`

When the minimum allocation value and the maximum allocation value are both zero, the file cannot be compressed.

`exepack: error reading relocation table`

The file cannot be compressed because the relocation table cannot be found or is invalid.

`exepack: invalid .EXE file (actual length < reported)`

The second and third fields in the file header indicate a file size greater than the actual size.

`exepack: invalid .EXE file (bad header)`

The given file is not an executable file or has an invalid file header.

`exepack:` *filename*`: No such file or directory`

The file specified by *filename* cannot be found.

`exepack:` *filename*`: Permission denied`

The file specified by *filename* is a read-only file.

`exepack: out of memory`

The **EXEPACK** utility does not have enough memory to operate.

`Out of space on output file`

The disk or root directory is full. Delete or move files to make space.

`exepack: too many segments in relocation table`

The given file is too large to be compressed in the available system memory.

`usage: exepack <infile> <outfile>`

The **EXEPACK** command line was not specified properly. Try again using the syntax shown.

You may also encounter MS-DOS error messages if the **EXEPACK** program cannot read from, write to, or create a file.

A.10 EXEMOD Error Messages

The Microsoft EXE File Header Utility, **EXEMOD**, generates the following error messages:

`exemod: can't change load-high program`

When the minimum allocation value and the maximum allocation value are both zero, the file cannot be modified.

`exemod: file not .EXE`

EXEMOD automatically appends the **.EXE** extension to any file name without an extension; in this case, no file with the given name and an **.EXE** extension could be found.

`exemod: invalid .EXE file (actual length < reported)`

The second and third fields in the file header indicate a file size greater than the actual size.

`exemod: invalid .EXE file (bad header)`

The specified file is not an executable file or has an invalid file header.

`exemod: min > max (correcting max)`

If the minimum allocation value is greater than the maximum allocation value, the maximum allocation value is adjusted. This is a warning message only; the modification is still performed.

`exemod: min < stack (correcting min)`

If the minimum allocation value is not enough to accommodate the stack (either the original stack request or the modified request), the minimum allocation value is adjusted. This is a warning message only; the modification is still performed.

exemod: *filename*: No such file or directory

The file specified by *filename* cannot be found.

exemod: *filename*: Permission denied

The file specified by *filename* is a read-only file.

`exemod: (warning) packed file`

The given file is a packed file. This is a warning only. **EXEMOD** will still modify the file. The values shown if you ask for a display of MS-DOS header values will be the values after the packed file is expanded.

`usage:exemod file [-/h] [-/stack n] [-/max n] [-/min n]`

The **EXEMOD** command line was not specified properly. Try again using the syntax shown. Note that the option indicator can be either a slash (/) or a dash (–). The single brackets ([ ]) in the error message indicate that your choice of the item within them is optional.

The **EXEMOD** utility also produces error messages when the file header is not in recognizable **.EXE** format, or if errors occur in reading from, or writing to, a file.

Appendix B
Exit Codes

B.1 Introduction

All the programs in the Microsoft Macro Assembler package return a code (sometimes called an "errorlevel" code) that can be used by MS-DOS batch files or other programs such as **MAKE**. If the program finishes without errors, it returns a code of 0. The code returned varies if the program encounters an error. This appendix lists the numbers returned when a program encounters an error.

B.2 Exit Codes with Make

MAKE automatically stops execution if a program executed by one of the commands in the **MAKE** description file encounters an error. The exit code is displayed as part of the error message.

For example, assume the **MAKE** description file `test` contains the following lines:

```
test.obj :     test.asm
      MASM test;
```

If the source code in `test.asm` contains an assembly error, you would see this message the first time you use **MAKE** with the file `test`:

```
make: MASM test; - error 7
```

This error message indicates that the command `MASM test;` in the **MAKE** description file returned code 7.

B.3 Exit Codes with MS-DOS Batch Files

If you prefer to use MS-DOS batch files instead of **MAKE**, you can test the code returned with the **IF ERRORLEVEL** command. The sample batch file below, called `ASMBL.BAT`, illustrates how:

```
MASM %1;
IF NOT ERRORLEVEL 1 LINK %1;
IF NOT ERRORLEVEL 1 %1
```

If you execute this sample batch file with the command `ASMBL test`, MS-DOS first executes the command `MASM test;` and returns a code of 0 if **MASM** is successful, or a higher code if **MASM** encounters an error. In the second line, MS-DOS tests to see if the code returned by the previous line is 1 or higher. If it is not (that is, if the code is 0), MS-DOS executes the command `LINK test;` and again returns a code which will be tested by the third line.

B.4 Exit Codes for Programs in the Macro Assembler Package

An exit code of 0 always indicates execution of the program with no fatal errors. Warning errors also return exit code 0. Some programs can return various codes indicating different kinds of errors, while other programs return only 1 to indicate that an error occurred. The exit codes for each program are listed in Sections B.4.1–B.4.9.

B.4.1 MASM Exit Codes

Code	**Meaning**
0	No error
1	Argument error
2	Unable to open input file
3	Unable to open listing file
4	Unable to open object file
5	Unable to open cross-reference file
6	Unable to open include file
7	Assembly error
8	Memory allocation error
10	Error defining symbol from command line
11	User interrupted

Note that if the exit code is 7, **MASM** automatically deletes the invalid object file.

B.4.2 LINK Exit Codes

Code	Meaning
0	No error
1	All **LINK** fatal errors not listed below
16	Data record too large
32	No object modules specified
33	Cannot open list file
66	Common area longer than 65536 bytes
96	Too many libraries
144	Invalid object module
145	Too many **TYPDEF**s
146	Too many group-, segment-, and/or class-names in one module
147	Too many segments, or too many segments in one module
148	Too many overlays
149	Segment size exceeds 64K
150	Too many groups or too many **GRPDEF**s in one module
151	Too many external symbols in one module
177	Group larger than 64K

B.4.3 SYMDEB Exit Codes

SYMDEB does not return exit codes. However, it does display return codes returned by programs run within **SYMDEB**. For example, if you run **LINK** from within **SYMDEB** and it encounters an error that returns 1, you will see the following line:

```
Program terminated normally (1)
```

B.4.4 MAPSYM Exit Codes

Code	Meaning
0	No error
1	Write failure, can't create symbol file, or no such map file.
4	Unexpected end-of-file (usually invalid map file), out of memory, too many segments, or no public symbols.

B.4.5 CREF Exit Codes

Code	Meaning
0	No error
1	Any **CREF** fatal error

B.4.6 LIB Exit Codes

Code	Meaning
0	No error
1	All **LIB** fatal errors not listed below
4	Internal error
13	Too many symbols
16	Page size too small

B.4.7 MAKE Exit Codes

Code	Meaning
0	No error
1	Any **MAKE** fatal error

If a program called by a command in the **MAKE** description file produces an error, the exit code will be displayed in the **MAKE** error message.

B.4.8 EXEPACK Exit Codes

Code	Meaning
0	No error
1	Any **EXEPACK** fatal error

B.4.9 EXEMOD Exit Codes

Code	Meaning
0	No error
1	Any **EXEMOD** fatal error

Appendix C
Using EXEPACK and EXEMOD

C.1 Introduction

The Microsoft EXE File Compression Utility, **EXEPACK**, and the Microsoft EXE File Header Utility **EXEMOD**, supplied with the Microsoft Macro Assembler package, allow you to modify executable program files.

EXEPACK compresses executable files by removing sequences of repeated characters from the file and by optimizing the relocation table. **EXEMOD** allows you to examine and modify file header information. The following sections explain how to use the **EXEPACK** and **EXEMOD** programs.

C.2 The EXEPACK Utility

EXEPACK compresses sequences of identical characters from a specified executable file and optimizes the relocation table. Using **EXEPACK**, you can significantly reduce the size of some files and decrease the time required to load them.

EXEPACK will not always give a significant savings in disk space (and may sometimes actually increase file size). Programs that have a large number of load-time relocations (about 500 or more) and long streams of repeated characters will usually be shorter if packed.

The **EXEPACK** program has exactly the same function as the **LINK /EXEPACK** option except that **EXEPACK** works on files that have already been linked. One use for this utility is to pack the files provided with the Microsoft Macro Assembler package. The savings in disk space is insignificant for most of these programs, but the size of **MAPSYM.EXE** can be reduced significantly.

The **EXEPACK** command line format is:

EXEPACK *executablefile packedfile*

The *executablefile* is the file to be packed and *packedfile* is the name for the packed file. The *packedfile* should have a different name or be on a different disk since **EXEPACK** will not pack a file onto itself.

Do not try to get around the limitation against packing a file onto itself by specifying the same file in a different way. You may be able to fool **EXE-PACK**, but the result will be a damaged file. If you want the packed file to replace the original, you should use a separate name for the packed file, then delete the original and rename the packed copy.

When using **EXEPACK** to pack an executable overlay file or a file that calls overlays, the packed file should be always be renamed back to the original name.

C.3 The EXEMOD Utility

EXEMOD modifies fields in the MS-DOS file header. In order to use this utility, you need to understand the MS-DOS conventions for file headers. They are explained in the *Microsoft MS-DOS Programmer's Reference Manual* and in some other reference books on MS-DOS.

Some of the options available with **EXEMOD** are the same as **LINK** options except that they work on files that have already been linked. Unlike the **LINK** options, the **EXEMOD** options require that values be given in hexadecimal.

To display the current status of the header fields, type:

EXEMOD *executablefile*

To modify one or more of the fields in the file header, type:

EXEMOD *executablefile* ⟦/H⟧ ¦ ⟦/STACK *number*⟧ ⟦/MIN *number*⟧ ⟦/MAX *number*⟧

EXEMOD expects the *executablefile* to be the name of an existing file with the **.EXE** extension. If the filename is given without an extension, **EXE-MOD** appends **.EXE** and searches for that file. If you supply a file with an extension other than **.EXE**, **EXEMOD** displays an error message.

The options in examples are shown with the forward slash (/) option designator, but a dash (–) may also be used. Options can be given in either upper- or lowercase, but they cannot be abbreviated. The options and their effects are described in the following list:

Option	Effect
/**STACK** *number*	Sets the initial **SP** (stack pointer) value to *number*, where *number* is a hexadecimal value setting the number of bytes. The minimum allocation value is adjusted upward, if necessary. This option has the same effect as the **LINK** /**STACK** option.
/**MIN** *number*	Sets the minimum allocation value to *number*, where *number* is a hexadecimal value setting the number of paragraphs. The actual value set may be different from the requested value if adjustments are necessary to accommodate the stack.
/**MAX** *number*	Sets the maximum allocation to *number*, where *number* is a hexadecimal value setting the number of paragraphs. The maximum allocation value must be greater than, or equal to, the minimum allocation value. This option has the same effect as the **LINK** /**CPARMAXALLOC** option.
/**H**	This option displays the current status of the MS-DOS program header. Its effect is the same as entering **EXEMOD** with an *executablefile*, but no options. The /**H** option should not be used with other options.

Note

The /**STACK** option can be used on programs assembled with **MASM** or programs compiled with the Microsoft C Compiler Version 3.0 or later, the Microsoft Pascal Compiler Version 3.3 or later, or the Microsoft FORTRAN Compiler Version 3.3 or later. Use of the /**STACK** option on programs developed with other compilers may cause the programs to fail, or **EXEMOD** may return an error message.

EXEMOD works on packed files. When it recognizes a packed file, it will print the following message:

```
exemod: (warning) packed file
```

It will then continue to modify the file header.

When packed files are loaded, they are expanded to their unpacked state in memory. If the **EXEMOD /STACK** option is used on a packed file, the value changed is the value that **SP** will have after expansion. If either the **/MIN** or **/STACK** option is used, the value will be corrected as necessary to accommodate unpacking of the modified stack. The **/MAX** option operates as it would for unpacked files.

If the header of a packed file is displayed, the **CS:IP** and **SS:SP** values are displayed as they will be after expansion, which is not the same as the actual values in the header of the packed file.

Examples

```
EXEMOD test.exe
test.exe                                   (hex)          (dec)

Minimum load size (bytes)                   419D          16797
Overlay number                                 0              0
Initial CS:IP                          0403:0000
Initial SS:SP                          0000:0000              0
Minimum allocation (para)                      0              0
Maximum allocation (para)                   FFFF          65535
Header size (para)                            20             32
Relocation table offset                       1E             30
Relocation entries                             1              1
```

The first example shows the file header for file `test.exe`. The following command line shows how to modify the header:

```
EXEMOD test.exe /STACK FF /MIN FF /MAX FFF
```

The second example shows a display of values after the modification:

```
EXEMOD test.exe
test.exe                                   (hex)          (dec)

Minimum load size (bytes)                   528D          20877
Overlay number                                 0              0
Initial CS:IP                          0403:0000
Initial SS:SP                          0000:00FF            256
Minimum allocation (para)                     FF            256
Maximum allocation (para)                    FFF           4095
Header size (para)                            20             32
Relocation table offset                       1E             30
Relocation entries                             1              1
```

Index (User's Guide)

Microsoft® Macro Assembler

for the MS-DOS® Operating System

Reference Manual

Microsoft Corporation

If you have comments about the software, complete the Software Problem Report at the back of this manual and return it to Microsoft Corporation.

If you have comments about the software documentation, complete the Documentation Feedback reply card at the back of this manual and return it to Microsoft Corporation.

Document Number 410610002-400-R00-1285

Contents

Figures

Tables

Chapter 1
Introduction

1.1 Overview

This reference manual describes the syntax and structure of assembly language for **MASM**, the Microsoft® Macro Assembler. **MASM** is an assembler for the Intel® 8086/80186/80286 family of microprocessors. It can assemble the instructions of the 8086, 8088, 80186, and 80286 microprocessors, and the 8087 and 80287 floating-point coprocessors. It has a powerful set of assembly-language directives that gives you complete control of the segmented architecture of the 8086, 80186, and 80286 microprocessors. **MASM** instruction syntax allows a wide variety of operand data types, including integers, strings, packed decimals, floating-point numbers, structures, and records.

The assembler produces 8086, 8088, 80186, or 80286 relocatable object modules from assembly-language source files. The relocatable object modules can be linked, using **LINK**, the Microsoft 8086 Object Linker, to create executable programs for the MS-DOS® operating system.

MASM is a macro assembler. It has a full set of macro directives that let you create and use macros in a source file. The directives instruct **MASM** to repeat common blocks of statements, or replace macro names with the blocks of statements they represent. **MASM** also has conditional directives that provide for selective exclusion of portions of a source file from assembly, or inclusion of additional program statements by simply defining a symbol.

MASM carries out strict syntax checking of all instruction statements, including strong typing for memory operands, and detects questionable operand usage that could lead to errors or unwanted results.

MASM produces object modules compatible with object modules created by many high-level-language compilers. Thus, programs can be constructed by combining **MASM** object modules with object modules created by C, Pascal, FORTRAN, or other language compilers.

1.2 About This Manual

This reference manual supplements the *Microsoft Macro Assembler User's Guide,* which explains program operation and the steps required to create executable programs from source files.

This reference manual does not teach assembly-language programming, nor does it give detailed descriptions of the 8086, 80186, and 80286 instruction sets. For further information on these topics, other references are available. Some of these are listed in the introduction to the *Microsoft Macro Assembler User's Guide.*

Chapter 1 concludes with an explanation of notational conventions used throughout the *Microsoft Macro Assembler Reference Manual.* Chapter 2 discusses the elements of the assembler, reserved words, characters that can be used in a program, and how to form numbers, names, statements and comments compatible with the assembler. Chapter 3 details the program-structure directives, which allow definition of code and data organization, and the instruction-set directives used for specifying which instruction set or sets will be used during assembly. Chapter 4 explains generating data for programs, declaration of labels, variables and other symbols, and type definition for data blocks. Chapter 5 deals with combining operators and operands into expressions for assembly-language statements and directives. Chapter 6 covers the global-declaration directives that allow transformation of local symbols into global symbols available to all program modules. Chapters 7 and 8 discuss the uses of, and relationship between, conditional-assembly directives and macro directives. Chapter 9 explains the file-control directives and how to use them to control source files and the files read and created by **MASM** during assembly.

Appendix A provides a list of the instruction names and syntax for the 8086/80186/80286 family of processors. For quick reference, the Microsoft Macro Assembler package also includes a copy of Intel Corporation's *8086/8088/8087/80186/80188 Programmer's Pocket Reference Guide.* Appendix B lists the directives you can use in **MASM** source files, while Appendix C gives some guidance on linking **MASM** object files to object files from high-level-language compilers.

1.3 Notational Conventions

This manual uses the following notational conventions in defining assembly-language syntax, and in presenting examples:

Convention	Meaning
Bold type	Bold type indicates commands, parameter names, or symbols that must be typed as shown. In most cases, upper- and lowercase letters can be freely intermixed. One exception is text within double

quotation marks ("*text*"). Text in quotation marks is usually case-sensitive.

Examples

[*displacement*] [**DI**]
[**DI**+*displacement*]
[**DI**].*displacement*
[**DI**]+*displacement*

Note that in the examples above, the brackets must be typed as shown. The register name **DI** must also be typed as shown, though you could use lowercase letters. The plus sign (+) in the second and fourth examples, and the period (.) in the third example must be typed as shown.

Italics

Italics indicate a placeholder: a name that you must replace with the value or file name required by the program.

Example

/**I***path*

In the example above, the slash (/) and the letter **I** must be entered as shown (except that the **I** could be lowercase). However, *path* is a placeholder representing a path name supplied by the user. You could enter any path name such as `B:\` or `\MASM\PROJECT1`. When a placeholder is used in a syntax example at the start of a section, the text below usually describes the types of values that can replace the placeholder.

⟦ ⟧

Double brackets indicate that the enclosed item is optional. Don't confuse double brackets with single brackets ([]), which must be typed as shown.

Example

BP ⟦*number*⟧ *address* ⟦*passcount*⟧ ⟦"*commands*"⟧

In the example, above, you must enter **BP** as shown. You must also enter a value for the *address* placeholder. Values for the placeholders *number*, *passcount*, and *commands* can be entered if you wish, or they can be left blank. If you enter a value for *commands*, it must be enclosed in quotation marks ("").

,,,

A series of commas indicates that you can repeat the preceding item type if you separate each of the items with commas.

Example

⟦*name*⟧ *recordname* <⟦*initialvalue,,,*⟧>

In the example above, you may provide a *name* and you must provide a *recordname*. You may provide more than one *initialvalue* as long as you separate the values with commas. Note that you must type the angle brackets even if you do not provide any *initialvalue*.

¦

A vertical bar between items indicates that only one of the separated items can be used. You must make a choice between the items.

Example

D ⟦*address* ¦ *range*⟧

In the example above, you must enter the letter **D**. You may enter either an *address* or a *range* (but not both).

`Special typeface for examples`

Example text in this manual is shown in a special typeface so that it will look more like listings on the screen or listings produced with a printer.

Examples that represent source code follow these conventions:

- Lowercase for symbols, labels, instructions, and registers
- Uppercase for reserved words
- Uppercase for hexadecimal digits
- Lowercase for radix indicators
- Upper- and lowercase for comments

These are conventions, not requirements. Your source code can use any combination of upper- and lowercase letters, though your code will be clearer if you choose a convention and use it consistently.

Examples

```
count   DB      0
        mov     ax,bx
print   PROC    near
```

Chapter 2
Elements of the Assembler

2.1 Introduction

All assembly-language programs consist of one or more statements and comments. A statement or comment is a combination of characters, numbers, and names. Names and numbers are used to identify values in instruction statements. Characters are used to form the names or numbers, or to form character constants.

Section 2.2 lists the characters that can be used in a program and Sections 2.3–2.12 describe how to form numbers, names, statements, and comments.

2.2 Character Set

MASM recognizes the following character set:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z

0 1 2 3 4 5 6 7 8 9

? @ _ $: . [ ] ( ) < > { }

+ - / * & % ! ' ~ ¦ \ = # ^ ; , ` "

2.3 Integers

Syntax

digits
*digits*B
*digits*Q
*digits*O
*digits*D
*digits*H
*digits*R

An integer is an integer number: a combination of binary, octal, decimal, or hexadecimal *digits* plus an optional radix. The *digits* are combinations of one or more digits of the specified radix: **B**, **Q**, **O**, **D**, or **H**. The real number designator **R** can also be used. If no radix is given, the assembler uses the current default radix (decimal, unless you have changed it with the **.RADIX** directive). The radix specifier can be either upper- or lowercase; sample code in this manual uses lowercase. Table 2.1 lists the digits that can be used with each radix.

Table 2.1

Digits Used with Each Radix

Radix	Type	Digits
B	Binary	0 1
Q or **O**	Octal	0 1 2 3 4 5 6 7
D	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
R	Real Number	0 1 2 3 4 5 6 7 8 9 A B C D E F

Hexadecimal numbers must always start with a decimal digit (0 to 9). If necessary, put a leading 0 at the left of the number to distinguish between hexadecimal numbers that start with a letter, and symbols. For example, `0ABCh` is interpreted as a hexadecimal number, but `ABCh` is interpreted as a symbol. The hexadecimal digits A through F can be either upper- or lowercase. Sample code in this manual uses uppercase.

The real number designator (**R**) can only be used with hexadecimal numbers consisting of 8, 16, or 20 significant digits (a leading 0 can be added).

The maximum number of digits in an integer depends on the instruction or directive in which the integer is used. The default radix can be specified by using the **.RADIX** directive (see Section 9.3).

Examples

```
01011010b      132q      5Ah      90d
01111b         17o       0Fh      15d
```

2.4 Real Numbers

Syntax

⟦+¦–⟧ *digits.digits*⟦**E**⟦+¦–⟧*digits*⟧

A real number is a number consisting of an integer, a fraction, and an exponent. The *digits* can be any combination of decimal digits. Digits before the decimal point (**.**) represent the integer. Those following the point represent the fraction. The digits after the exponent mark (**E**) represent the exponent, which is optional. If an exponent is given, a plus (+) or minus (–) sign may be used to indicate its sign.

Real numbers can be used only with the **DD**, **DQ**, and **DT** directives. The maximum number of digits in the number and the maximum range of exponent values depend on the directive. See Sections 4.3.3, 4.3.4, and 4.3.5 in this reference manual.

Examples

```
25.23
2.523E1
2523.0E-2
```

2.5 Encoded Real Numbers

Syntax

*digits***R**

An encoded real number is an 8-, 16-, or 20-digit hexadecimal number that represents a real number in encoded format. An encoded real number has a sign, a biased exponent, and a mantissa. These values are encoded as bit fields within the number. The exact size and meaning of each bit field depends on the number of bits in the number. The *digits* must be hexadecimal digits. The number must begin with a decimal digit (0-9) and must be followed by the real number designator (**R**).

Encoded real numbers can be used only with the **DD**, **DQ**, and **DT** directives. The number of digits for the encoded numbers used with **DD**, **DQ**, and **DT** must be 8, 16, and 20 digits, respectively. (If a leading 0 is supplied, the number must be 9, 17, or 21 digits.) See Sections 4.3.3, 4.3.4, and 4.3.5.

Examples

```
DD      3F800000r                 ; 1.0 for DD
DQ      3FF0000000000000r         ; 1.0 for DQ
```

2.6 Packed Decimal Numbers

Syntax

[[+|–]]*digits*

A packed decimal number represents a decimal integer to be stored in packed decimal format. Packed decimal storage has a leading-sign byte and 9 value bytes. Each value byte contains two decimal digits. The high-order bit of the sign byte is 0 for positive values, and 1 for negative values.

Packed decimals have the same format as other decimal integers, except that they can take an optional plus (+) or minus (–) sign and can be defined only with the **DT** directive. A packed decimal must not have more than 18 digits.

Examples

```
DT      1234567890       ; Encoded as 00000000001234567890h
DT      -1234567890      ; Encoded as 80000000001234567890h
```

2.7 Character and String Constants

Syntax

'*characters*'
"*characters*"

A character constant consists of a single ASCII (American Standard Code for Information Interchange) character. A string constant consists of two or more ASCII characters. Constants must be enclosed in right single quotation marks or double quotation marks. String constants are case-sensitive.

Single quotation marks must be encoded twice when used literally within constants that are also enclosed by single quotation marks. Similarly, double quotation marks must be encoded twice when used in constants that are also enclosed within double quotation marks.

Examples

```
'a'
'ab'
"a"
"This is a message."
'Can''t find file.'              ; Can't find file.
"Can't find file."               ; Can't find file.
"This ""value"" not found."      ; This "value" not found.
'This "value" not found.'        ; This "value" not found.
```

2.8 Names

Syntax

characters

A name is a combination of letters, digits, and special characters used as a label, variable, or symbol in an assembly-language statement. Names have the following formatting rules:

- A name must begin with a letter, an underscore (_), a question mark (?), a dollar sign ($), or an at sign (@).
- A name can have any combination of upper- and lowercase letters. All lowercase letters are converted to uppercase by the assembler, unless the /ML option is used during assembly, or unless the name is declared with a **PUBLIC** or **EXTRN** directive and the /MX option is used during assembly.
- A name can have any number of characters, but only the first 31 characters are used. All other characters are ignored.

Examples

```
subrout3
Array
_main
```

2.9 Reserved Names

A reserved name is any name with a special, predefined meaning to the assembler. Reserved names include instruction and directive mnemonics, register names, and operator names. These names can be used only as defined and must not be redefined.

All upper- and lowercase combinations of these names are treated as the same name. For example, the names `Length` and `LENGTH` are the same name for the **LENGTH** operator.

Table 2.2 lists all reserved names except instruction mnemonics. For a complete list of instruction mnemonics, see Appendix A.

Table 2.2

Reserved Names

.186	DI	.ERRNZ	LENGTH	.SALL
.286c	DL	ES	.LFCOND	SEG
.286p	DQ	EVEN	.LIST	SEGMENT
.287	DS	EXITM	LOCAL	.SFCOND
.8086	DT	EXTRN	LOW	SHL
.8087	DW	FAR	LT	SHORT
=	DWORD	GE	MACRO	SHR
AH	DX	GROUP	MASK	SI
AL	ELSE	GT	MOD	SIZE
AND	END	HIGH	NAME	SP
ASSUME	ENDIF	IF	NE	SS
AX	ENDM	IF1	NEAR	STRUC
BH	ENDP	IF2	NOT	SUBTTL
BL	ENDS	IFB	OFFSET	TBYTE
BP	EQ	IFDEF	OR	.TFCOND
BX	EQU	IFDIF	ORG	THIS
BYTE	.ERR	IFE	%OUT	TITLE
CH	.ERR1	IFIDN	PAGE	TYPE
CL	.ERR2	IFNB	PROC	.TYPE
COMMENT	.ERRB	IFNDEF	PTR	WIDTH
.CREF	.ERRDEF	INCLUDE	PUBLIC	WORD
CS	.ERRDIF	IRP	PURGE	.XALL
CX	.ERRE	IRPC	QWORD	.XCREF
DB	.ERRIDN	LABEL	.RADIX	.XLIST
DD	.ERRNB	.LALL	RECORD	XOR
DH	.ERRNDEF	LE	REPT	

2.10 Statements

Syntax

⟦*name*⟧ *mnemonic* ⟦*operands*⟧ ⟦;*comment*⟧

A statement is a combination of an optional *name*, a mandatory instruction or directive *mnemonic*, one or more optional *operands*, and an optional *comment*. A statement represents an action to be taken by the assembler, such as generating a machine instruction or generating 1 or more bytes of data.

Statements are formed according to the following rules:

- A statement can begin in any column.
- A statement must not have more than 128 characters and must not contain an embedded carriage-return/line-feed combination. In other words, continuing a statement on multiple lines is not allowed.
- All statements except the last one in the file must be terminated by a carriage-return/line-feed combination.

Examples

```
count   DB      0
        mov     ax,bx
        ASSUME  cs:_text,ds:DGROUP
print   PROC    near
```

2.11 Comments

Syntax

; *text*

A comment is any combination of characters preceded by a semicolon (;) and terminated by an embedded carriage-return/line-feed combination. Comments describe the action of a program at the given point, but are otherwise ignored by the assembler and have no effect on assembly.

Comments can be placed anywhere in a program, even on the same line as a statement. However, if the comment shares the line with a statement, it must be to the right of all names, mnemonics and operands. A comment following a semicolon must not continue past the end of the line on which it begins; that is, it must not contain any embedded carriage-return/line-feed combination characters. For very long comments, the **COMMENT** directive can be used.

Examples

```
; This comment is alone on a line.
        mov      ax, bx   ; This comment follows a statement.
; Comments can contain reserved words like PUBLIC.
```

2.12 COMMENT Directive

Syntax

COMMENT *delimiter*
text
delimiter ⟦*text*⟧

The **COMMENT** directive causes the assembler to treat all *text* between *delimiter* and *delimiter* as a comment. The *delimiter* character must be the first nonblank character after the **COMMENT** keyword. The text is all remaining characters up to the next occurrence of the delimiter. The text must not contain the delimiter character.

The **COMMENT** directive is typically used for multiple-line comments. Although text can appear anywhere on the same line as the last *delimiter*, all text on the same line as the last *delimiter* is ignored by the assembler.

Examples

```
comment *
This comment continues until the
next asterisk.
*
```

The preceding and following examples illustrate how blocks of text can be designated as comments.

```
comment +
The assembler ignores the statement
following the last delimiter
+ mov   ax, 1
```

Chapter 3
Program Structure

3.1 Introduction

The program-structure directives let you define the organization that a program's code and data will have when loaded into memory. The program-structure directives include the following:

Directive	Meaning
SEGMENT	Segment definition
ENDS	Segment end
END	Source-file end
GROUP	Segment groups
ASSUME	Segment registers
ORG	Segment origin
EVEN	Segment alignment
PROC	Procedure definition
ENDP	Procedure end

Section 3.2 and Sections 3.4–3.10 describe these directives in detail. Section 3.3 describes the instruction-set directives, which let you specify the instruction set or sets to be used during assembly.

3.2 Source Files

Every assembly-language program is created from one or more "source" files: text files that contain statements defining the program's data and instructions. MASM reads source files and assembles the statements to create object modules. LINK, the Microsoft 8086 Object Linker, can then be used to prepare these object modules for execution.

Source files must be in standard ASCII format: they must not contain control codes, and each line must be separated by a carriage-return/line-feed combination. Statements can be entered in upper- or lowercase. Sample code in this manual uses uppercase letters for MASM reserved words and for class types, but this is a convention, not a requirement.

All source files have the same form: zero or more program segments followed by an **END** directive (a source file containing only macros, structures, or records might have zero segments). The **END** directive, required in every source file, signals the end of the source file. The **END** directive also provides a way to define the program entry point or starting address (if any).

The following example illustrates the source-file format. It is a complete assembly-language program that uses MS-DOS functions (or system calls) to print the message `Hello world` on the screen.

Example

```
data    SEGMENT                  ; Program Data Segment
string  DB     "Hello world",13,10,"$"
data    ENDS

code    SEGMENT                  ; Program Code Segment
        ASSUME cs:code,ds:data
start:                           ; Program Entry Point
        mov    ax,data           ; Load data segment location
        mov    ds,ax             ;   into DS register
        mov    dx,OFFSET string  ; Load string location
        mov    ah,09h            ; Call string display
        int    21h
        mov    ah,4Ch            ; Call terminate function
        int    21h
code    ENDS

stack   SEGMENT stack            ; Program Stack Segment
        DW     64 DUP(?)         ; Define stack space
stack   ENDS

        END    start             ; Mark end and define start
```

The following main features of this source file should be noted:

1. The **SEGMENT** and **ENDS** statements, which define segments named `data`, `code`, and `stack`.
2. The variable `string` in the `data` segment, which defines the string to be displayed. The variable data are defined in the `data` segment. They include the quoted dollar sign (`"$"`) required by the MS-DOS display-string function, as well as the ASCII codes for a carriage-return/line-feed combination.

3. The instruction label `start` in the `code` segment, which marks the start of the program instructions.
4. The **DW** statement in the `stack` segment, which defines the uninitialized data space to be used for the program stack.
5. The **ASSUME** statement for the `data` and `code` segments, which specifies which segment registers will be associated with the labels, variables, and symbols defined within the segments. An assume statement is not needed for the `stack` segment since the combine type `stack` tells **MASM** that the segment is associated with the **SS** register. See Section 3.4.2 for more information on combine types.
6. The first two code instructions, which load the address of the data segment into the **DS** register. These instructions are not necessary for the code and stack segments because the code-segment address is always loaded into the **CS** register and the stack-segment address is automatically loaded into the **SS** register when you use the **stack** combine type.
7. The last two instructions in the `code` segment, which use MS-DOS function `4Ch` to return to DOS. While there are other techniques for returning to DOS, this is the one recommended for most assembly-language programs.
8. The **END** directive, which indicates the end of the source file, and specifies `start` as the program entry point.

3.3 Instruction-Set Directives

Syntax

.8086
.8087
.186
.286c
.286p
.287

The instruction-set directives enable the instruction sets for the given microprocessors. When a directive is given, **MASM** will recognize and assemble any subsequent instructions belonging to that microprocessor.

The instruction-set directives, if used, must be placed at the beginning of the program source file to ensure all instructions in the file are assembled using the same instruction set.

The **.8086** directive enables assembly of instructions for the 8086 and 8088 microprocessors. It also disables assembly of the instructions unique to the 80186 and 80286 processors. Similarly, the **.8087** directive enables assembly of instructions for the 8087 floating-point coprocessor and disables assembly of instructions unique to the 80287 coprocessor.

Since MASM assembles 8086 and 8087 instructions by default, the **.8086** and **.8087** directives are not required if the source files contain 8086 and 8087 instructions only. Using the default instruction sets ensures that your programs will be usable on all processors in the 8086/80186/80286 family. However, they will not take advantage of the more powerful instructions available on the 80186, 80286, and 80287 processors.

The **.186** directive enables assembly of the 8086 instructions plus the additional instructions for the 80186 microprocessor. This directive should be used for programs that will be executed only by an 80186 microprocessor.

The **.286c** directive enables assembly of 8086 instructions and nonprotected 80286 instructions (identical to the 80186 instructions). The **.286p** directive enables assembly of the protected instructions of the 80286 in addition to the 8086 and nonprotected 80286 instructions. The **.286c** directive should be used with programs that will be executed only by an 80286 microprocessor, but do not use the protected instructions of the 80286. The **.286p** directive can be used with programs that will be executed only by an 80286 processor using both protected and nonprotected instructions.

The **.287** directive enables assembly of instructions for the 80287 floating-point coprocessor. This directive should be used with programs that have floating-point instructions and are intended for execution only by an 80286 microprocessor.

Even though a source file may contain the **.8087** or **.287** directive, MASM also requires the /**R** or /**E** option in the MASM command line to define how to assemble floating-point instructions. The /**R** option directs the assembler to generate the actual instruction code for the floating-point instruction. The /**E** option enables the assembler to generate code that can be used by a floating-point-emulator routine. See Sections 2.3.12 and 2.3.13 of the *Microsoft Macro Assembler User's Guide.*

3.4 SEGMENT and ENDS Directives

Syntax

name SEGMENT ⟦*align*⟧ ⟦*combine*⟧ ⟦'*class*'⟧
name ENDS

The **SEGMENT** and **ENDS** directives mark the beginning and end of a program segment. A program segment is a collection of instructions and/or data whose addresses are all relative to the same segment register.

The *name* defines the name of the segment. This name can be unique or be the same name given to other segments in the program. Segments with identical names are treated as the same segment.

The optional *align*, *combine*, and *class* types give the linker instructions on how to set up segments. They should be specified in order, but it is not necessary to enter all types, or any type, for a given segment.

Note

Don't confuse the **byte** and **word** align types with the **BYTE** and **WORD** reserved words used to specify data type with operators such as **THIS** and **PTR**. Also, the **page** align type and the **public** combine type should not be confused with the **PAGE** and **PUBLIC** directives. The distinction should be clear from context since the align and combine types are only used on the same line as the **SEGMENT** directive. To make the difference even clearer, align and combine types are shown with lowercase letters in this manual, although you can actually enter them in either case.

Sections 3.4.1–3.4.4 describe the three program-loading options and give an example program. Segment nesting is also explained in Section 3.4.5. Some of the information in this section is also discussed in Section 3.4 of the *Microsoft Macro Assembler User's Guide.*

3.4.1 Align Type

The optional *align* type defines the alignment of the given segment. The alignment defines the range of memory addresses from which a starting address for the segment can be selected. The align type can be any one of the following:

Align Type	Meaning
byte	Use any byte address
word	Use any word address (2 bytes/word)
para	Use paragraph addresses (16 bytes/paragraph)
page	Use page addresses (256 bytes/page)

If no *align* type is given, **para** is used by default. The actual start address is not computed until the program is loaded. The linker ensures that the address will be on the given boundary.

3.4.2 Combine Type

The optional *combine* type defines how to combine segments having the same name. The combine type can be any one of the following:

Combine Type	Meaning
public	Concatenates all segments having the same name to form a single, contiguous segment. All instruction and data addresses in the new segment are relative to a single segment register, and all offsets are adjusted to represent the distance from the beginning of the new segment.
stack	Concatenates all segments having the same name to form a single, contiguous segment. This combine type is the same as the **public** combine type, except that all addresses in the new segment are relative to the **SS** segment register. The stack pointer (**SP**) register is initialized to the ending address of the segment. Stack segments should normally use the **stack** type, since this automatically initializes the **SS** register. If you create a stack segment and do not use the **stack** type, you must give instructions to load the segment address into the **SS** register.

common — Creates overlapping segments by placing the start of all segments having the same name at the same address. The length of the resulting area is the length of the longest segment. All addresses in the segments are relative to the same base address. If data are declared in more than one segment having the same name and **common** type, the most recently declared data replace any previously declared data.

memory — Is treated by the Microsoft 8086 Object Linker (**LINK**) exactly like a **public** segment. **MASM** allows you to define segments with **memory** type even though **LINK** does not support a separate **memory** type. This feature is provided for compatibility with other linkers that may support a combine type conforming to the Intel definition of **memory** type.

at *address* — Causes all label and variable addresses defined in the segment to be relative to the given *address*. The *address* can be any valid expression, but must not contain a forward reference, that is, a reference to a symbol defined later in the source file. An **at** segment typically contains no code or initialized data. Instead, it represents an address template that can be placed over code or data already in memory, such as the screen buffer. The labels and variables in the **at** segments can then be used to access the fixed instructions and data.

If no *combine* type is given, the segment is not combined. Instead, it receives its own physical segment when loaded into memory.

Note

Normally you should provide at least one stack segment in a program. If no stack segment is declared, **LINK** will display a warning message. You can ignore this message if you have a specific reason for not declaring a stack segment.

3.4.3 Class Type

The optional *class* type defines which segments are to be loaded in contiguous memory. Segments having the same class name are loaded into memory one after another. All segments of a given class are loaded before segments of any other class. The *class* name must be enclosed in single quotation marks ('). Class names are not case-sensitive unless the **/ML** or **/MX** option is used during assembly, or the **/NOIGNORECASE** option is used when linking.

Note

The names assigned for class types of segments should not be used for other symbol definitions in the source file. For example, if you give a segment the class name `'CONSTANT'`, you should not give the name `constant` to any variable or labels in the source file. If you do, the error `Symbol already different kind` will be generated.

If class types are not specified, **LINK** copies segments to the executable file in the same order they are encountered 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 contiguous blocks to the executable file.

Example

```
DATAX   segment 'DATA'
DATAX   ends

TEXT    segment 'CODE'
TEXT    ends

DATAZ   segment 'DATA'
DATAZ   ends
```

In the preceeding example-program fragment, the segments `DATAX` and `DATAZ` both have class type `'DATA'`. As a result, both segments are copied to the executable file before the `TEXT` segment.

All segments belong to a class. Segments for which no class name is explicitly stated have the null-class name, and will be loaded as contiguous blocks 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 64K.

Since **LINK** processes modules in the order in which it receives them on the command line, you may not always be able to easily specify the order in which you want segments to be loaded. For example, assume your program has four segments that you want loaded in the following order: CODE, DATA, CONST, STACK. The CODE, CONST, and STACK segments are defined in the first module of your program, but the DATA segment is defined in the second module. **LINK** will not put the segments in the proper order because it will first load the segments encountered in the first module.

You can avoid this problem by creating and assembling a dummy program file containing empty segment definitions in the order in which you wish to load your real segments. Once this file is assembled, you can give it as the first object file in any invocation of **LINK**. The linker 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, CONST, and STACK.

```
CODE     segment para public 'CODE'
CODE     ends
DATA     segment para public 'DATA'
DATA     ends
CONST    segment para public 'CONST'
CONST    ends
STACK    segment para stack 'STACK'
STACK    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.

Do not use a dummy program file with Microsoft C, Pascal, FORTRAN, or compiled BASIC. These languages follow the MS-DOS segment-ordering convention described in Section 3.3.15 of the *Microsoft Macro Assembler User's Guide.* This loading order must not be modified.

Another way to control segment order is with the **MASM** **/A** option. This option directs **MASM** to write segments to the object file in alphabetical order. You can give segments names with alphabetical order that matches the order in which you want them loaded and then use the **/A** option. To make this strategy work with multiple-module programs, you should define all segments in the first module specified in the **LINK** command line. Some of the definitions may be dummy segments. See Section 2.3.1 of the *Microsoft Macro Assembler User's Guide* for more information on the **/A** option.

Note

Some previous versions of the assembler ordered segments alphabetically by default. If you have trouble assembling and linking source-code listings from books or magazines, try using the **/A** option. Listings written for the old version assemblers may not work without this option.

3.4.4 Program Example

The following source code illustrates one way in which the *align* and *combine* types can be used. Figure 3.1 (following the example below) shows the way **LINK** would load the given program into memory. The **memory** combine type is not shown since it is the same as **public**. The *class* types are not used in the sample program, but they are illustrated in Section 3.4.3 and in the example in Section 3.6.

Note

Although a given segment name can be used more than once in a source file, each segment definition using that name must have either exactly the same attributes, or attributes that do not conflict.

Example

```
                NAME module_1

seg_a           SEGMENT word public
start:          .
                .
                .
seg_a           ENDS

seg_b           SEGMENT page stack
                .
                .
                .
seg_b           ENDS

seg_c           SEGMENT para common
                .
                .
                .
seg_c           ENDS

seg_d           SEGMENT at 0B800h
                .
                .
                .
seg_d           ENDS
                END start

                NAME module_2

seg_a           SEGMENT word public
                .
                .
                .
seg_a           ENDS

seg_b           SEGMENT page stack
                .
                .
                .
seg_b           ENDS

seg_c           SEGMENT para common
                .
                .
                .
seg_c           ENDS
                END
```

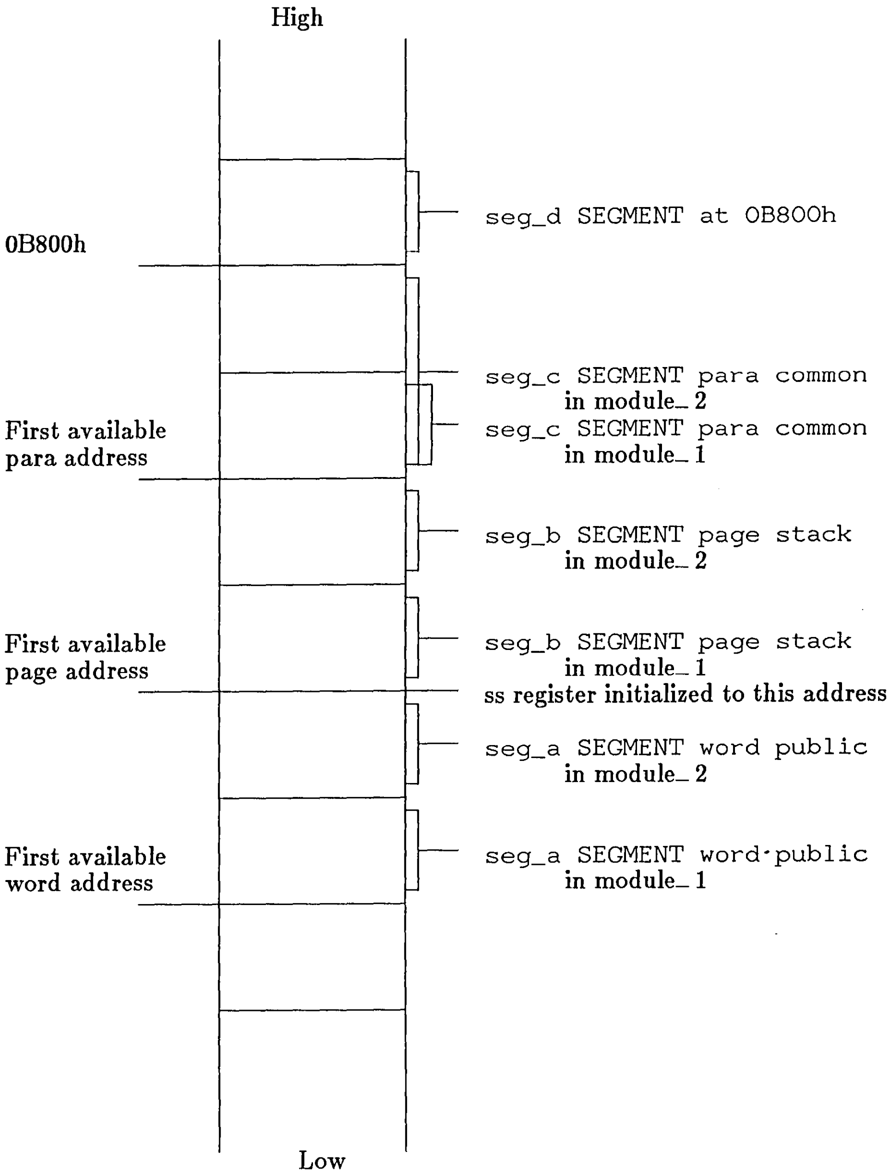


Figure 3.1 LINK Program Loading Order

3.4.5 Segment Nesting

Segments can be nested. When **MASM** encounters a nested segment, it temporarily suspends assembly of the enclosing segment and begins assembly of the nested segment. When the nested segment has been assembled, **MASM** continues assembly of the enclosing segment. Overlapping segments are not permitted.

Example

```
sample  SEGMENT word public 'CODE'      ; outside segment
main    PROC far
        .
        .
        .
const   SEGMENT word public 'CONST'     ; nested segment
array   DW      array_data
const   ENDS                            ; end nesting
        .
        .
        .
        RET
main    ENDP
sample  ENDS
```

This example-code fragment contains two segments: a code segment called `sample` and a data segment called `const`. The `const` segment is nested within the `sample` segment.

3.5 END Directive

Syntax

END [[*expression*]]

The **END** directive marks the end of a module. The assembler ignores any statements following this directive.

The optional *expression* defines the program entry point, the address at which program execution is to start. If the program has more than one module, only one of these modules can define an entry point. The module with the entry point is called the "main module". If no entry point is given, none is assumed.

Note

If you fail to define an entry point for the main module, your program may not be able to initialize correctly. The program will assemble and link without error messages, but it may crash when you attempt to run it. Remember, one (and only one) module must define an entry point.

Examples

```
        end
        end     start
```

3.6 GROUP Directive

Syntax

name **GROUP** *segmentname*,,,

The **GROUP** directive associates a group *name* with one or more segments, and causes all labels and variables defined in the given segments to have addresses relative to the beginning of the group rather than to the beginning of the segments in which they are defined. The *segmentname* must be the name of a segment defined using the **SEGMENT** directive, or a **SEG** expression (see Sections 3.4 and 5.3.12). The *name* must be unique.

The **GROUP** directive does not affect the order in which segments of a group are loaded. Loading order depends on each segment's class, or on the order in which object modules are given to the linker. Section 3.4.5 of the *Microsoft Macro Assembler User's Guide* also discusses groups and how they are handled by the linker.

Segments in a group need not be contiguous. Segments that do not belong to the group can be loaded between segments that do. The only restriction is that the distance (in bytes) between the first byte in the first segment of the group and the last byte in the last segment must not exceed 65535. Therefore, if the segments of a group are contiguous, the group can occupy up to 64K of memory.

Group names can be used with the **ASSUME** directive (Section 3.7) and as an operand prefix with the segment override operator (:) (Section 5.3.7).

Note

A group name must not be used in more than one **GROUP** directive in any source file. If several segments within the source file belong to the same group, all segment names must be given in the same **GROUP** directive.

Example

```
dgroup  GROUP   aseg,bseg
        ASSUME  ds:dgroup

aseg    SEGMENT byte public 'DATA1'
          .
sym_a:    .
          .
aseg    ENDS

bseg    SEGMENT byte public 'DATA2'
          .
sym_b:    .
          .
bseg    ENDS

cseg    SEGMENT byte public 'DATA1'
          .
sym_c:    .
          .
cseg    ENDS
        END
```

The order in which **LINK** will load these segments is shown in Figure 3.2. **LINK** loads `aseg` first because it occurs first in the source file. Next, **LINK** loads `cseg` because it has the same class type as `aseg`. **LINK** loads `bseg` last. However, `aseg` and `bseg` are declared part of the same group, despite their separation in memory. This means that the symbols `sym_a` and `sym_b` have offsets from the beginning of the group, which is also the beginning of `aseg`. The offset of `sym_c` is from the beginning of `cseg`. This sample is intended to illustrate the way **LINK** organizes segments in a group, rather than to show a typical use of a group.

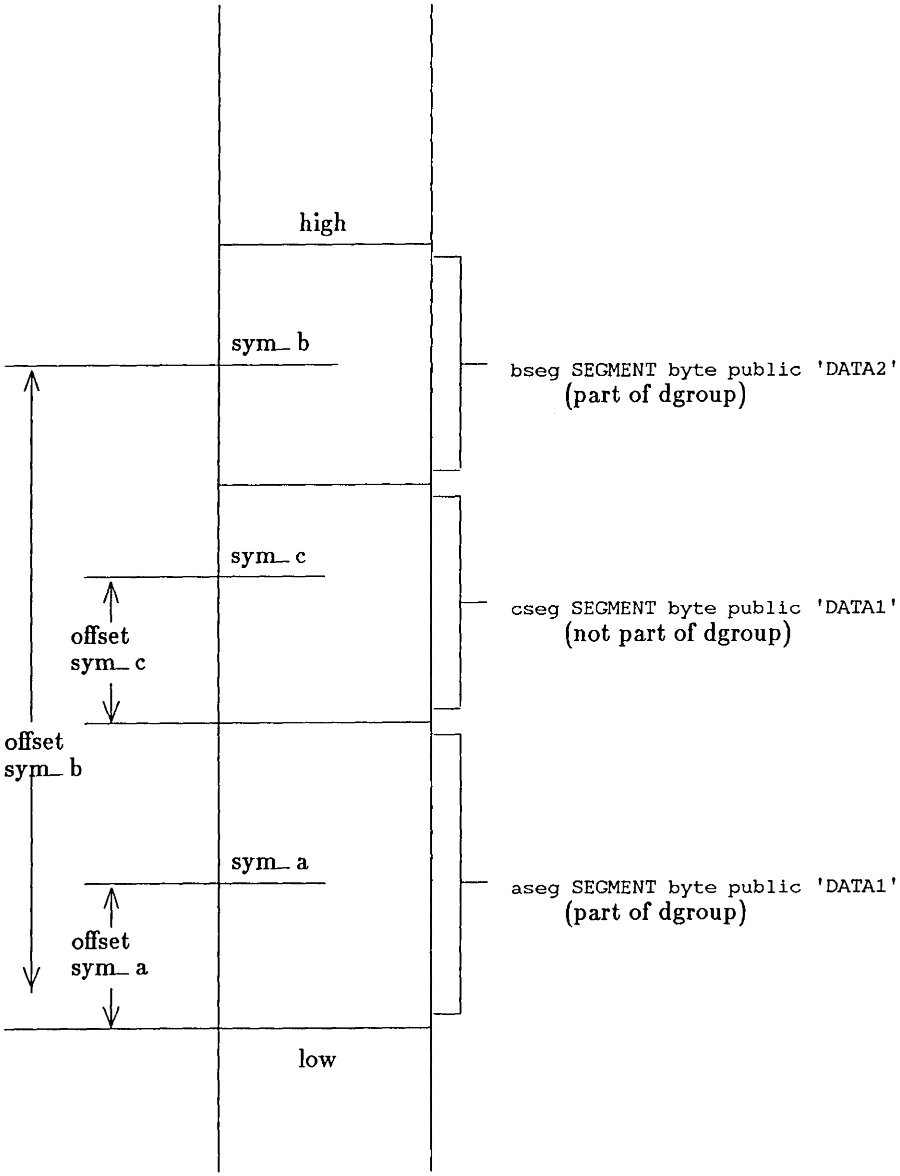


Figure 3.2 LINK Segment Loading Order

3.7 ASSUME Directive

Syntax

ASSUME *segmentregister:segmentname,,,*
ASSUME NOTHING

The **ASSUME** directive specifies *segmentregister* as the default segment register for all labels and variables defined in the segment or group given by *segmentname*. Subsequent references to the label or variable will automatically assume the selected register when the effective address is computed.

The **ASSUME** directive can define up to four selections: one for each of the four segment registers. The *segmentregister* can be any one of the segment register names: **CS**, **DS**, **ES**, or **SS**. The *segmentname* must be one of the following:

- The name of a segment that was previously defined with the **SEGMENT** directive
- The name of a group that was previously defined with the **GROUP** directive
- The keyword **NOTHING**

The keyword **NOTHING** cancels the current segment selection. The statement `ASSUME NOTHING` cancels all register selections made by a previous **ASSUME** statement.

Note

The segment-override operator (:) can be used to override the current segment register selected by the **ASSUME** directive.

Examples

```
ASSUME cs:CODE
ASSUME cs:cgroup,ds:dgroup,ss:nothing,es:nothing
ASSUME NOTHING
```

3.8 ORG Directive

Syntax

ORG *expression*

The **ORG** directive sets the location counter to *expression*. Subsequent instruction and data addresses begin at the new value.

The *expression* must resolve to an absolute number. In other words, all symbols used in the expression must be known on the first pass of the assembler. The location-counter symbol ($) can also be used.

Examples

```
        ORG     120h
        mov     ax,dx
```

In the first example, the statement `mov ax,dx` begins at byte 120h in the current segment.

```
        ORG     $+2
array   DW      100 dup (0)
```

In the second example, the variable `array` is declared to start at the address 2 bytes beyond the current address. See Section 5.2.4 for more information on the location-counter symbol ($).

3.9 EVEN Directive

Syntax

EVEN

The **EVEN** directive aligns the next data or instruction byte on a word boundary. If the current value of the location counter is odd, the directive increments the location counter to an even value and generates one **NOP** (no operation) instruction. If the location counter is already even, the directive does nothing.

Note

The **EVEN** directive must not be used in **byte**-aligned segments.

Example

```
        ORG     0
test1   DB      1
        EVEN
test2   DW      513
```

In this example, the **EVEN** directive tells **MASM** to increment the location counter, and generates a single **NOP** instruction (90h). This means the offset of `test2` is 2, not 1, as it would be without the **EVEN** directive.

3.10 PROC and ENDP Directives

Syntax

name **PROC** ⟦*distance*⟧
    statements
name **ENDP**

The **PROC** and **ENDP** directives mark the beginning and end of a procedure. A procedure is a block of instructions that forms a program subroutine. Every procedure has a *name* with which it can be called.

The *name* must be a unique name, not previously defined in the program. The optional *distance* can be either **NEAR** or **FAR**. **NEAR** is assumed if no *distance* is given. The *name* has the same attributes as a label, and can be used as an operand in a jump, call, or loop instruction.

Any number of *statements* can appear between the **PROC** and **ENDP** statements. The procedure should contain at least one **RET** directive to return control to the point of call. Nested procedures are allowed.

Example

```
        push    ax          ; Push third parameter
        push    bx          ; Push second parameter
        push    cx          ; Push first parameter
        call    addup       ; Call the procedure
        add     sp,6        ; Destroy the pushed parameters
        .
        .
        .
addup   PROC    near        ; Return address for near call
                            ;   takes two bytes
        push    bp          ; Save base pointer - takes two more
                            ;   so parameters start at 4th byte
        mov     bp,sp       ; Load stack into base pointer
        mov     ax,[bp+4]   ; Get first parameter
                            ;   4th byte above pointer
        add     ax,[bp+6]   ; Get second parameter
                            ;   6th byte above pointer
        add     ax,[bp+8]   ; Get third paramter
                            ;   8th byte above pointer
        pop     bp          ; Restore base
        RET                 ; Return
addup   ENDP
```

In this example, three numbers are passed as parameters for the procedure addup. Parameters are often passed to procedures by pushing them before the call so that the procedure can read them off the stack.

Note

The parameter-passing method in this example conforms to the standard used in Microsoft high-level languages. As a result, this procedure could be traced using the Stack Trace command (**K**) of the Microsoft Symbolic Debug Utility (**SYMDEB**), described in Section 4.6.28 of the *Microsoft Macro Assembler User's Guide.*

Chapter 4
Types and Declarations

4.1 Introduction

This chapter explains how to generate data for a program; how to declare labels, variables, and other symbols that refer to instruction and data locations; and how to define types that can be used to generate data blocks containing multiple fields, such as structures and records.

4.2 Label Declarations

Label declarations create "labels." A label is a name that represents the address of an instruction. Labels can be used in jump, call, and loop instructions to direct program execution to the instruction at the address of the label.

4.2.1 Near-Label Declarations

Syntax

name:

A near-label declaration creates an instruction label that has **NEAR** type. The label can be used in subsequent instructions in the same segment to pass execution control to the corresponding instruction.

The *name* must be unique, not previously defined, and it must be followed by a colon (:). Furthermore, the segment containing the declaration must be associated with the **CS** segment register (see Section 3.7 for information on the **ASSUME** directive). The assembler sets the name to the current value of the location counter.

A near-label declaration can appear on a line by itself or on a line with an instruction. Labels must be declared with the **PUBLIC** or **EXTRN** directive if they are located in one module but called from another module (see Chapter 6).

Examples

```
start:
cycle:  inc     si
```

4.2.2 Procedure Labels

Syntax

name PROC ⟦*distance*⟧

The **PROC** directive creates a label *name* and optionally assigns it a *distance*. The distance can be **NEAR** or **FAR**. The label then represents the address of the first instruction of a procedure. The label can be used in a **CALL** instruction (or in a jump or loop instruction) to direct execution control to the first instruction of the procedure. If you do not specify the type for a procedure, the assembler assumes **NEAR** as the default.

When the **PROC** label definition is encountered, the assembler sets the label's value to the current value of the location counter and sets its type to **NEAR** or **FAR**. If the label has **FAR** type, the assembler also sets its segment value to that of the enclosing segment.

NEAR labels can be used with jump, call, or loop instructions to transfer program control to any address in the current segment. **FAR** labels can be used to transfer program control to an address in any segment outside the current segment.

Labels must be declared with the **PUBLIC** and **EXTRN** directive if they are located in one module but called from another module (see Chapter 6).

4.3 Data Declarations

The data-declaration directives let you generate data for a program. The directives translate numbers, strings, and expressions into individual bytes, words, or other units of data. The encoded data are copied to the object file.

The data-declaration directives are listed below:

Directive	**Meaning**
DB	Define byte
DW	Define word
DD	Define doubleword
DQ	Define quadword
DT	Define ten bytes

Sections 4.3.1–4.3.5 describe these directives in detail.

4.3.1 DB Directive

Syntax

⟦*name*⟧ **DB** *initialvalue,,,*

The **DB** directive allocates and initializes a byte (8 bits) of storage for each *initialvalue*. The *initialvalue* can be an integer, a character string constant, a **DUP** operator, a constant expression, or a question mark (?). The question mark represents an undefined initial value. If two or more initial values are given, they must be separated by commas (,).

The *name* is optional. If *name* is given, the directive creates a variable of type **BYTE** whose offset value is the current location-counter value.

A string constant can have any number of characters, as long as it fits on a single line. When the string is encoded, the characters are stored in the order given, with the first character in the constant at the lowest address and the last at the highest.

Examples

```
integer         DB      16
string          DB      'ab'
message         DB      "Enter your name: "
constantexp     DB      4*3
empty           DB      ?
multiple        DB      1,2,3,'$'
duplicate       DB      10 dup(?)
high_byte       DB      255
```

4.3.2 DW Directive

Syntax

⟦*name*⟧ **DW** *initialvalue*,,,

The **DW** directive allocates and initializes a word (2 bytes) of storage for each *initialvalue*. The *initialvalue* can be an integer, a one- or two-character string constant, a **DUP** operator, a constant expression, an address expression, or a question mark (?). The question mark represents an undefined initial value. If two or more expressions are given, they must be separated by commas (,).

The *name* is optional. If *name* is given, the directive creates a variable of type **WORD** whose offset value is the current location-counter value.

String constants must not consist of more than two characters. The last (or only) character in the string is placed in the low-order byte. Either 0 or the first character is placed in the high-order byte.

Examples

```
integer         DW      16728
character       DW      'a'
string          DW      'bc'
constantexp     DW      4*3
addressexp      DW      string
empty           DW      ?
multiple        DW      1,2,3,'$'
duplicate       DW      10 dup(?)
high_word       DW      65535
arrayptr        DW      array
arrayptr2       DW      offset DGROUP:array
```

4.3.3 DD Directive

Syntax

⟦*name*⟧ **DD** *initialvalue*,,,

The **DD** directive allocates and initializes a doubleword (4 bytes) of storage for each *initialvalue*. The *initialvalue* can be an integer, a real number, a one- or two-character string constant, an encoded real number, a **DUP** operator, a constant expression, an address expression, or a question mark

(?). The question mark represents an undefined initial value. If two or more initial values are given, they must be separated by commas (,).

The *name* is optional. If *name* is given, the directive creates a variable of type **DWORD** whose offset value is the current location-counter value.

String constants must not consist of more than two characters. The last (or only) character in the string is placed in the low-order byte, and the first character (if there are two in the string) is placed in the next byte. Zeroes are placed in all remaining bytes.

Examples

```
integer         DD      16728
character       DD      'a'
string          DD      'bc'
real            DD      1.5
encodedreal     DD      3F000000r
constantexp     DD      4*3
aDDsegexp       DD      real
empty           DD      ?
multiple        DD      1,2,3,'$'
duplicate       DD      10 dup(?)
high_double     DD      4294967295
```

4.3.4 DQ Directive

Syntax

⟦*name*⟧ **DQ** *initialvalue*,,,

The **DQ** directive allocates and initializes a quadword (8 bytes) of storage for each *initialvalue*. The *initialvalue* can be an integer, a real number, a one- or two-character string constant, an encoded real number, a **DUP** operator, a constant expression, or a question mark (?). The question mark represents an undefined initial value. If two or more initial values are given, they must be separated by commas (,).

The *name* is optional. If *name* is given, the directive creates a variable of type **QWORD** whose offset value is the current location-counter value.

String constants must not consist of more than two characters. The last (or only) character in the string is placed in the low-order byte, and the first character (if there are two in the string) is placed in the next byte. Zeroes are placed in all remaining bytes.

Examples

```
integer         DQ      16728
character       DQ      'a'
string          DQ      'bc'
real            DQ      1.5
encodedreal     DQ      3F00000000000000r
constantexp     DQ      4*3
empty           DQ      ?
multiple        DQ      1,2,3,'$'
duplicate       DQ      10 dup(?)
high_quad       DQ      18446744073709551615
```

4.3.5 DT Directive

Syntax

⟦*name*⟧ DT *initialvalue*,,,

The **DT** directive allocates and initializes 10 bytes of storage for each *initialvalue*. The *initialvalue* can be an integer expression, a packed decimal, a one- or two-character string constant, an encoded real number, a **DUP** operator, or a question mark (?). The question mark represents an undefined initial value. If two or more initial values are given, they must be separated by commas (,).

The *name* is optional. If *name* is given, the directive creates a variable of type **TBYTE** whose offset value is the current location-counter value.

String constants must not consist of more than two characters. The last (or only) character in the string is placed in the low-order byte, and the first character (if there are two in the string) is placed in the next byte. Zeroes are placed in all remaining bytes.

Note

The **DT** directive assumes that constants with decimal digits are packed decimals, not integers. If you want to specify a 10-byte integer, you must follow the number with the letter that specifies the number system you are using (for example, "D" or "d" for decimal or "H" or "h" for hexadecimal).

Examples

```
packeddecimal   DT      1234567890
integer         DT      16728d
character       DT      'a'
string          DT      'bc'
real            DT      1.5
encodedreal     DT      3F00000000000000000r
empty           DT      ?
multiple        DT      1,2,3,'$'
duplicate       DT      10 dup(?)
high_tbyte      DT      1208925819614629174706175d
```

4.3.6 DUP Operator

Syntax

count DUP(*initialvalue*,,,)

The **DUP** operator is a special operator that can be used with the data-declaration directives and other directives to specify multiple occurrences of one or more initial values. The *count* sets the number of times to define *initialvalue*. The initial value can be any expression that evaluates to an integer value, a character constant, or another **DUP** operator. If more than one initial value is given, the values must be separated by commas (,). **DUP** operators can be nested up to 17 levels. The initial value (or values) must always be placed within parentheses.

Examples

```
        DB      100     DUP(1)
```

The first example generates 100 bytes with initial value 1.

```
        DW      20     DUP( 1,2,3,4 )
```

The second example generates 80 words of data. The first four words have the initial values 1, 2, 3, and 4, respectively. This pattern is duplicated for the remaining words.

```
        DB      5     DUP( 5 DUP( 5 DUP (1)))
```

The third example generates 125 bytes of data, each byte having the initial value 1.

```
        DD      14      DUP(?)
```

The final example generates 14 doublewords of uninitialized data.

4.4 Symbol Declarations

The symbol-declaration directives let you create and use symbols. A symbol is a descriptive name representing a number, text, an instruction, or an address. Symbols make programs easier to read and maintain by using descriptive names to represent values. A symbol can be used anywhere its corresponding value is allowed.

The symbol declaration directives are listed below:

Directive	**Meaning**
=	Assign absolutes
EQU	Equate absolutes, aliases, or text symbols
LABEL	Create instruction or data labels

Sections 4.4.1–4.4.3 describe the directives in detail.

4.4.1 Equal-Sign (=) Directive

Syntax

name=*expression*

The equal-sign (=) directive creates an absolute symbol by assigning the numeric value of *expression* to *name*. An absolute symbol is simply a name that represents a 16-bit value. No storage is allocated for the number. Instead, the assembler replaces each subsequent occurrence of *name* with the value of *expression*. The value is variable during assembly, but is a constant at run time.

The expression can be an integer, a one- or two-character string constant, a constant expression, or an address expression. Its value must not exceed 65535. The name must be either a unique name, or a name previously defined using the equal-sign (=) directive.

Absolute symbols can be redefined at any time.

Examples

```
integer         =       16728
string          =       'ab'
constantexp     =       3 * 4
addressexp      =       string
```

4.4.2 EQU Directive

Syntax

name **EQU** *expression*

The **EQU** directive creates absolute symbols, aliases, or text symbols by assigning *expression* to *name*. An absolute symbol is a name that represents a 16-bit value; an alias is a name that represents another symbol; and a text symbol is a name that represents a character string or other combination of characters. The assembler replaces each subsequent occurrence of the name with either the text or the value of the expression, depending on the type of expression given.

The name must be a unique name, one which has not been previously defined. The expression can be an integer, a string constant, a real number, an encoded real number, an instruction mnemonic, a constant expression, or an address expression. Expressions that evaluate to values in the range 0 to 65535 create absolute symbols and cause **MASM** to replace the name with a value. All other expressions cause the assembler to replace the name with text.

The **EQU** directive is sometimes used to create simple macros. Note that the assembler replaces a name with text or a value before attempting to assemble the statement containing the name.

Symbols defined using the **EQU** directive cannot be redefined.

Examples

```
k           EQU  1024           ; Replaced with value
pi          EQU  3.14159        ; Replaced with text
matrix      EQU  20 * 30        ; Replaced with value
staptr      EQU  [bp]           ; Replaced with text
clearax     EQU  xor ax,ax      ; Replaced with text
prompt      EQU  'Type Enter'   ; Replaced with text
bpt         EQU  BYTE PTR       ; Replaced with text
```

4.4.3 LABEL Directive

Syntax

name **LABEL** *type*

The **LABEL** directive creates a new variable or label by assigning the current location-counter value and the given *type* to *name*.

The name must be unique and not previously defined. The type can be any one of the following:

BYTE

WORD

DWORD

QWORD

TBYTE

NEAR

FAR

The type can also be the name of a valid structure type.

Examples

```
barray          LABEL   BYTE
warray          DW      100 DUP(?)
```

In this example, `barray` and `warray` refer to the same data. The data can be accessed by byte with `barray` or by word with `warray`.

4.5 Type Declarations

The type-declaration directives let you define data types that can be used to create program variables consisting of multiple elements or fields. The directives associate one or more named fields with a given type name. The type name can then be used in a data declaration to create a variable of the given type.

The type-declaration directives are listed below:

Directive	Declaration
STRUC and **ENDS**	Structure types
RECORD	Record types

Sections 4.5.1 and 4.5.2 describe these directives in detail.

4.5.1 STRUC and ENDS Directives

Syntax

name STRUC
fielddefinitions
name ENDS

The **STRUC** and **ENDS** directives mark the beginning and end of a type definition for a structure. A type definition for a structure defines the name of a structure type and the number, type, and default values of the fields contained in the structure.

A structure definition creates a template for data. Though this template is used by **MASM** during assembly, it does not in itself create any data. Data can only be created when you declare a structure, as described in Section 4.6.1.

The *name* defines the new name of the structure type. It must be unique. The *fielddefinitions* define the structure's fields. Any number of field definitions can be given. The definitions must have one of the following forms:

⟦*name*⟧ **DB** *defaultvalue,,,*
⟦*name*⟧ **DW** *defaultvalue,,,*
⟦*name*⟧ **DD** *defaultvalue,,,*
⟦*name*⟧ **DQ** *defaultvalue,,,*
⟦*name*⟧ **DT** *defaultvalue,,,*

The optional *name* specifies the field name; the **DB**, **DW**, **DD**, **DQ**, and **DT** directives define the size of each field; and *defaultvalue* defines the value to be given to the field if no initial value is given when the structure variable is declared. The name must be unique, and, once defined, represents the offset from the beginning of the structure to the corresponding field.

The default value can define a number, character or string constant, or symbol. It may also contain the **DUP** operator to define multiple values for the field. If the default value is a string constant, the field has the same number of bytes as characters in the string. If multiple default values are given, they must be separated by commas (,).

A definition of a structure type can contain field definitions and comments only. It must not contain any other statements. Therefore, structures cannot be nested.

Example

```
table   STRUC
        count   DB      10
        value   DW      10 DUP(?)
        tname   DB      'font3'
table   ENDS
```

In this example, the fields are `count`, `value`, and `tname`. The `count` field is a single-byte value initialized to 10; `value` is an array of 10 uninitialized word values; and `tname` is a character array of 5 bytes initialized to `'font3'`. The field names `count`, `value`, and `tname` have the offset values 0, 1, and 21, respectively.

4.5.2 RECORD Directive

Syntax

recordname **RECORD** *fieldname*:*width*⟦=*expression*⟧,,,

The **RECORD** directive defines a record type for an 8- or 16-bit record that contains one or more fields. The *recordname* is the name of the record type to be used when creating the record; *fieldname* is the name of a field in the record, *width* is the number of bits in the field; and *expression* is the initial (or default) value for the field.

Any number of *fieldname:width=expression* combinations can be given for a record, as long as each is separated from its predecessor by a comma (,). The sum of the widths for all fields must not exceed 16 bits.

The width must be a constant in the range 1 to 16. If the total width of all declared fields is larger than 8 bits, then the assembler uses 2 bytes. Otherwise, only 1 byte is used.

If =*expression* is given, it defines the initial value for the field. If the field is at least 7 bits wide, you can use an ASCII character for *expression*. The expression must not contain a forward reference to any symbol.

In all cases, the first field you declare goes into the most significant bits of the record. Successively declared fields are placed in the succeeding bits to the right. If the fields you declare do not total exactly 8 bits or exactly 16 bits, the entire record is shifted right so that the last bit of the last field is the lowest bit of the record. Unused bits will be initialized to 0 in the high end of the record.

The **RECORD** directive creates a template for data. This template is used by the assembler during assembly, but it does not in itself create any data. Data can only be created when you declare a record, as described in Section 4.6.2.

Examples

```
encode  RECORD  hi:4, mid:3, lo:3
```

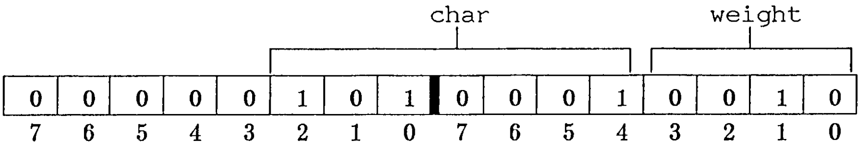
The example above creates a record type `encode` having three fields: `hi`, `mid`, and `lo`. Each record declared using this type will occupy 16 bits of memory. The `hi` field will be in bits 6 to 9 (bit 9 is bit 1 in the high byte); the `mid` field will be in bits 3 to 5; and the `lo` field will be in bits 0 to 2. The remaining high-order bits will be unused. The bit diagram below shows what the record type will look like:

						hi				mid			lo		
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0

Since no initial values are given, the record type has all bits set to 0. Note that this is only a template maintained by the assembler. No data are created.

```
item    RECORD  char:7='Q', weight:4=2
```

The example above creates a record type `item` having two fields: `char` and `weight`. These values are initialized to the letter `Q` and the number `2`, respectively. Unused bits are set to 0, as shown in the bit diagram below.



4.6 Structure and Record Declarations

Structure and record declarations allow you to generate blocks of data bytes with many elements or fields. A structure or record declaration consists of the name of a previously defined structure or record, and a set of initial values.

Sections 4.6.1–4.6.2 describe these declarations in detail.

4.6.1 Structure Declarations

Syntax

⟦*name*⟧ *structurename* <⟦*initialvalue*,,,⟧>

A structure variable is a variable with one or more fields of different sizes. The *name* is the name of the variable; *structurename* is the name of a structure type created using the **STRUC** directive; and *initialvalue* is one or more values defining the initial value of the structure. One initial value can be given for each field in the structure.

The *name* is optional. If not given, the assembler allocates space for the structure, but does not create a name you can use to access the structure.

The *initialvalue* can be an integer, string constant, or expression that evaluates to a value having the same type as the corresponding field. The angle brackets (< >) are required even if no initial value is given. If more than one initial value is given, the values must be separated by commas (,). If the **DUP** operator (see Section 4.3.6) is used, only the values within the parentheses need to be enclosed in angle brackets.

You need not initialize all fields in a structure. If an initial value is left blank, the assembler automatically uses the default initial value of the field, which was originally determined by the structure type. If there is no default value, the field is uninitialized. Section 5.2.9 illustrates several ways to use structure data after they have been declared.

Note

You cannot initialize any structure field that has multiple values if this field was given a default initial value when the structure was defined. For example, assume the following structure definition:

```
strings     STRUC
            buffer  DB 100 DUP (?)   ; Can't override
            crlf    DB 13,10         ; Can't override
            query   DB 'Filename: '  ; String <= can override
            endmark DB 36            ;
strings     ENDS
```

The `buffer` and `crlf` variables cannot be overridden because they have multiple values. The `query` variable can be overridden as long as the overriding data are no longer than `query` (10 bytes). Similarly, the `endmark` field can be overridden by any byte value.

Examples

```
struct1 table    <>
```

The preceding example creates a structure variable named `struct1` whose type is given by the structure type `table`. The initial values of the fields in the structure are set to the default values for the structure type, if any. For example, if `table` were defined with the structure definition in the example in Section 4.5.1, the first byte of `struct1` would be 10; 10 uninitialized words would follow; and finally would come the byte string `font3`.

```
struct2 table   <0,,>
```

The second example creates a structure variable named `struct2`. Its type is also `table`. The initial value for the first field is set to `0`. The default values defined by the structure type are used for the remaining two fields. If `table` were defined with the structure definition in the example in Section 4.5.1, the initial value of `0`, set with the structure declaration above, would override the initial value of `10`, set with the original structure definition.

```
struct3 table   10 DUP(<0,,>)
```

This final example creates a variable, `struct3`, containing 10 structures of the type `table`. The first field in each structure is set to the initial value of `0`. All remaining fields receive the default values.

4.6.2 Record Declarations

Syntax

⟦*name*⟧ *recordname* <⟦*initialvalue,,,*⟧>

A record variable is an 8- or 16-bit value whose bits are divided into one or more fields. The *name* is the name of the variable; *recordname* is the name of a record type that has been created using the **RECORD** directive; and *initialvalue* is one or more values defining the initial value of the record. One *initialvalue* can be given for each field in the record.

The name is optional. If no *name* is given, **MASM** allocates space for the record, but does not create a variable that you can use to access the record.

The optional *initialvalue* can be an integer, string constant, or any expression that resolves to a value no larger than can be represented in the field width specified when the record was defined. Angle brackets (< >) are required even if no initial value is given. If more than one initial value is given, the values must be separated by commas (,). If the **DUP** operator (see Section 4.3.6) is used, only the values within the parentheses need to be enclosed in angle brackets. You do not have to initialize all fields in a record. If an initial value is left blank, the assembler automatically uses the default initial value of the field. This is defined by the record type. If there is no default value, the field is uninitialized.

Sections 5.2.10 and 5.2.11 illustrate ways to use record data after it has been defined.

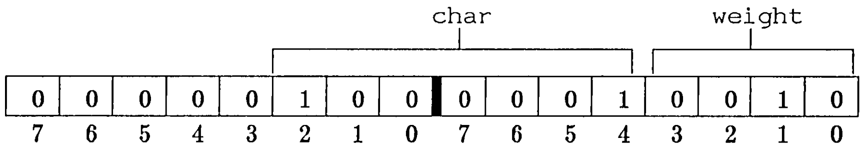
Examples

```
rec1    encode  <>
```

The first example creates a variable named `rec1` whose type is given by the record type `encode`. The initial values of the fields in the record are set to the default values for the record type, if any. For example, if `encode` were defined with the definition in the example in Section 4.5.2, `rec1` would be 0, since the fields were not initialized in the definition.

```
table   item    10 DUP(<'A',2>)
```

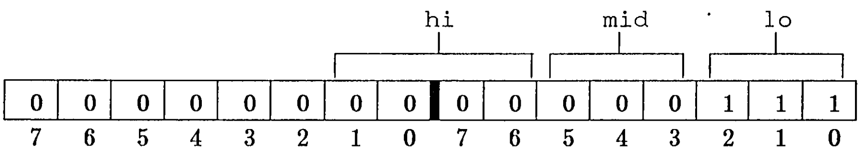
This second example creates a variable named `table` containing `10` records of the record type `item`. The fields in these records are all set to the initial values `A` and `2`. If the `item` definition from the example in Section 4.5.2 were used, the `A` would override the initial value of `Q` in the record definition.



The bit diagram above shows the value of the 10 bytes created by the record declaration.

```
passkey encode  <,,7>
```

The final example creates a record variable named `passkey`. Its type is `encode`. The initial values for the first two fields are the default values defined by the record type. The initial value for the third field is `7`. If the record definition from Section 4.5.2 were used, the first two fields would remain 0, since they were not initialized. The bit diagram below shows what the record looks like.



Chapter 5
Operands and Expressions

5.1 Introduction

This chapter describes the syntax and meaning of operands and expressions used in assembly-language statements and directives. Operands represent values, registers, or memory locations to be acted on by instructions or directives. Expressions combine operands with arithmetic, logical, bitwise, and attribute operators to calculate a value or memory location that can be acted on by an instruction or directive. Operators indicate what operations will be performed on one or more values in an expression to calculate the value of the expression.

5.2 Operands

An operand is a constant, label, variable, or other symbol that is used in an instruction or directive to represent a value, register, or memory location to be acted on.

The operand types are listed below:

Constant

Direct-memory

Relocatable

Location-counter

Register

Based

Indexed

Based-indexed

Structure

Record

Record-field

5.2.1 Constant Operands

Syntax

number|*string*|*expression*

A constant operand is a number, string constant, symbol, or expression that evaluates to a fixed value. Constant operands, unlike other operands, represent values to be acted on, rather than memory addresses.

Examples

```
mov     ax,9
mov     al,'c'
mov     bx,65535/3
mov     cx,count
```

Note that `count` in the last example is a constant only if it was defined with the **EQU** or equal-sign (=) operator. If `count` is a symbol representing a relocatable value or address, it is not a constant.

5.2.2 Direct-Memory Operands

Syntax

segment:*offset*

A direct-memory operand is a pair of segment and offset values that represents the absolute memory address of 1 or more bytes of memory. The *segment* can be a segment register (**CS**, **DS**, **SS**, or **ES**), a segment name, or a group name. The *offset* must be an integer, absolute symbol, or expression that resolves to a value within the range 0 to 65535.

Examples

```
mov     dx,ss:0031h
mov     bx,data:0
mov     ax,DGROUP:block
```

5.2.3 Relocatable Operands

Syntax

symbol

A relocatable operand is any symbol that represents the memory address (segment and offset) of an instruction or of data to be acted upon. Relocatable operands, unlike direct-memory operands, are relative to the start of the segment or group in which the symbol is defined, and have no explicit value until the program has been linked.

Examples

```
call    main
mov     bx,value
mov     bx,OFFSET dgroup:table
mov     cx,count
```

Note that `count` in the last example is a relocatable operand if it was defined with the **DW** directive. If `count` was defined with the **EQU** or equal-sign (=) operator, it is a constant.

5.2.4 Location-Counter Operand

Syntax

$

The location counter is a special operand that, during assembly, represents the current location within the current segment. The location counter has the same attributes as a near label. It represents an instruction address that is relative to the current segment. Its offset is equal to the number of bytes generated for that segment to that point. After each statement in the segment has been assembled, the assembler increments the location counter by the number of bytes generated.

Example

```
help     DB      'Program options:',13,10
F1       DB      '  F1      This help screen',13,10
F2       DB      '  F2      Save file',13,10
          .
          .
          .
F10      DB      '  F10     Exit program',13,10,'$'
DISTANCE =       $-help
```

In this example, the location counter forces the assembler to count the total length of a group of declared strings, saving the programmer the trouble of counting each byte.

5.2.5 Register Operands

Syntax

registername

A register operand is the name of a CPU register. Register operands direct instructions to carry out actions on the contents of the given registers. The *registername* can be any of the register names in Table 5.1.

Table 5.1

Register Operands

Register Operand Type	Register Name			
16-bit general purpose	**AX**	**BX**	**CX**	**DX**
8-bit high registers	**AH**	**BH**	**CH**	**DH**
8-bit low registers	**AL**	**BL**	**CL**	**DL**
16-bit segment	**CS**	**DS**	**SS**	**ES**
16-bit pointer and index	**SP**	**BP**	**SI**	**DI**

Any combination of upper- and lowercase letters is allowed.

The **AX**, **BX**, **CX**, and **DX** registers are 16-bit, general-purpose registers. They can be used for any data or numeric manipulation. The **AH**, **BH**,

CH, **DH** registers represent the high-order 8 bits of the corresponding general-purpose registers. Similarly, **AL**, **BL**, **CL**, and **DL** represent the low-order 8 bits of the general-purpose registers.

The **CS**, **DS**, **SS**, and **ES** registers are the segment registers. They contain the current segment addresses of the code, data, stack, and extra segments, respectively. All instruction and data addresses are relative to the segment address in one of these registers.

The **SP** register is the 16-bit stack-pointer register. The stack pointer contains the current top-of-stack address. This address is relative to the segment address in the **SS** register and is automatically modified by instructions that access the stack.

The **BX**, **BP**, **DI**, and **SI** registers are 16-bit, base and index registers. These are general-purpose registers typically used for pointers to program data. Address expressions using the **BP** register have offsets in the **SS** segment by default. Expressions using **BX**, **SI**, or **DI** have offsets in the **DS** segment by default. The **DI** register always has an offset in the **ES** segment when used with string instructions.

The unnamed, 16-bit flag register contains nine 1-bit flags whose positions and meanings are defined in Table 5.2.

Table 5.2

Flag Positions

Flag Bit	Meaning
0	Carry flag
2	Parity flag
4	Auxiliary flag
6	Zero flag
7	Sign flag
8	Trap flag
9	Interrupt-enable flag
10	Direction flag
11	Overflow flag

Although the 16-bit flag register has no name, the contents of the register can be accessed using the **LAHF**, **SAHF**, **PUSHF**, and **POPF** instructions. See Appendix A.2, 8086 Instructions.

5.2.6 Based Operands

Syntax

displacement[BP]
displacement[BX]

A based operand represents a memory address relative to one of the base registers: **BP** or **BX**. The *displacement* can be any immediate or direct-memory operand. It must evaluate to an absolute number or memory address. If no displacement is given, zero is assumed.

The effective address of a based operand is the sum of the displacement value and the contents of the given register. If **BP** is used, the operand's address is relative to the segment pointed to by the **SS** register. If **BX** is used, the address is relative to the segment pointed to by the **DS** register.

Based operands have a variety of alternate forms. Equivalent forms include the following:

[*displacement*][**BP**]
[**BP**+*displacement*]
[**BP**].*displacement*
[**BP**]+*displacement*

In each case, the effective address is the sum of the displacement and the contents of the given register.

Examples

```
        mov     ax, [bp]
        mov     ax, [bx]
        mov     ax,12[bx]
        mov     ax,fred[bp]
```

5.2.7 Indexed Operands

Syntax

displacement[SI]
displacement[DI]

An indexed operand represents a memory address relative to one of the index registers: **SI** or **DI**. The *displacement* can be any immediate or direct-memory operand. It must evaluate to an absolute number or memory address. If no displacement is given, zero is assumed.

The effective address of an indexed operand is the sum of the displacement value and the contents of the given register. The address is relative to the segment pointed to by the **DS** register.

Indexed operands have a variety of alternate forms. Equivalent forms include the following:

[*displacement*][**DI**]
[**DI**+*displacement*]
[**DI**].*displacement*
[**DI**]+*displacement*

In each case, the effective address is the sum of the displacement and the contents of the given register.

Examples

```
        mov     ax,[si]
        mov     ax,[di]
        mov     ax,12[di]
        mov     ax,fred[si]
```

5.2.8 Based-Indexed Operands

Syntax

displacement[**BP**][**SI**]
displacement[**BP**][**DI**]
displacement[**BX**][**SI**]
displacement[**BX**][**DI**]

A based-indexed operand represents a memory address relative to a combination of base and index registers. The *displacement* can be any immediate or direct-memory operand. It must evaluate to an absolute number or memory address. If no displacement is given, zero is assumed.

The effective address of a based-indexed operand is the sum of the displacement value and the contents of the given registers. If the **BD** register is used, the address is relative to the segment pointed to by the **SS** register. Otherwise, the address is relative to the segment pointed to by the **DS** register.

Based-indexed operands have a variety of alternate forms. Equivalent forms include the following:

[*displacement*][**BP**][**DI**]
[**BP+DI+***displacement*]
[**BP+DI**].*displacement*
[**DI**]+*displacement*+[**BP**]

In each case, the effective address is the sum of the displacement and the contents of the given registers. Either base register can be combined with either index register, but combining two base or two index registers is not allowed.

Examples

```
        mov     ax, [bp] [si]
        mov     ax, [bx+di]
        mov     ax,12[bp+di]
        mov     ax,fred[bx] [si]
        mov     ax,fred[bx] [bp]     ; Error - base registers combined
        mov     ax,fred[di] [si]     ; Error - index registers combined
```

5.2.9 Structure Operands

Syntax

variable.field

A structure operand represents the memory address of one member of a structure. The *variable* must either be the name of a structure or it must be a memory operand that resolves to the address of a structure. The *field* must be the name of a field within that structure. The *variable* is separated from *field* by the structure field-name operator (.), which is described in Section 5.3.8.

The effective address of a structure operand is the sum of the offsets of *variable* and *field.* The address is relative to the segment or group in which the variable is defined.

Examples

```
date    STRUC
        month   DW  ?
        day     DW  ?
        year    DW  ?
date    ENDS

current_date  date <'ja','01','84'>

        mov     ax,current_date.day
        mov     current_date.year,'85'
```

In the example above, the structure is first defined and declared. The first **MOV** instruction puts '01' (the value of current_date.day) in the **AX** register. The next instruction puts the value '85' in the variable current_date.year.

```
stframe  STRUC          ; stack frame
 retadr  DW     ?       ; from lowest...
 dest    DW     ?
 source  DW     ?
 nbytes  DW     ?       ; ...to highest address
stframe  ENDS

copy     PROC   NEAR    ; Push nbytes, source, dest before calling
         mov    bx,sp   ; Load stack into base register
         mov    ax,ds
         mov    es,ax               ; (es) = data segment
         mov    di,ss:[bx].dest     ; (di) = destination
         mov    si,ss:[bx].source   ; (si) = source
         mov    cx,ss:[bx].nbytes   ; (cx) = nbytes
         rep    movsb   ; move bytes from ds:si to es:di
         ret
copy     ENDP
```

In this example, structure operands are used to access values on the stack.

Note

The procedure in the example above does not conform to the method of passing parameters used in Microsoft high-level languages. As a result, you could not use the **SYMDEB** Stack Trace command (**K**) in this case procedure. See Section 4.6.27 in the *Microsoft Macro Assembler User's Guide.*

5.2.10 Record Operands

Syntax

recordname <⟦*value*⟧,,,>

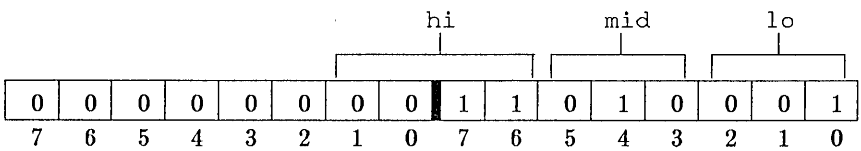
A record operand refers to the value of a record type. The operands can be in expressions. The *recordname* must be the name of a record type defined in the source file. The optional *value* is the value of a field in the record. If more than one *value* is given, the values must be separated by commas (,). Values include expressions or symbols that evaluate to constants. The enclosing angle brackets (< >) are required, even if no value is given. If no value for a field is given, the default value for that field is used. In the next example, assume the following record definition:

```
encode    RECORD hi:4, mid:3, lo:3
```

Example

```
rec1    encode <3,2,1>
        mov     ax,rec1
```

In this example, a constant with the value 209 (0D1h) is moved into the **AX** register. The following bit diagram illustrates how the value is obtained:



Using record operands is similar to declaring a record and then using the declared data except that, in using record operands, you are using constant data. See Section 4.6.2 for information on declaring record data.

5.2.11 Record-Field Operands

Syntax

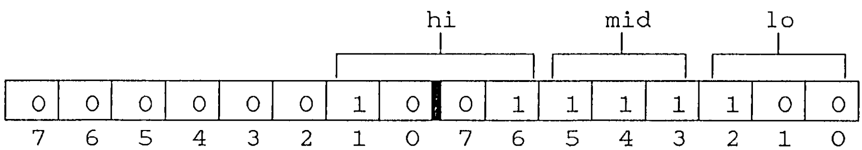
record-fieldname

The record-field operand represents the location of a field in its corresponding record. The operand evaluates to the bit position of the low-order bit in the field and can be used as a constant operand.

The *record-fieldname* must be the name of a previously defined record field. In the next example, assume the following record definition and declaration:

```
encode   RECORD   hi:4, mid:3, lo:3
rec1     encode   <9,7,4>
```

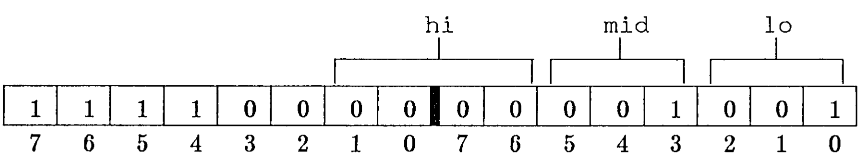
At this point `rec1` has a value of 636 (27Ch), shown in this bit diagram:



Example

```
        mov     cl,hi
        mov     dx,rec1
        ror     dx,cl
        mov     rec1,dx
```

This example copies 6, the shift count for `hi`, to register **CL**. The contents of `rec1` are copied to **DX**. The shift count of field three (`hi`) is then used to rotate the value of `rec1` so that the value of `hi` is now at the lowest bit. The new value is then put back into `rec1`. At this point `rec1` has a value of 61449 (0F009h), as shown in the bit diagram below.



5.3 Operators and Expressions

An expression is a combination of operands and operators that evaluates to a single value. Operands in expressions can include any of the operands described in this chapter. The result of an expression can be a value or a memory location, depending on the types of operands and operators used.

The assembler provides a variety of operators. Arithmetic, shift, relational, and bitwise operators manipulate and compare the values of operands. Attribute operators manipulate the attributes of operands, such as their type, address, and size.

Sections 5.3.1–5.3.4 describe the arithmetic, relational, and logical operators in detail. Attribute operators are described in Sections 5.3.5–5.3.19. In addition to the operators described here, you can use the **DUP** operator (Section 4.3.6) and the special macro operators (Section 8.3).

5.3.1 Arithmetic Operators

Syntax

*expression1***expression2*
expression1/*expression2*
*expression1*MOD*expression2*
expression1+*expression2*
expression1–*expression2*
+*expression*
–*expression*

Arithmetic operators provide the common mathematical operations. Table 5.3 lists the operators and their meanings.

Table 5.3

Arithmetic Operators

Operator	Meaning
+	Positive (unary)
–	Negative (unary)
*	Multiplication
/	Integer division
MOD	Remainder after division (modulus)
+	Addition
–	Subtraction

For all arithmetic operators except + and –, *expression1* and *expression2* must be integer numbers. The + operator can be used to add an integer number to a relocatable memory operand. The – operator can be used to subtract an integer number from a relocatable memory operand. The – operator can also be used to subtract one relocatable operand from another, but only if the operands refer to locations within the same segment. The result is an absolute value.

Note

The unary plus and minus (used to designate positive or negative numbers) are not the same as the binary plus and minus (used to designate addition or subtraction). The unary plus and minus have a higher level of precedence, as shown in Table 5.7 in Section 5.4.

Examples

```
14  *  4                ; Equals  56
14  /  4                ; Equals  3
14  MOD  4              ; Equals  2
14  +  4                ; Equals  18
14  -  4                ; Equals  10
14  -  +4               ; Equals  10
14  -  -4               ; Equals  18
alpha + 5               ; Add 5 to alpha's offset
```

```
alpha - 5                ; Subtract 5 from alpha's offset
alpha - beta             ; Subtract beta's offset from alpha's
```

5.3.2 SHR and SHL Operators

Syntax

expression **SHR** *count*
expression **SHL** *count*

The **SHR** and **SHL** operators shift *expression* right or left by *count* number of bits. Bits shifted off the end of the expression are lost. If the count is greater than or equal to 16, the result is 0. The bits will be shifted by 8 or 16 bits, depending on whether the value being shifted is a word or a byte.

Note

Do not confuse the assembler's **SHR** and **SHL** operators with the processor instructions having the same names.

Examples

```
mov ax,01110111b SHL 3          ; Move 0000001110111000b
mov ah,01110111b SHR 3          ; Move 00001110b
```

Notice that 16 bits are shifted into a word register (ax) in the first example. In the second example, only 8 bits are shifted because the register (ah) holds only 1 byte.

5.3.3 Relational Operators

Syntax

expression1 EQ *expression2*
expression1 NE *expression2*
expression1 LT *expression2*
expression1 LE *expression2*
expression1 GT *expression2*
expression1 GE *expression2*

The relational operators compare *expression1* and *expression2* and return true (0FFFFh) if the condition specified by the operator is satisfied, or false (0000h) if it is not. The expressions must resolve to absolute values. Table 5.4 lists the operators and the values they return if the specified condition is satisfied.

Table 5.4

Relational Operators

Operator	Returned Value
EQ	True (0FFFh) if expressions are equal.
NE	True (0FFFh) if expressions are not equal.
LT	True (0FFFh) if left expression is less than right.
LE	True (0FFFh) if left expression is less than or equal to right.
GT	True (0FFFh) if left expression is greater than right.
GE	True (0FFFh) if left expression is greater than or equal to right.

Relational operators are typically used with conditional directives and conditional instructions to direct program control.

Note

The **EQ** and **NE** operators treat their arguments as 16-bit numbers. Numbers specified with the 16th bit on are considered negative (0FFFFh is -1). Therefore, the expression `-1 EQ 0FFFFh` is true, while the expression `-1 NE 0FFFFh` is false.

The **LT**, **LE**, **GT**, and **GE** operators treat their arguments as 17-bit numbers, where the 17th bit specifies the sign. Therefore, 0FFFFh is the largest positive unsigned number (65535); it is not –1. The expression `1 GT -1` is true (0FFFFh), while the expression `1 GT 0FFFFh` is false (0).

Examples

```
1    EQ    0              ; False
1    NE    0              ; True
1    LT    0              ; False
1    LE    0              ; False
1    GT    0              ; True
1    GE    0              ; True
```

5.3.4 Bitwise Operators

Syntax

NOT *expression*
expression1 **AND** *expression2*
expression1 **OR** *expression2*
expression1 **XOR** *expression2*

The logical operators perform bitwise operations on expressions. In a bit-wise operation, the operation is performed on each bit in an expression rather than on the expression as a whole. The expressions must resolve to absolute values.

Table 5.5 lists the logical operators and their meanings:

Table 5.5

Logical Operators

Operator	Meaning
NOT	Inverse
AND	Boolean AND
OR	Boolean OR
XOR	Boolean exclusive OR

Examples

```
NOT    11110000b          ; Equals 1111111100001111b or 00001111b
01010101b    AND    11110000b       ; Equals 01010000b
01010101b    OR     11110000b       ; Equals 11110101b
01010101b    XOR    11110000b       ; Equals 10100101b
```

5.3.5 Index Operator

Syntax

⟦*expression1*⟧[*expression2*]

The index operator, [ ], adds the value of *expression1* to *expression2*. This operator is identical to the + operator, except that *expression1* is optional.

If *expression1* is given, the expression must appear to the left of the operator. It can be any integer value, absolute symbol, or relocatable operand. If no *expression1* is given, the integer value 0 is assumed. If *expression1* is a relocatable operand, *expression2* must be an integer value or absolute symbol. Otherwise, *expression2* can be any integer value, absolute symbol, or relocatable operand.

The index operator is typically used to index elements of an array, such as individual characters in a character string.

Examples

```
mov   al,string[3]      ; Move 4th element of string
mov   ax,array[4]       ; Move 5th element of array
mov   string[last],al   ; Move into LAST element of string
mov   cx,DGROUP:[1]     ; Move 2nd byte of DGROUP
```

Note that the last example is identical to the following statement:

```
mov   cx, dgroup:1.
```

5.3.6 PTR Operator

Syntax

type **PTR** *expression*

The **PTR** operator forces the variable or label given by *expression* to be treated as a variable or label having the type given by *type*. The type must be one of the following names or values:

Type	Value
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10
NEAR	0FFFFh
FAR	0FFFEh

The expression can be any operand. The **BYTE**, **WORD**, and **DWORD** types can be used with memory operands only. The **NEAR** and **FAR** types can be used with labels only.

The **PTR** operator is typically used with forward references to explicitly define what size or distance a reference has. If it is not used, the assembler assumes a default size or distance for the reference. The **PTR** operator is also used to enable instructions to access variables in ways that would otherwise generate errors. For example, you could use the **PTR** operator to access the high-order byte of a **WORD** size variable.

Section 5.6 discusses how the **PTR** operator can be used to avoid errors associated with strong type checking. These errors include `Illegal size for item` and `Operand types must match.`

Examples

```
call        FAR PTR subrout3
mov         BYTE PTR [array],1
add         al,BYTE PTR [full_word]
```

In these examples the **PTR** operator overrides a previous data declaration. The procedure `subrout3` might have been declared **NEAR**, while `array` and `full_word` could have been declared with the **DW** directive.

5.3.7 Segment-Override Operator

Syntax

*segmentregister***:***expression*
*segmentname***:***expression*
*groupname***:***expression*

The segment-override operator (**:**) forces the address of a given variable or label to be computed using the beginning of the given *segmentregister*, *segmentname*, or *groupname*. If either *segmentname* or *groupname* is given, the name must have been assigned to a segment register with a previous **ASSUME** directive and defined using a **SEGMENT** or **GROUP** directive. The *expression* can be an absolute symbol or relocatable operand. The *segmentregister* must be **CS**, **DS**, **SS**, or **ES**.

By default, the effective address of a memory operand is computed relative to the **DS**, **SS**, or **ES** register, depending on the instruction and operand type. Similarly, all labels are assumed to be **NEAR**. These default types can be overridden using the segment-override operator.

Examples

```
mov     ax,es:[bx][si]
mov     _TEXT:far_label,ax
mov     ax,DGROUP:variable
mov     al,cs:0001H
```

5.3.8 Structure Field-Name Operator

Syntax

*variable***.***field*

The structure field-name operator (**.**) is used to designate a field within a structure. The *variable* is an operand (often a previously declared structure variable) and *field* is the name of a field within a structure. This operator is equivalent to the addition operator (+) in based or indexed operands.

Example

```
inc     month.day
mov     time.min,0
mov     [bx].dest
```

5.3.9 SHORT Operator

Syntax

SHORT *label*

The **SHORT** operator sets the type of the given *label* to **SHORT**. Short labels can be used in **JMP** instructions whenever the distance from the label to the instruction is not more than 127 bytes. Instructions using short labels are 1 byte smaller than identical instructions using near labels.

Example

```
jmp     SHORT do_again   ; Jump less than 128 bytes
```

5.3.10 THIS Operator

Syntax

THIS *type*

The **THIS** operator creates an operand whose offset and segment values are equal to the current location-counter value and whose type is given by *type*. The *type* can be any one of the following:

BYTE

WORD

DWORD

QWORD

TBYTE

NEAR

FAR

The **THIS** operator is typically used with the **EQU** or equal-sign (=) directive to create labels and variables. This is similar to using the **LABEL** directive to create labels and variables.

Examples

```
tag   EQU      THIS BYTE
```

The preceding example is equivalent to the statement `tag LABEL BYTE`.

```
check =        THIS NEAR
```

The final example is equivalent to the statement `check LABEL NEAR`.

5.3.11 HIGH and LOW Operators

Syntax

HIGH *expression*
LOW *expression*

The **HIGH** and **LOW** operators return the high and low 8 bits, respectively, of *expression*. The **HIGH** operator returns the high-order 8 bits of *expression*; the **LOW** operator returns the low-order 8 bits. The expression can be any value.

Examples

```
    mov     ah,HIGH word_value       ; Move high byte of word_value
    mov     al,LOW OABCDh            ; Move OCDh
```

5.3.12 SEG Operator

Syntax

SEG *expression*

The **SEG** operator returns the segment value of *expression*. The expression can be any label, variable, segment name, group name, or other symbol.

Examples

```
mov     ax,SEG variable_name
mov     ax,SEG label_name
```

5.3.13 OFFSET Operator

Syntax

OFFSET *expression*

The **OFFSET** operator returns the offset of *expression*. The expression can be any label, variable, segment name, or other symbol. The returned value is the number of bytes between the item and the beginning of the segment in which it is defined. For a segment name, the returned value is the offset from the start of the segment to the most recent byte generated for that segment.

The segment-override operator (:) can be used to force **OFFSET** to return the number of bytes between the item in *expression* and the beginning of a named segment or group. This is the method used to generate valid offsets for items in a group. See the second example below.

Examples

```
mov     bx,OFFSET subrout3
mov     bx,OFFSET dgroup:array
```

The returned value is always a relative value that is subject to change by the linker when the program is actually linked.

5.3.14 TYPE Operator

Syntax

TYPE *expression*

The **TYPE** operator returns a number representing the type of *expression*. If *expression* is a variable, the operator returns the size of the operand in bytes. If *expression* is a label, the operator returns 0FFFFh if the label is **NEAR**, and 0FFFEh if the label is **FAR**. Note that the returned value can be used to specify the type for a **PTR** operator, as in the second of the following two examples.

Examples

```
mov     ax,TYPE array
jmp     (TYPE get_loc) PTR destiny
```

5.3.15 .TYPE Operator

Syntax

.TYPE *expression*

The **.TYPE** operator returns a byte that defines the mode and scope of *expression*. If *expression* is not valid, **.TYPE** returns a 0.

Table 5.6 lists the variable's attributes as returned in bits 0, 1, 5, and 7.

Table 5.6

.TYPE Operator and Variable Attributes

Bit Position	If Bit = 0	If Bit = 1
0	Not program-related	Program-related
1	Not data-related	Data-related
5	Not defined	Defined
7	Local or public scope	External scope

If both the scope bit and defined bit are zero, *expression* is not valid.

The **.TYPE** operator is typically used with conditional directives, where an argument may need to be tested in order to make a decision regarding program flow.

Example

```
x     DB     12
z     EQU    .TYPE x
```

This example sets `z` to 22h (00100010b). Bit 0 is not set in `z` because `x` is not program-related. Bit 1 is set because `x` is data-related. Bit 5 is set because x is defined. Bit 7 is not set because x is local. The remaining bits are never set.

5.3.16 LENGTH Operator

Syntax

LENGTH *variable*

The **LENGTH** operator returns the number of **BYTE**, **WORD**, **DWORD**, **QWORD**, or **TBYTE** elements in *variable*. The size of each element depends on the variable's defined type.

Only variables defined using the **DUP** operator return values that are greater than 1. The returned value is always the number preceding the first **DUP** operator.

In the next two examples, assume the following definitions:

```
array  DW    100   DUP(1)
table  DW    100   DUP(1,10 DUP(?))
```

Examples

```
        mov     cx,LENGTH array
```

In the preceding example, LENGTH returns 100.

```
        mov     cx,LENGTH table
```

In the final example, **LENGTH** returns 100. The returned value does not depend on any nested **DUP** operators.

5.3.17 SIZE Operator

Syntax

SIZE *variable*

The **SIZE** operator returns the total number of bytes allocated for *variable*. The returned value is equal to the value of **LENGTH** times the value of **TYPE**.

In the next example, assume the following definition:

```
array DW    100    DUP(1)
```

Example

```
      mov   bx,SIZE array
```

In this example, **SIZE** returns 200.

5.3.18 WIDTH Operator

Syntax

WIDTH *recordfieldname*|*record*

The **WIDTH** operator returns the width (in bits) of the given record field or record. The *recordfieldname* must be the name of a field defined in a record. The *record* must be the name of a record.

In the next examples, assume the following record definition and record declaration:

```
rtype  RECORD field1:3,field2:6,field3:7
rec1   rtype <>
```

Examples

```
wid1     =  WIDTH field1       ; Equals 3
wid2     =  WIDTH field2       ; Equals 6
wid3     =  WIDTH field3       ; Equals 7
widrec   =  WIDTH rtype        ; Equals 16
```

Remember, the field name represents the bit count. For example, `field1` equals 13 (the width of `field2` plus the width of `field3`) while `WIDTH field1` equals 3.

5.3.19 MASK Operator

Syntax

MASK *recordfieldname*|*record*

The **MASK** operator returns a bit mask for the bit positions in a record occupied by the given record field. A bit in the mask contains a 1 if that bit corresponds to a field bit. All other bits contain 0.

The *recordfieldname* must be the name of a field defined in a record.

In the next example, assume the following record definition and record declaration:

```
rtype   RECORD field1:3,field2:6,field3:7
rec1    rtype <>
```

Example

```
m1   =  MASK field1  ; Equals E000h  (1110000000000000b)
m2   =  MASK field2  ; Equals 1F80h  (1111110000000b)
m3   =  MASK field3  ; Equals 007Fh  (1111111b)
mrec =  MASK rtype   ; Equals 0FFFFh (1111111111111111b)
```

5.4 Expression Evaluation and Precedence

Expressions are evaluated according to the rules of operator precedence and order. Operations of highest precedence are performed first. Operations of equal precedence are performed from left to right. This default order of evaluation can be overridden by using enclosing parentheses. Operations in parentheses are always performed before any adjacent operations. Table 5.7 lists the precedence of all operators. Operators on the same line have equal precedence.

Table 5.7

Operator Precedence

Precedence	Operators
(Highest)	
1	LENGTH, SIZE, WIDTH, MASK, (), [], <>
2	. (structure field-name operator)
3	:
4	PTR, OFFSET, SEG, TYPE, THIS
5	HIGH, LOW
6	+,– (unary)
7	*,/, MOD, SHL, SHR
8	+, – (binary)
9	EQ, NE, LT, LE, GT, GE
10	NOT
11	AND
12	OR, XOR
13	SHORT, .TYPE
(Lowest)	

Examples

```
8 / 4 * 2               ; Equals 4
8 / (4 * 2)             ; Equals 1
8 + 4 * 2               ; Equals 16
(8 + 4) * 2             ; Equals 24
8 EQ 4 AND 2 LT 3       ; Equals 0000h (false)
8 EQ 4 OR 2 LT 3        ; Equals 0FFFFh (true)
```

5.5 Forward References

Although the assembler permits forward references to labels, variable names, segment names, and other symbols, such references can lead to assembly errors if not used properly. A forward reference is any use of a name before it has been declared. For example, in the **JMP** instruction below, the label `target` is a forward reference.

```
        jmp     target
        mov     ax, 0
target:
```

Whenever the assembler encounters an undefined name in Pass 1, it assumes that the name is a forward reference. If only a name is given, the assembler makes assumptions about that name's type and segment register, and uses these assumptions to generate code or data for the statement. For example, in the **JMP** instruction above, **MASM** assumes that `target` is an instruction label having **NEAR** type. It generates 3 bytes of instruction code for the instruction.

The assembler bases its assumptions on the statement containing the forward reference. Errors can occur when these assumptions are incorrect. For example, if `target` were really a **FAR** label and not a **NEAR** label, the assumption made by the assembler in Pass 1 would cause a phase error. In other words, the assembler would generate 5 bytes of instruction code for the **JMP** instruction in Pass 2 but only 3 in Pass 1.

To avoid errors with forward references, the segment override (**:**), **PTR**, and **SHORT** operators should be used whenever necessary to override the assumptions made by the assembler. The following guidelines list situations in which these operators should be used:

- If a forward reference is a variable that is relative to the **ES**, **SS**, or **CS** register, then use the segment-override operator (**:**) to specify the variable's segment register, segment, or group.

  Examples

  ```
  mov     ax,ss:stacktop
  inc     data:time[1]
  add     ax,dgroup:_I
  ```

  If the segment-override operator is not used, the assembler assumes that the variable is relative to the **DS** register.

- If a forward reference is an instruction label in a **JMP** instruction, then use the **SHORT** operator if the instruction is less than 128 bytes from the point of reference.

  Example

  ```
  jmp     SHORT target
  ```

  If **SHORT** is not used, the assembler assumes that the instruction is greater than 128 bytes away. This does not cause an error, but it does cause the assembler to generate an extra, and unnecessary, **NOP** instruction.

- If a forward reference is an instruction label in a **CALL** or **JMP** instruction, then use the **PTR** operator to specify the label's type.

Examples

```
call    FAR PTR print
jmp     FAR PTR exit
```

The assembler assumes that the label has **NEAR** type, so **PTR** need not be used for **NEAR** labels. If the label has **FAR** type, however, and **FAR PTR** is not used, a phase error will result.

- If the forward reference is a segment name with a segment-override operator (:), use the **GROUP** statement to associate the segment name with a group name, then use the **ASSUME** statement to associate the group name with a segment register.

Example

```
dgroup  GROUP   stack
        ASSUME  ss:dgroup

code    SEGMENT
        .
        .
        .
        mov     ax,stack:stacktop
        .
        .
        .
```

If you do not associate a group with the segment name, the assembler may ignore the segment override and use the default segment register for the variable. This usually results in a phase error in Pass 2.

5.6 Strong Typing for Memory Operands

The assembler carries out strict syntax checks for all instruction statements, including strong typing for operands that refer to memory locations. This means that any relocatable operand used in an instruction that operates on an implied data type must either have that type, or have an explicit type override (**PTR** operator).

For example, in the following program segment, the variable `string` is incorrectly used in a move instruction.

```
string DB   "A message."

       mov  ax,string[1]
```

This statement will result in an `Operand types must match` error message since `string` has **BYTE** type and the instruction expects a variable having **WORD** type.

To avoid this error, the **PTR** operator must be used to override the variable's type. The following statement will assemble correctly and execute as expected:

```
mov    ax,WORD PTR string[1]
```

Note

Many assembly-language program listings in books and magazines are written for assemblers with weak typing for operands. These programs may produce error messages such as `Illegal size for item` or `Operand types must match` when assembled as listed using the Microsoft Macro Assembler. You can correct lines that produce errors by using the **PTR** operator to assign the correct size to variables.

Chapter 6
Global Declarations

6.1 Introduction

The global-declaration directives allow you to define labels, variables, and absolute symbols that can be accessed globally, that is, from all modules in a program. Global declarations transform "local" symbols (labels, variables, and other symbols that are specific to the source files in which they are defined) into "global" symbols that are available to all other modules of the program.

The two global-declaration directives are **PUBLIC** and **EXTRN**. The **PUBLIC** directive is used in public declarations, which transform locally defined symbols into global symbols, making them available to other modules. The **EXTRN** directive is used in external declarations, making a global symbol's name and type known in a source file so that the global symbol may be used in that file. Every global symbol must have a public declaration in exactly one source file of the program. A global symbol can have external declarations in any number of other source files. Sections 6.2–6.4 describe and demonstrate the global-declaration directives in detail.

6.2 PUBLIC Directive

Syntax

PUBLIC *name*,,,

The **PUBLIC** directive makes the variable, label, or absolute symbol specified by *name* available to all other modules in the program. The name must be the name of a variable, label, or absolute symbol defined within the current source file. Absolute symbols, if given, can only represent 1- or 2-byte integer or string values.

The assembler converts all lowercase letters in *name* to uppercase before copying the name to the object file. The /ML and /MX options can be used in the **MASM** command line to direct the assembler to preserve lowercase letters when copying public and external symbols to the object file. Sections 2.3.7 and 2.3.8 of the *Microsoft Macro Assembler User's Guide* describe the /ML and /MX options.

Symbols must be declared public before they can be used for symbolic debugging. See Section 4.2 of the *Microsoft Macro Assembler User's Guide* for details on how to prepare and use symbol files with **SYMDEB**.

Example

```
       PUBLIC   true,status,start,clear
true   =        0FFFFH
status DB       1
start  LABEL    FAR
clear  PROC     NEAR
```

The values declared public in this example include an absolute symbol, a variable, a label, and a procedure.

6.3 EXTRN Directive

Syntax

EXTRN *name:type,,,*

The **EXTRN** directive defines an external variable, label, or symbol of the specified *name* and *type*. An external item is any variable, label, or symbol that has been declared with a **PUBLIC** directive in another module of the program. The *type* must match the type given to the item in its actual definition. It can be any one of the following:

BYTE

WORD

DWORD

QWORD

TBYTE

NEAR

FAR

ABS

The **ABS** type is for symbols that represent absolute numbers.

Although the actual address is not determined until the object files are linked, the assembler may assume a default segment for the external item, based on where the **EXTRN** directive is placed in the module. If the directive is placed inside a segment, the external item is assumed to be relative to that segment, and the item's public declaration (in some other module) must be in a segment having the same name and attributes. If the directive is outside all segments, no assumption is made about what segment the item is relative to, and the item's public declaration can be in any segment in any module. In either case, the segment-override operator (:) can be used to override the default segment of an external variable or label.

Example

```
EXTRN       tagn:near
EXTRN       var1:word,var2:dword
```

6.4 Program Example

The following source files illustrate a program that uses public and external declarations to access instruction labels. The program consists of two modules, named `main` and `task`. The `main` module is the program's initializing module. Execution starts at the instruction labeled `start` in `main`, and passes to the instruction labeled `print` in `task`. An MS-DOS system call in the `task` module is used to print `Hello` on the screen. Execution then returns to the instruction labeled `exit` in the `main` module.

Main Module

```
        NAME    main
        PUBLIC  exit
        EXTRN   print:near

stack   SEGMENT word stack 'STACK'
        DW      64 DUP(?)
stack   ENDS

data    SEGMENT word public 'DATA'
data    ENDS

code    SEGMENT byte public 'CODE'
        ASSUME  cs:code,ds:data
start:
        mov     ax,data             ; Load segment location
        mov     ds,ax               ;   into DS register
        jmp     print               ; Go to PRINT in other module
```

```
exit:
        mov     ah, 4Ch             ; Call terminate function
        int     21h
code    ENDS
        END     start
```

Task Module

```
        NAME    task
        PUBLIC  print
        EXTRN   exit:near

data    SEGMENT word public 'DATA'
string  DB      "Hello",13,10,"$"
data    ENDS

code    SEGMENT byte public 'CODE'
        ASSUME  cs:code, ds:data
print:
        mov     dx,OFFSET string ; Load string location
        mov     ah,09h           ; Call string display function
        int     21h
        jmp     exit             ; Go back to other module
code    ENDS
        END
```

In this example, the symbol `exit` is declared public in the `main` module so that it can be accessed from another source module (`task` in the example). The `main` module also contains an external declaration of the symbol `print`. This declaration defines `print` to be a near label so that it can be accessed from the `main` module, even though it is assumed to be located and declared public in another source module. A **JMP** instruction later in the module has this label as its destination.

The symbol `print` is declared public in the `task` module so that it can be accessed from another module (`main` in the example). The symbol `exit` is defined as a near label so that it can be accessed from this module, even though it is assumed to be located and declared public in the other module.

Before this program can be executed, these source files must be assembled individually, then linked together using **LINK**.

Chapter 7
Conditional Directives

7.1 Introduction

The Microsoft Macro Assembler provides two types of conditional directives. Conditional-assembly directives test for a specified condition and assemble a block of statements if the condition is true. Conditional error directives test for a specified condition and generate an error if the condition is true.

Both kinds of conditional directives only test assembly-time conditions. They cannot test run-time conditions since these are not known until an executable program is run. Only expressions that evaluate to constants during assembly can be compared or tested.

Since macros and conditional-assembly directives are often used together, you may need to refer to Chapter 8 to understand some of the examples in this chapter. In particular, conditional directives are frequently used with the special macro operators described in Section 8.3.

7.2 Conditional-Assembly Directives

The conditional-assembly directives include the following:

IF

IFE

IF1

IF2

IFDEF

IFNDEF

IFB

IFNB

IFIDN

IFDIF

ELSE

ENDIF

The **IF** directives and the **ENDIF** and **ELSE** directives can be used to enclose the statements to be considered for conditional assembly. The conditional block takes the following form:

IF
statements
[**ELSE**
statements]
ENDIF

The *statements* following **IF** can be any valid statements, including other conditional blocks. The **ELSE** directive and its *statements* are optional. **ENDIF** ends the block.

The statements in the conditional block are assembled only if the condition specified by the corresponding **IF** directive is satisfied. If the conditional block contains an **ELSE** directive, only the statements up to the **ELSE** directive will be assembled. The statements following the **ELSE** directive are assembled only if the **IF** condition is not met. An **ENDIF** directive must mark the end of any conditional-assembly block. No more than one **ELSE** directive is allowed for each **IF** directive.

IF directives can be nested up to 255 levels. To avoid ambiguity, a nested **ELSE** directive always belongs to the nearest preceding **IF** directive that does not have its own **ELSE**.

7.2.1 IF and IFE Directives

Syntax

IF *expression*
IFE *expression*

The **IF** and **IFE** directives test the value of an *expression*. The **IF** directive grants assembly if the value of *expression* is true (nonzero). The **IFE** directive grants assembly if the value of *expression* is false (0). The *expression* must resolve to an absolute value and must not contain forward references.

Example

```
IF      debug
        EXTRN dump:FAR
        EXTRN trace:FAR
        EXTRN breakpoint:FAR
ENDIF
```

In this example, the variables within the block will only be declared external if the symbol `debug` evaluates to true (nonzero).

7.2.2 IF1 and IF2 Directives

Syntax

IF1
IF2

The **IF1** and **IF2** directives test the current assembly pass. The **IF1** directive grants assembly only on Pass 1. **IF2** grants assembly only on Pass 2. The directives take no arguments.

Example

```
IF1
      %OUT Beginning Pass 1
ELSE
      %OUT Beginning Pass 2
ENDIF
```

7.2.3 IFDEF and IFNDEF Directives

Syntax

IFDEF *name*
IFNDEF *name*

The **IFDEF** and **IFNDEF** directives test whether or not the given *name* has been defined. The **IFDEF** directive grants assembly only if *name* is a label, variable, or symbol. The **IFNDEF** directive grants assembly if *name* has not yet been defined.

The name can be any valid name. Note that if *name* is a forward reference, it is considered undefined on Pass 1, but defined on Pass 2.

Example

```
IFDEF   buffer
        buf1   DB 10 DUP(?)
ENDIF
```

In this example, `bufl` is allocated only if `buffer` has been previously defined. One way to use this conditional block would be to leave `buffer` undefined in the source file and define it if you needed it by using the /D*symbol* option when you start **MASM**. For example, if the conditional block is in `test.asm`, you could start the assembler with the command line:

```
MASM test /Dbuffer;
```

The symbol `buffer` would be defined, and as a result the conditional-assembly block would allocate `bufl`. However, if you didn't need `bufl`, you could use the command line:

```
MASM test;
```

7.2.4 IFB and IFNB Directives

Syntax

IFB <*argument*>
IFNB <*argument*>

The **IFB** and **IFNB** directives test *argument.* The **IFB** directive grants assembly if *argument* is blank. The **IFNB** directive grants assembly if *argument* is not blank. The arguments can be any name, number, or expression. The angle brackets (< >) are required.

The **IFB** and **IFNB** directives are intended for use in macro definitions. They can control conditional-assembly of statements in the macro, based on the parameters passed in the macro call. In such cases, *argument* should be one of the dummy parameters listed by the **MACRO** directive.

Example

```
pushall     MACRO    reg1,reg2,reg3,reg4,reg5,reg6
            IFNB     <reg1>           ;; If parameter not blank
                     push    reg1     ;;   push one register and repeat
                     pushall reg2,reg3,reg4,reg5,reg6
            ENDIF
            ENDM

pushall     ax,bx,si,ds
pushall     cs,es
```

In this example, `pushall` is a recursive macro that continues to call itself until it encounters a blank argument. Any register or list of registers (consisting of up to six registers) can be passed to the macro for pushing.

7.2.5 IFIDN and IFDIF Directives

Syntax

IFIDN <*argument1*>,<*argument2*>
IFDIF <*argument1*>,<*argument2*>

The **IFIDN** and **IFDIF** directives compare *argument1* and *argument2*. The **IFIDN** directive grants assembly if the arguments are identical. The **IFDIF** directive grants assembly if the arguments are different. The arguments can be any names, numbers, or expressions. To be identical, each character in *argument1* must match the corresponding character in *argument2*. Case is significant. The angle brackets (< >) are required. The arguments must be separated by a comma (,).

The **IFIDN** and **IFDIF** directives are intended for use in macro definitions. They can control conditional assembly of macro statements, based on the parameters passed in the macro call. In such cases, the arguments should be dummy parameters listed by the **MACRO** directive.

Example

```
divide   MACRO    numerator, denominator
         IFDIF    <denominator>,<0>   ;; If not dividing by zero
         mov      ax, numerator       ;;   divide AX by BX
         mov      bx, denominator
         div      bx                  ;; Result in accumulator
         ENDIF
         ENDM

divide   6,%test
```

In this example, a macro uses the **IFDIF** directive to check against dividing by a constant that evaluates to 0. The macro is then called, using a percent sign (%) on the second parameter so that the value of the parameter, rather than its name, will be evaluated. See Section 8.3.4 for a discussion of the expression (%) operator.

If the parameter `test` was previously defined with the statement

```
test        EQU     0
```

then the condition fails and the code in the block will not be assembled. However, if the parameter `test` was defined with the statement

```
test        DW      0
```

error 42, `Constant was expected`, will be generated. This is because the assembler has no way of knowing the run-time value of `test`. Remember, conditional directives can only evaluate constants that are known at assembly time.

7.3 Conditional Error Directives

Conditional error directives can be used to debug programs and check for assembly-time errors. By inserting a conditional error directive at a key point in your code, you can test assembly-time conditions at that point. You can also use conditional error directives to test for boundary conditions in macros.

The conditional error directives, and the errors they produce, are listed in Table 7.1.

Table 7.1

Conditional Error Directives

Directive	Number	Message
.ERR1	87	`Forced error - pass1`
.ERR2	88	`Forced error - pass2`
.ERR	89	`Forced error`
.ERRE	90	`Forced error - expression equals 0`
.ERRNZ	91	`Forced error - expression not equal 0`
.ERRNDEF	92	`Forced error - symbol not defined`
.ERRDEF	93	`Forced error - symbol defined`
.ERRB	94	`Forced error - string blank`
.ERRNB	95	`Forced error - string not blank`
.ERRIDN	96	`Forced error - strings identical`
.ERRDIF	97	`Forced error - strings different`

Like other fatal assembler errors, those generated by conditional error directives cause the assembler to return exit code 7. If a fatal error is encountered during assembly, **MASM** will delete the object module. All conditional error directives except **ERR1** generate fatal errors.

7.3.1 .ERR, .ERR1, and .ERR2 Directives

Syntax

.ERR
.ERR1
.ERR2

The **.ERR**, **.ERR1**, and **.ERR2** directives force an error at the points at which they occur in the source file. The **.ERR** directive forces an error regardless of the pass, while the **.ERR1** and **.ERR2** directives force the error only on their respective passes. The **.ERR1** directive only appears on the screen or in the listing file if you use the **/D** option to request a Pass 1 listing. Unlike other conditional error directives, it is not a fatal error.

You can place these directives within conditional-assembly blocks or macros to see which blocks are being expanded.

Example

```
IFDEF dos
      .
      .
      .
ELSE
      IFDEF   xenix
      .
      .
      .
      ELSE
      .ERR
      ENDIF
ENDIF
```

This example makes sure that either the symbol `dos` or the symbol `xenix` is defined. If neither is defined, the nested **ELSE** condition is assembled and an error message is generated. Since the **.ERR** directive is used, an error would be generated on each pass. You could use the **.ERR2** directive if you wanted only a fatal error, or you could use the **.ERR1** directive if you wanted only a warning error.

7.3.2 .ERRE and .ERRNZ Directives

Syntax

.ERRE *expression*
.ERRNZ *expression*

The **.ERRE** and **.ERRNZ** directives test the value of an *expression.* The **.ERRE** directive generates an error if the *expression* is false (0). The **.ERRNZ** directive generates an error if the *expression* is true (nonzero). The *expression* must resolve to an absolute value and must not contain forward references.

Example

```
buffer  MACRO   count,bname
        .ERRE   count LE 128      ;; Allocate memory, but
        bname   DB    count DUP(0);;   no more than 128 bytes
        ENDM

buffer  128,buf1          ; Data allocated - no error
buffer  129,buf2          ; Error generated
```

In this example, the **.ERRE** directive is used to check the boundaries of a parameter passed to the macro `buffer`. If `count` is less than or equal to 128, the expression being tested by the error directive will be true (nonzero) and no error will be generated. If `count` is greater than 128, the expression will be false (0) and the error will be generated.

7.3.3 .ERRDEF and .ERRNDEF Directives

Syntax

.ERRDEF *name*
.ERRNDEF *name*

The **.ERRDEF** and **.ERRNDEF** directives test whether or not *name* has been defined. The **.ERRDEF** directive produces an error if *name* is defined as a label, variable, or symbol. The **.ERRNDEF** directive produces an error if *name* has not yet been defined. If *name* is a forward reference, it is considered undefined on Pass 1, but defined on Pass 2.

Example

```
.ERRDEF   symbol
IFDEF     config1
          .
          .symbol  EQU  0
          .
ENDIF
IFDEF     config2
          .
          .symbol  EQU  1
          .
ENDIF
.ERRNDEF symbol
```

In this example, the **.ERRDEF** directive at the beginning of the conditional blocks makes sure that `symbol` has not been defined before entering the blocks. The **.ERRNDEF** directive at the end ensures that `symbol` was defined somewhere within the blocks.

7.3.4 .ERRB and .ERRNB Directives

Syntax

.ERRB <*string*>
.ERRNB <*string*>

The **.ERRB** and **.ERRNB** directives test the given *string*. The **.ERRB** directive generates an error if *string* is blank. The **.ERRNB** directive generates an error if *string* is not blank. The string can be any name, number, or expression. The angle brackets (< >) are required.

These conditional error directives can be used within macros to test for the existence of parameters.

Example

```
work  MACRO  realarg,testarg
      .ERRB  <realarg>     ;; Error if no parameters
      .ERRNB <testarg>     ;; Error if more than one parameter
      .
      .
      .
      ENDM
```

In this example, error directives are used to make sure that one, and only one, argument is passed to the macro. The **.ERRB** directive generates an error if no argument is passed to the macro. The **.ERRNB** directive generates an error if more than one argument is passed to the macro.

7.3.5 .ERRIDN and .ERRDIF Directives

Syntax

.ERRIDN <*string1*>,<*string2*>
.ERRDIF <*string1*>,<*string2*>

The **.ERRIDN** and **.ERRDIF** directives test whether two strings are identical. The **.ERRIDN** directive generates an error if the strings are identical. The **.ERRDIF** generates an error if the strings are different. The strings can be names, numbers, or expressions. To be identical, each character in *string1* must match the corresponding character in *string2*. String checks are case-sensitive. The angle brackets (< >) are required.

Example

```
addem   MACRO ad1,ad2,sum
        .ERRIDN <ax>,<ad2> ;; Error if ad2 is 'ax'
        .ERRIDN <AX>,<ad2> ;; Error if ad2 is 'AX'
        mov   ax,ad1       ;; Would overwrite if ad2 were AX
        add   ax,ad2
        mov   sum,ax       ;; Sum must be register or memory
        ENDM
```

In this example, the **.ERRIDN** directive is used to protect against passing the **AX** register as the second parameter, because the macro won't work if the **AX** register is passed as the second parameter. Note that the directive is used twice to protect against the two most likely spellings.

Chapter 8
Macro Directives

8.1 Introduction

This chapter explains how to create and use macros in your source files. It discusses the macro directives and the special macro operators. Since macros are closely related to conditional directives, you may need to review Chapter 7 to follow some of the examples in this chapter.

Macro directives enable you to write a named block of source statements, then use that name in your source file to represent the statements. During assembly, **MASM** automatically replaces each occurrence of the macro name with the statements in the macro definition. You can place a block of statements anywhere in your source file any number of times by simply defining a macro block once, then inserting the macro name at each location where you want the macro block to be assembled. You can also pass parameters to macros.

A macro can be defined any place in the source file as long as the definition precedes the first source line that calls that macro. Macros can be kept in a separate file and made available to the program through an **INCLUDE** directive (see Section 9.2).

Often a task can be done by either a macro or procedure. For example, the `Addup` procedure shown in Section 3.10 does the same thing as the `Addup` macro in Section 8.2.1. Macros are expanded on every occurrence of the macro name, so they can increase the length of the executable file if called repeatedly. Procedures take up less space, but the increased overhead of saving and restoring addresses and parameters can make them slower.

8.2 Macro Directives

The macro directives are listed below:

MACRO

ENDM

LOCAL

PURGE

REPT

IRP

IRPC

EXITM

The **MACRO** and **ENDM** directives designate the beginning and end of a macro block. The **LOCAL** directive lets you define labels used only within a macro, and the **PURGE** directive lets you delete previously defined macros. The **EXITM** directive allows you to exit from a macro before all the statements in the block are expanded.

The **REPT**, **IRP**, and **IRPC** directives let you create contiguous blocks of repeated statements. These repeat blocks are frequently placed within macros, but they can also be used independently. You can control the number of repetitions by specifying a number; or by allowing the block to be repeated once for each parameter in a list; or by having the block repeated once for each character in a string.

8.2.1 MACRO and ENDM Directives

Syntax

name **MACRO** ⟦*dummyparameter,,,*⟧
statements
ENDM

The **MACRO** and **ENDM** directives create a macro having *name* and containing the given *statements.*

The name must be a valid name and must be unique. It is used in the source file to invoke the macro. The *dummyparameter* is a name that acts as a placeholder for values to be passed to the macro when it is called. Any number of *dummyparameters* can be specified, but they must all fit on one line. If you give more than one, you must separate them with commas (,). The statements are any valid **MASM** statements, including other macro directives. Any number of statements can be used. The dummy parameters can be used any number of times in these statements.

A macro is "called" any time its name appears in a source file (macro names in comments are ignored). **MASM** copies the statements in the macro definition to the point of the call, replacing any dummy parameters in these statements with actual parameters passed in the call.

Macro definitions can be nested. This means a macro can be defined within another macro. **MASM** does not process nested definitions until the outer macro has been called. Therefore, nested macros cannot be called until the outer macro has been called at least once. Macro definitions can be nested to any depth. Nesting is limited only by the amount of memory available when the source file is assembled.

Macro definitions can contain calls to other macros. These nested macro calls are expanded like any other macro call, but only when the outer macro is called. Macro definitions can also be recursive: they can call themselves, as illustrated in the example in Section 7.2.4.

Example

```
addup   MACRO   ad1,ad2,ad3
        mov     ax, ad1         ;; First parameter in AX
        add     ax, ad2         ;; Add next two parameters
        add     ax, ad3         ;;   and leave sum in AX
        ENDM
```

The preceding example defines a macro named `addup`, which uses three dummy parameters to add three values and leave their sum in the **AX** register. The three dummy parameters will be replaced with actual values when the macro is called.

MASM assembles the statements in the macro only if the macro is called, and only at the point in the source file from which it is called. Thus, all addresses in the assembled code will be relative to the macro call, not the macro definition. The macro definition itself is never assembled.

You must be careful when using the word **MACRO** after the **TITLE**, **SUBTTL**, and **NAME** directives. Since the **MACRO** directive overrides these directives, placing the word macro immediately after these directives would cause the assembler to begin to create macros named **TITLE**, **SUBTTL**, and **NAME**. For example, the line:

```
TITLE Macro File
```

may be intended to give an include file the title "Macro File", but its effect will be to create a macro called `TITLE` that accepts the dummy parameter `File`. Since there will be no corresponding **ENDM** directive, an error will usually result.

To avoid this problem, you should alter the word macro in some way when using it in a title or name. For example, change the spelling or add an underline character (`MAKRO` or `_MACRO`).

Note

MASM replaces all occurrences of a dummy parameter's name, even if you do not intend it to. For example, if you use a register name such as **AX** or **BH** for a dummy parameter, **MASM** replaces all occurrences of that register name when it expands the macro. If the macro definition contains statements that use the register, not the dummy, the macro will be incorrectly expanded.

Note

Macros can be redefined. You need not purge the first macro before redefining it. The new definition automatically replaces the old definition. If you redefine a macro from within the macro itself, make sure there are no lines between the **ENDM** directive of the nested redefinition and the **ENDM** directive of the original macro. The following example may produce incorrect code:

```
dostuff    MACRO
           .
           .
           .
           dostuff    MACRO
                      .
                      .
                      .
                      ENDM
           ;; Comments or statements not allowed
           ENDM
```

To correct the error, remove the line between the **ENDM** directives.

8.2.2 Macro Calls

Syntax

name ⟦*actualparameter,,,*⟧

A macro call directs **MASM** to copy the statements of the macro *name* to the point of call and to replace any dummy parameters in these statements with the corresponding actual parameters. The *name* must be the name of a macro defined earlier in the source file. The *actualparameter* can be any name, number, or other value. Any number of actual parameters can be given, but they must all fit on one line. Multiple parameters must be separated by commas, spaces, or tabs.

MASM replaces the first dummy parameter with the first actual parameter, the second with the second, and so on. If a macro call has more actual parameters than dummy parameters, the extra actual parameters are ignored. If a call has fewer actual parameters than dummy parameters, any remaining dummy parameters are replaced with a null (blank) string. You can use the **IFB**, **IFNB**, **.ERRB**, and **.ERRNB** directives to have your macros check for null strings and take appropriate action. See Sections 7.2.4 and 7.3.4.

If you wish to pass a list of values as a single actual parameter, you must place angle brackets (< >) around the list. The items in the list must be separated by commas (,).

Examples

```
allocblock 1,2,3,4,5
```

The first example passes five numeric parameters to the macro called `allocblock`.

```
allocblock <1,2,3,4,5>
```

The second example passes one parameter to `allocblock`. The parameter is a list of five numbers.

```
addup       bx, 2, count
```

The final example passes three parameters to the macro addup. **MASM** replaces the corresponding dummy parameters with exactly what is typed in the macro call parameters. Assuming that addup is the same macro defined at the end of Section 8.2.1, the assembler would expand the macro to the following code:

```
mov     ax, bx
add     ax, 2
add     ax, count
```

See Section 2.4 of the *Microsoft Macro Assembler User's Guide* for an example of how macros are shown in listing files.

8.2.3 LOCAL Directive

Syntax

LOCAL *dummyname,,,*

The **LOCAL** directive creates unique symbol names for use in macros. The *dummyname* is a name for a placeholder that is to be replaced by a unique name when the macro is expanded. At least one *dummyname* is required. If you give more than one, you must separate the names with commas (,). A *dummyname* can be used in any statement within the macro.

MASM creates a new actual name for the dummy name each time the macro is expanded. The actual name has the following form:

??number

The *number* is a hexadecimal number in the range 0000 to FFFF. Do not give other symbols names in this format, since doing so will produce a label or symbol with multiple definitions. In listings, the dummy name is shown in the macro definition, but the actual names are shown for each expansion of the macro.

The **LOCAL** directive is typically used to create a unique label that will only be used in a macro. Normally, if a macro containing a label is used more than once, **MASM** will display an error message indicating the file contains a label or symbol with multiple definitions, since the same label will appear in both expansions. To avoid this problem, all labels in macros should be dummy names declared with the **LOCAL** directive.

Note

The **LOCAL** directive can be used only in a macro definition, and it must precede all other statements in the definition. If you try to put a comment line or an instruction before the **LOCAL** directive, a warning error will result.

Example

```
power     MACRO    factor,exponent
          LOCAL    again,gotzero     ;; Declare symbols for macro
          mov      cx,exponent       ;; Exponent is count for loop
          mov      ax,1              ;; Multiply by 1 first time
          jcxz     gotzero           ;; Get out if exponent is zero
          mov      bx,factor
again:    mul      bx                ;; Multiply until done
          loop     again
gotzero:
          ENDM
```

In this example, the **LOCAL** directive defines the dummy names `again` and `gotzero`. These names will be replaced with unique names each time the macro is expanded. For example, the first time the macro is called, `again` will be assigned the name `??0000` and `gotzero` will be assigned `??0001`. The second time through `again` will be assigned `??0002` and `gotzero` will be assigned `??0003`, and so on.

8.2.4 PURGE Directive

Syntax

PURGE *macroname*,,,

The **PURGE** directive deletes the current definition of the macro called *macroname*. Any subsequent call to that macro causes the assembler to generate an error.

The **PURGE** directive is intended to clear memory space no longer needed by a macro. If *macroname* is an instruction or directive mnemonic, the directive name is restored to its previous meaning.

The **PURGE** directive is often used with a "macro library" to let you choose those macros from the library that you really need in your source file. A macro library is simply a file containing macro definitions. You add this library to your source file using the **INCLUDE** directive, then remove unwanted definitions using the **PURGE** directive.

It is not necessary to **PURGE** a macro before redefining it. Any redefinition of a macro automatically purges the previous definition. Also, any macro can purge itself as long as the **PURGE** directive is on the last line of the macro.

Examples

```
PURGE    addup
```

The first example deletes the macro named `addup`.

```
PURGE    mac1, mac2, mac9
```

The second example deletes the macros named `mac1` , `mac2` , and `mac9`.

8.2.5 REPT and ENDM Directives

Syntax

REPT *expression*
statements
ENDM

The **REPT** and **ENDM** directives enclose a block of *statements* to be repeated *expression* number of times. The expression must evaluate to a 16-bit unsigned number. It must not contain external or undefined symbols. The statements can be any valid statements.

Example

```
x       =       0
        REPT    10
x       =       x + 1
        DB      x
        ENDM
```

This example repeats the equal-sign (=) and **DB** directives 10 times. The resulting statements create 10 bytes of data whose values range from 1 to 10.

8.2.6 IRP and ENDM Directives

Syntax

IRP *dummyname*,<*parameter*,,,>
statements
ENDM

The **IRP** and **ENDM** directives designate a block of *statements* to be repeated once for each *parameter* in the list enclosed by angle brackets (< >). The *dummyname* is a name for a placeholder to be replaced by the current *parameter*. The parameter can be any legal symbol, string, numeric, or character constant. Any number of parameters can be given. If you give more than one parameter, you must separate them with commas (,). The angle brackets (< >) around the parameter list are required. The *statements* can be any valid assembler statements. The *dummyname* can be used any number of times in these statements.

When **MASM** encounters an **IRP** directive, it makes one copy of the statements for each parameter in the enclosed list. While copying the statements, it substitutes the current parameter for all occurrences of *dummyname* in these statements. If a null parameter (< >) is found in the list, the dummy name is replaced with a null value. If the parameter list is empty, the **IRP** directive is ignored and no statements are copied.

Example

```
IRP     x,<0,1,2,3,4,5,6,7,8,9>
        DB   10 DUP(x)
ENDM
```

This example repeats the **DB** directive 10 times, duplicating the numbers in the list once for each repetition. The resulting statements create 100 bytes of data with the values 0 through 9 duplicated 10 times.

Notes

Assume an **IRP** directive is used inside a macro definition and the parameter list of the **IRP** directive is also a dummy parameter of the macro. In this case, you must enclose that dummy parameter within angle brackets. For example, in the following macro definition, the dummy parameter x is used as the parameter list for the **IRP** directive:

```
alloc       MACRO   x
            IRP     y,<x>
            DB      y
            ENDM
            ENDM
```

If this macro is called with

```
alloc <0,1,2,3,4,5,6,7,8,9>
```

the macro expansion becomes

```
IRP         y,<0,1,2,3,4,5,6,7,8,9>
DB          y
ENDM
```

The macro removes the brackets from the actual parameter before replacing the dummy parameter. You must provide the angle brackets for the parameter list yourself.

8.2.7 IRPC and ENDM Directives

Syntax

IRPC *dummyname*,*string*
statements
ENDM

The **IRPC** and **ENDM** directives enclose a block of *statements* that is repeated once for each character in *string.* The *dummyname* is a name for a placeholder to be replaced by the current character in the string. The string can be any combination of letters, digits, and other characters. The string should be enclosed with angle brackets (< >) if it contains spaces, commas, or other separating characters. The statements can be any valid assembler statements. The dummyname can be used any number of times in these statements.

When **MASM** encounters an **IRPC** directive, it makes one copy of the statements for each character in the string. While copying the statements, it substitutes the current character for all occurrences of *dummyname* in these statements.

Example

```
IRPC    x,0123456789
        DB      x + 1
ENDM
```

This example repeats the **DB** directive 10 times, once for each character in the string `0123456789`. The resulting statements create 10 bytes of data having the values 1 through 10.

8.2.8 EXITM Directive

Syntax

EXITM

The **EXITM** directive tells the assembler to terminate macro or repeat-block expansion and continue assembly with the next statement after the macro call or repeat block. The **EXITM** directive is typically used with **IF** directives to allow conditional expansion of the last statements in a macro or repeat block.

When **EXITM** is encountered, the assembler exits the macro or repeat block immediately. Any remaining statements in the macro or repeat block are not processed. If **EXITM** is encountered in a macro or repeat block nested in another macro or repeat block, **MASM** returns to expanding the outer level block.

Example

```
alloc MACRO  times
      x      =     0
      REPT   times          ;; Repeat up to 256 times
             IFE   x - OFFh ;; Does x = 255 yet?
             EXITM          ;; If so, quit
             ELSE
             DB    x        ;; Else allocate x
             ENDIF
      x      =     x + 1    ;; Increment x
      ENDM
      ENDM
```

This example defines a macro that creates no more than 255 bytes of data. The macro contains an **IFE** directive that checks the expression `x-OFFh`. When this expression is 0 (x equal to 255), the **EXITM** directive is processed and expansion of the macro stops.

8.3 Macro Operators

The macro and conditional directives use the following special set of macro operators:

Operator	Definition
&	Substitute operator
< >	Literal-text operator
!	Literal-character operator
%	Expression operator
;;	Macro comment

When used in a macro definition or a conditional-assembly directive, these operators carry out special control operations, such as text substitution. They are described in Sections 8.3.1–8.3.5.

8.3.1 Substitute Operator

Syntax

&dummyparameter

or

dummyparameter&

The substitute operator (&) forces **MASM** to replace *dummyparameter* with its corresponding actual parameter value. The operator is used anywhere a dummy parameter immediately precedes or follows other characters, or whenever the parameter appears in a quoted string.

Example

```
errgen    MACRO   y,x
error&x   DB      'Error &y - &x'
          ENDM
```

In the example above, **MASM** replaces `&x` with the value of the actual parameter passed to the macro `errgen`. If the macro is called with the statement

```
          errgen  1,wait
```

the macro is expanded to

```
errorwait DB     'Error 1 - wait'
```

Note

For complex, nested macros, you can use extra ampersands (&) to delay the actual replacement of a dummy parameter. In general, you need to supply as many ampersands as there are levels of nesting.

For example, in the following macro definition, the substitute operator is used twice with z to make sure its replacement occurs while the **IRP** directive is being processed:

```
alloc   MACRO   x
        IRP     z,<1,2,3>
        x&&z    DB   z
        ENDM
        ENDM
```

In this example, the dummy parameter x is replaced immediately when the macro is called. The dummy parameter z, however, is not replaced until the **IRP** directive is processed. This means the parameter is replaced once for each number in the **IRP** parameter list. If the macro is called with

```
        alloc   var
```

the expanded macro will be

```
var1    DB      1
var2    DB      2
var3    DB      3
```

8.3.2 Literal-Text Operator

Syntax

<*text*>

The literal-text operator directs **MASM** to treat *text* as a single literal element regardless of whether it contains commas, spaces, or other separators. The operator is most often used with macro calls and the **IRP** directive to ensure that values in a parameter list are treated as a single parameter.

The literal text operator can also be used to force MASM to treat special characters such as the semicolon (;) or the ampersand (&) literally. For example, the semicolon inside angle brackets <;> becomes a semicolon, not a comment indicator.

MASM removes one set of angle brackets each time the parameter is used in a macro. When using nested macros, you will need to supply as many sets of angle brackets as there are levels of nesting.

8.3.3 Literal-Character Operator

Syntax

!*character*

The literal-character operator forces the assembler to treat *character* as a literal character. For example, you can use it to force MASM to treat special characters such as the semicolon (;) or the ampersand (&) literally. Therefore, !; is equivalent to <;>.

8.3.4 Expression Operator

Syntax

%*text*

The expression operator (%) causes the assembler to treat *text* as an expression. MASM computes the expression's value, using numbers of the current radix, and replaces *text* with this new value. The *text* must represent a valid expression.

The expression operator is typically used in macro calls where the programmer needs to pass the result of an expression to the macro instead of to the actual expression.

Example

```
printe  MACRO   msg,num
        IF2                     ;; On pass 2 only
        %OUT    * &msg&num *    ;; Display message and number
        ENDIF                   ;;   to screen
        ENDM

sym1    EQU     100
sym2    EQU     200

        printe  <sym1 + sym2 = >,%(sym1 + sym2) ; Macro call
```

In this example, the macro call

```
        printe  <sym1 + sym2 = >,%(sym1 + sym2)
```

passes the text literal `sym1 + sym2 =` to the dummy parameter `msg`. It passes the value 300 (the result of the expression `sym1 + sym2`) to the dummy parameter `num`. The result is that **MASM** displays the message `sym1+sym2=300` when it reaches the macro call during the assembly. The **%OUT** directive, which sends a message to the screen, is described in Section 9.4 and the **IF2** directive is described in Section 7.2.2.

8.3.5 Macro Comment

Syntax

;;*text*

A macro comment is any text in a macro definition that does not need to be copied in the macro expansion. All *text* following the double semicolon (;;) is ignored by the assembler and will appear only in the macro definition when the source listing is created.

The regular comment operator (;) can also be used in macros. However, regular comments may appear in listings when the macro is expanded. Macro comments will appear in the macro definition, but not in macro expansions. Whether or not regular comments are listed in macro expansions depends on the use of the **.LALL**, **.XALL**, and **.SALL** directives described in Section 9.11.

Chapter 9
File Control Directives

9.1 Introduction

This chapter describes the MASM file-control directives, which provide control of the source, object, and listing files read and created by the assembler.

The file-control directives include the following:

Directive	Meaning
INCLUDE	Include a source file
.RADIX	Change default input radix
%OUT	Display message on console
NAME	Copy name to object file
TITLE	Set program-listing title
SUBTTL	Set program-listing subtitle
PAGE	Set program-listing page size and line width
.LIST	List statements in program listing
.XLIST	Suppress listing of statements
.LFCOND	List false conditional in program listing
.SFCOND	Suppress false-conditional listing
.TFCOND	Toggle false-conditional listing
.LALL	Include macro expansions in program listing
.SALL	Suppress listing of macro expansions
.XALL	Exclude comments from macro listing
.CREF	List symbols in cross-reference file
.XCREF	Suppress symbol listing

Sections 9.2–9.12 describe these directives in detail.

9.2 INCLUDE Directive

Syntax

INCLUDE *filename*

The **INCLUDE** directive inserts source code from the source file given by *filename* into the current source file during assembly. The *filename* must name an existing file. A full or partial path name may be given if the file is not in the current working directory. **MASM** first looks for the "include" file (the source file specified by *filename)* in any paths specified with the **MASM /I** option, then it checks the current directory. If the named file is not found, the assembler displays an error message and stops.

When the assembler encounters an **INCLUDE** directive, it opens the specified source file and immediately begins assembling its statements. When all statements have been read, **MASM** continues assembly with the statement immediately following the directive.

Nested **INCLUDE** directives are allowed. A file named by an **INCLUDE** directive can contain **INCLUDE** directives. **MASM** marks included statements with the letter C in listings.

Directories can be specified in **INCLUDE** path names with either the backslash (\) or the forward slash (/). This is for XENIX® compatibility.

You should specify a file name, but no path name with the **INCLUDE** directive if you plan to set a search path with the **MASM /I** option. The **/I** option is discussed in Section 2.3.6 of the *Microsoft Macro Assembler User's Guide.*

Examples

```
INCLUDE entry                   ; File name
INCLUDE b:\include\record       ; Path name
INCLUDE /include/as/stdio       ; Path name
INCLUDE localinc\define.inc     ; Partial path name
```

9.3 .RADIX Directive

Syntax

.RADIX *expression*

The **.RADIX** directive sets the input radix for numbers in the source file. The *expression* is a number in the range 2 to 16. It defines whether the numbers are binary, octal, decimal, hexadecimal, or numbers of some other base. The most common bases are listed below:

Base	**Number type**
2	binary
8	octal
10	decimal
16	hexadecimal

The *expression* is always considered a decimal number, regardless of the current input radix. The default input radix is decimal.

Notes

The **.RADIX** directive does not affect the **DD**, **DQ**, or **DT** directives. Numbers entered in the expression of these directives are always evaluated as decimal unless a radix specifier is appended to the value.

The **.RADIX** directive does not affect the optional radix specifiers, **B** and **D**, used with integer numbers. When **B** or **D** appears at the end of any integer, it is always considered to be a radix specifier even if the current input radix is 16.

For example, if the input radix is 16, the number `0ABCD` will be interpreted as 0ABC decimal, an illegal number, instead of as 0ABCD hexadecimal, as intended. Type `0ABCDh` to specify 0ABCD in hexadecimal. Similarly, the number `11B` will be treated as 11 binary, a legal number, but not 11B hexadecimal, as intended. Type `11Bh` to specify 11B in hexadecimal.

Examples

```
.RADIX  16
.RADIX  2
```

The first example sets the input radix to hexadecimal, while the second sets the input radix to binary.

9.4 %OUT Directive

Syntax

%OUT *text*

The **%OUT** directive instructs the assembler to display the *text* on the screen when it reaches the line containing the specified *text* during assembly. The directive is useful for displaying messages at specific points of a long assembly.

The **%OUT** directive generates output for both assembly passes. The **IF1** and **IF2** directives can be used to control when the directive is processed.

Example

```
IF1
        %OUT First Pass - OK
ENDIF
```

This sample block could be placed at the end of a source file so that the message `First Pass - OK` would be displayed at the end of the first pass, but ignored on the second pass.

9.5 NAME Directive

Syntax

NAME *modulename*

The **NAME** directive sets the name of the current module to *modulename*. A module name is used by the linker when displaying error messages.

The *modulename* can be any combination of letters and digits. Although the module name can be any length, only the first six characters are used. The name must be unique and not a reserved word.

If the **NAME** directive is not used, the assembler creates a default module name using the first six characters of the text specified in the **TITLE** directive. If no **TITLE** directive is found, the default name A is used.

Example

```
NAME Grafix
```

This example sets the module name to `Grafix`.

9.6 TITLE Directive

Syntax

TITLE *text*

The **TITLE** directive specifies the program-listing title. It directs **MASM** to copy *text* to the first line of each new page in the program listing. The text can be any combination of characters up to 60 characters in length.

No more than one **TITLE** directive per module is allowed. The first 6 non-blank characters of the title are used as the module name if the module does not contain a **NAME** directive.

Example

```
TITLE Graphics - First program
```

This example sets the title to `Graphics - First program`. If the module does not contain a **NAME** directive, the module name will be set to `Graphi` (the first six characters of `Graphics`.)

9.7 SUBTTL Directive

Syntax

SUBTTL *text*

The **SUBTTL** directive specifies the listing subtitle. It directs the assembler to copy *text* to the line immediately following the title on each new page in the program listing. The *text* can be any combination of characters. Only the first 60 characters are used. If no *text* is given, the subtitle line is left blank.

Any number of **SUBTTL** directives can be given in a program. Each new directive replaces the current subtitle with the new *text.*

Examples

```
SUBTTL Point Plotting Routines
```

The example above creates the subtitle `Point Plotting Routines`.

```
SUBTTL
```

The example above creates a blank subtitle.

9.8 PAGE Directive

Syntax

PAGE *length,width*
PAGE +
PAGE

The **PAGE** directive can be used to designate the line length and width for the program listing, to increment the section and adjust the section number accordingly, or to generate a page break in the listing.

If *length* and *width* are specified, the **PAGE** directive sets the maximum number of lines per page to *length*, and the maximum number of characters per line to *width*. The *length* must be in the range 10 to 255. The default page length is 50. The *width* must be in the range 60 to 132. The default page width is 80. If *width* is specified, but *length* is not, a comma (,) must precede *width*.

If a plus sign (+) follows **PAGE**, the section number is incremented and the page number is reset to 1. Program listing page numbers have the form

section-page

where *section* is the section number within the module, and *page* is the page number within the section. By default, section and page numbers begin with 1-1.

If no argument is given, **PAGE** starts a new output page in the program listing. It copies a form-feed character to the file and generates a title and subtitle line.

Examples

```
PAGE
```

The first example creates a page break.

```
PAGE 58,60
```

The second example sets the maximum page length to 58 lines, and the maximum width to 60 characters.

```
PAGE ,132
```

The third example sets the maximum width to 132 characters. The current page length (either the default of 50 or a previously set value) remains unchanged.

```
PAGE +
```

The final example increments the current section number and sets the page number to 1. For example, if the preceding page was 3-6, the new page would be 4-1.

9.9 .LIST and .XLIST Directives

Syntax

.LIST
.XLIST

The **.LIST** and **.XLIST** directives control which source-program lines are copied to the program listing. The **.XLIST** directive suppresses copying of subsequent source lines to the program listing. The **.LIST** directive restores copying. The directives are typically used in pairs, to prevent a particular section of a source file from being copied to the program listing.

The **.XLIST** directive overrides all other listing directives.

Example

```
.XLIST          ; Listing suspended here
       .
       .
       .
.LIST           ; Listing resumes here
       .
       .
       .
```

9.10 .SFCOND, .LFCOND, and .TFCOND Directives

Syntax

.SFCOND
.LFCOND
.TFCOND

The **.SFCOND** and **.LFCOND** directives determine whether false-conditional blocks should be listed.

The **.SFCOND** directive suppresses the listing of any subsequent conditional blocks whose **IF** condition is false. The **.LFCOND** directive restores the listing of these blocks. Like **.LIST** and **.XLIST**, false-conditional listing directives can be used to suppress listing of conditional blocks in sections of a program.

The **.TFCOND** directive sets the default mode for listing of conditional blocks. This directive works in conjunction with the **/X** option of the assembler. If **/X** is not given in the **MASM** command line, **.TFCOND** causes false-conditional blocks to be listed by default. If **/X** is given, **.TFCOND** causes false-conditional blocks to be suppressed. Every time a new **.TFCOND** is inserted in the source code, listing of false-conditionals is turned off if it was on, or on if it was off.

The **/X** option is discussed in Section 2.3.15 of the *Microsoft Macro Assembler User's Guide.*

Example

```
test1   DB     0     ; Symbol defined so all conditionals false

                     ; /X not used       /X used
.SFCOND
IFNDEF  test1        ; Not listing       Not listed
test2   DB     128
ENDIF
.LFCOND
IFNDEF  test1        ; Listed            Listed
test2   DB     128
ENDIF
.TFCOND
IFNDEF  test1        ; Listed            Not listed
test2   DB     128
ENDIF
.TFCOND
IFNDEF  test1        ; Not listed        Listed
test2   DB     128
ENDIF
```

In the example above, the listing for the last two conditionals would be reversed if the **/X** option were used. The first block with **.TFCOND** would not be listed and the second block would be listed.

9.11 .LALL, .XALL, and .SALL Directives

Syntax

.LALL
.XALL
.SALL

The **.LALL**, **.XALL**, and **.SALL** directives control the listing of the statements in macros that have been expanded in the source file. The assembler lists the full macro definition, but lists macro expansions only if the appropriate directive is set.

The **.LALL** directive causes **MASM** to list all the source statements in a macro, including comments preceded by a single semicolon (;), but not those preceded by a double semicolon (;;). The **.XALL** directive lists only those source statements that generate code or data. Comments are ignored.

The **.SALL** directive suppresses listing of all macro expansions. That is, the assembler copies the macro call to the source listing, but does not copy the source lines generated by the call.

The **.XALL** directive is in effect when **MASM** first begins execution.

For the sample listing below, assume that the following macro has been defined at the beginning of the source file:

```
tryout  MACRO
;;Macro comment line
; Normal comment line
        IF2                     ; No code or data
        ASSUME cs:code          ; No code or data
        DW     20 DUP(?)        ; Generates data
        mov    ax,bx            ; Generates code
        ENDIF                   ; No code or data
        ENDM
```

Assume also that the macro has been called once in the source file with each of the following macro listing directives:

```
.LALL
        tryout                  ; Call with .LALL
.XALL
        tryout                  ; Call with .XALL
.SALL
        tryout                  ; Call with .SALL
```

Example

```
                                .LALL
                                        tryout
                          1             ; Normal comment line
                          1             IF2                  ; No code or data
                          1             ASSUME cs:code       ; No code or data
0005  0014[               1             DW      20 DUP(?)    ; Generates data
           ????           1
                          1
002D  8B C3               1             mov     ax,bx        ; Generates code
                          1             ENDIF                ; No code or data

                                .XALL
                                        tryout
002F  0014[               1             DW      20 DUP(?)    ; Generates data
0057  8B C3               1             mov     ax,bx        ; Generates code

                                .SALL
                                        tryout
```

Notice that the macro comment line is never listed in macro expansions. The normal comment line is listed only with the **.LALL** directive.

9.12 .CREF and .XCREF Directives

Syntax

.CREF
.XCREF ⟦*name*,,,⟧

The **.CREF** and **.XCREF** directives control the generation of cross-references for the macro assembler's cross-reference file. The **.XCREF** directive suppresses the generation of label, variable, and symbol cross-references. The **.CREF** directive restores this generation.

If *name* is specified with **.XCREF**, only that label, variable, or symbol will be suppressed. All other names will be cross-referenced. The named label, variable, or symbol will also be omitted from the symbol table of the program listing. If two or more names are to be given, they must be separated by commas (,).

Example

```
.XCREF                  ; Suppress cross-referencing
      .                 ;   of symbols in this block
      .
      .
.CREF                   ; Restore cross-referencing
      .                 ;   of symbols in this block
      .
      .
.XCREF test1,test2      ; Don't cross-reference test1 or test2
      .                 ;   in this block
      .
      .
```

Appendixes

Appendix A
Instruction Summary

A.1 Introduction

The Microsoft Macro Assembler (**MASM**) is an assembler for the Intel 8086/80186/80286 family of microprocessors. It is capable of assembling instructions for the 8086, 8088, 80186, and 80286 microprocessors and the 8087 and 80287 floating-point coprocessors. Programs must use the instruction syntax described in this chapter.

By default, **MASM** recognizes the 8086 and 8087 instruction sets only (the 8088 set is identical to the 8086 set). If a source program contains 80186, 80286, or 80287 instructions, one or more instruction-set directives must be used in the source file to enable assembly of the additional instructions available in those instruction sets. Sections A.2–A.7 provide lists of the syntax of all instructions recognized by **MASM** with the various instruction-set directives.

Table A.1 explains the abbreviations used in the syntax descriptions.

Table A.1

Syntax Abbreviations

Abbreviation	Meaning
accum	One of the accumulators: **AX** or **AL**
reg	One of the byte or word registers Byte: **AL, AH, BL, BH, CL, CH, DL, DH** Word: **AX, BX, CX, DX, SI, DI, BP, SP**
segreg	One of the segment registers: **CS, DS, SS, ES**
r/m	One of the general operands: register, memory address, indexed operand, based operand, based-indexed operand
immed	8- or 16-bit immediate value: constant or symbol
mem	One of the memory operands: label, variable, symbol
label	Instruction label
src	Source in string operations
dest	Destination in string operations

A.2 8086 Instructions

The 8086 instructions are listed below. (The 8088 instructions are identical to 8086 instructions.) **MASM** assembles 8086 instructions by default.

Syntax	Action
AAA	ASCII adjust for addition
AAD	ASCII adjust for division
AAM	ASCII adjust for multiplication
AAS	ASCII adjust for subtraction
ADC *accum,immed*	Add immediate with carry to accumulator
ADC *r/m,immed*	Add immediate with carry to operand
ADC *r/m,reg*	Add register with carry to operand
ADC *reg,r/m*	Add operand with carry to register
ADD *accum,immed*	Add immediate to accumulator
ADD *r/m,immed*	Add immediate to operand
ADD *r/m,reg*	Add register to operand
ADD *reg,r/m*	Add operand to register
AND *accum,immed*	Bitwise **AND** immediate with accumulator
AND *r/m,immed*	Bitwise **AND** immediate with operand
AND *r/m,reg*	Bitwise **AND** register with operand
AND *reg,r/m*	Bitwise **AND** operand with register
CALL *label*	Call instruction at label
CALL *r/m*	Call instruction indirect
CBW	Convert byte to word
CLC	Clear carry flag
CLD	Clear direction flag
CLI	Clear interrupt flag

CMC	Complement carry flag
CMP *accum,immed*	Compare immediate with accumulator
CMP *r/m,immed*	Compare immediate with operand
CMP *r/m,reg*	Compare register with operand
CMP *reg,r/m*	Compare operand with register
CMPS *src,dest*	Compare strings
CMPSB	Compare strings byte for byte
CMPSW	Compare strings word for word
CWD	Convert word to doubleword
DAA	Decimal adjust for addition
DAS	Decimal adjust for subtraction
DEC *r/m*	Decrement operand
DEC *reg*	Decrement 16-bit register
DIV *r/m*	Divide accumulator by operand
ESC *immed,r/m*	Escape with 6-bit immediate and operand
HLT	Halt
IDIV *r/m*	Integer divide accumulator by operand
IMUL *r/m*	Integer multiply accumulator by operand
IN *accum,immed*	Input from port (8-bit immediate)
IN *accum*,**DX**	Input from port given by **DX**
INC *r/m*	Increment operand
INC *reg*	Increment 16-bit register
INT 3	Software interrupt 3 (encoded as one byte)
INT *immed*	Software interrupts 0–255
INTO	Interrupt on overflow
IRET	Return from interrupt
JA *label*	Jump on above
JAE *label*	Jump on above or equal

JB *label*	Jump on below
JBE *label*	Jump on below or equal
JC *label*	Jump on carry
JCXZ *label*	Jump on **CX** zero
JE *label*	Jump on equal
JG *label*	Jump on greater
JGE *label*	Jump on greater or equal
JL *label*	Jump on less than
JLE *label*	Jump on less than or equal
JMP *label*	Jump to instruction at label
JMP *r/m*	Jump to instruction indirect
JNA *label*	Jump on not above
JNAE *label*	Jump on not above or equal
JNB *label*	Jump on not below
JNBE *label*	Jump on not below or equal
JNC *label*	Jump on no carry
JNE *label*	Jump on not equal
JNG *label*	Jump on not greater
JNGE *label*	Jump on not greater or equal
JNL *label*	Jump on not less than
JNLE *label*	Jump on not less than or equal
JNO *label*	Jump on not overflow
JNP *label*	Jump on not parity
JNS *label*	Jump on not sign
JNZ *label*	Jump on not zero
JO *label*	Jump on overflow
JP *label*	Jump on parity
JPE *label*	Jump on parity even
JPO *label*	Jump on parity odd

JS *label*	Jump on sign
JZ *label*	Jump on zero
LAHF	Load **AH** with flags
LDS *r/m*	Load operand into **DS**
LEA *r/m*	Load effective address of operand
LES *r/m*	Load operand into **ES**
LOCK	Lock bus
LODS *src*	Load string
LODSB	Load byte from string into **AL**
LODSW	Load word from string into **AX**
LOOP *label*	Loop
LOOPE *label*	Loop while equal
LOOPNE *label*	Loop while not equal
LOOPNZ *label*	Loop while not zero
LOOPZ *label*	Loop while zero
MOV *accum,mem*	Move memory to accumulator
MOV *mem,accum*	Move accumulator to memory
MOV *r/m,immed*	Move immediate to operand
MOV *r/m,reg*	Move register to operand
MOV *r/m,segreg*	Move segment register to operand
MOV *reg,immed*	Move immediate to register
MOV *reg,r/m*	Move operand to register
MOV *segreg,r/m*	Move operand to segment register
MOVS *dest,src*	Move string
MOVSB	Move string byte by byte
MOVSW	Move string word by word
MUL *r/m*	Multiply accumulator by operand
NEG *r/m*	Negate operand (2's complement)
NOP	No operation

NOT *r/m*	Invert operand bits (1's complement)
OR *accum,immed*	Bitwise **OR** immediate with accumulator
OR *r/m,immed*	Bitwise **OR** immediate with operand
OR *r/m,reg*	Bitwise **OR** register with operand
OR *reg,r/m*	Bitwise **OR** operand with register
OUT DX,*accum*	Output to port given by **DX**
OUT *immed,accum*	Output to port (8-bit immediate)
POP *r/m*	Pop 16-bit operand
POP *reg*	Pop 16-bit register from stack
POP *segreg*	Pop segment register
POPF	Pop flags
PUSH *r/m*	Push 16-bit operand
PUSH *reg*	Push 16-bit register onto stack
PUSH *segreg*	Push segment register
PUSHF	Push flags
RCL *r/m*,**1**	Rotate left through carry by 1 bit
RCL *r/m*,**CL**	Rotate left through carry by **CL**
RCR *r/m*,**1**	Rotate right through carry by 1 bit
RCR *r/m*,**CL**	Rotate right through carry by **CL**
REP	Repeat
REPE	Repeat if equal
REPNE	Repeat if not equal
REPNZ	Repeat if not zero
REPZ	Repeat if zero
RET ⟦*immed*⟧	Return after popping bytes from stack
ROL *r/m*,**1**	Rotate left by 1 bit
ROL *r/m*,**CL**	Rotate left by **CL**
ROR *r/m*,**1**	Rotate right by 1 bit
ROR *r/m*,**CL**	Rotate right by **CL**

SAHF	Store **AH** into flags
SAL *r/m*,**1**	Shift arithmetic left by 1 bit
SAL *r/m*,**CL**	Shift arithmetic left by **CL**
SAR *r/m*,**1**	Shift arithmetic right by 1 bit
SAR *r/m*,**CL**	Shift arithmetic right by **CL**
SBB *accum,immed*	Subtract immediate and carry flag
SBB *r/m,immed*	Subtract immediate and carry flag
SBB *r/m,reg*	Subtract register and carry flag
SBB *reg,r/m*	Subtract operand and carry flag
SCAS *dest*	Scan string
SCASB	Scan string for byte in **AL**
SCASW	Scan string for word in **AX**
SHL *r/m*,**1**	Shift left by 1 bit
SHL *r/m*,**CL**	Shift left by **CL**
SHR *r/m*,**1**	Shift right by 1 bit
SHR *r/m*,**CL**	Shift right by **CL**
STC	Set carry flag
STD	Set direction flag
STI	Set interrupt flag
STOS *dest*	Store string
STOSB	Store byte in **AL** at string
STOSW	Store word in **AX** at string
SUB *accum,immed*	Subtract immediate from accumulator
SUB *r/m,immed*	Subtract immediate from operand
SUB *r/m,reg*	Subtract register from operand
SUB *reg,r/m*	Subtract operand from register
TEST *accum,immed*	Compare immediate bits with accumulator
TEST *r/m,immed*	Compare immediate bits with operand

TEST *r/m,reg*	Compare register bits with operand
TEST *reg,r/m*	Compare operand bits with register
WAIT	Wait
XCHG *accum,reg*	Exchange accumulator with register
XCHG *r/m,reg*	Exchange operand with register
XCHG *reg,accum*	Exchange register with accumulator
XCHG *reg,r/m*	Exchange register with operand
XLAT *mem*	Translate
XOR *accum,immed*	Bitwise **XOR** immediate with accumulator
XOR *r/m,immed*	Bitwise **XOR** immediate with operand
XOR *r/m,reg*	Bitwise **XOR** register with operand
XOR *reg,r/m*	Bitwise **XOR** operand with register

The string instructions (**CMPS**, **LODS**, **MOVS**, **SCAS**, and **STOS**) use the **DS**, **SI**, **ES**, and **DI** registers to compute operand locations. Source operands are assumed to be at **DS:**[**SI**]; destination operands at **ES:**[**DI**]. The operand type (**BYTE** or **WORD**) may be defined by the instruction mnemonic. For example, **CMPSB** specifies **BYTE** operands and **CMPSW** specifies **WORD** operands. For the **CMPS**, **LODS**, **MOVS**, **SCAS**, and **STOS** instructions, the *src* and *dest* operands are dummy operands that define the operand type only. The offsets associated with these operands are not used. The *src* operand can also be used to specify a segment override. The **ES** register for the destination operand cannot be overridden.

Examples

```
cmps   WORD PTR string,WORD PTR es:0
lods   BYTE PTR string
mov    BYTE PTR es:0,BYTE PTR string
```

The **REP**, **REPE**, **REPNE**, **REPNZ**, and **REPZ** instructions provide ways to repeatedly execute a string instruction for a given count or while a given condition is true. If a repeat instruction immediately precedes a string instruction (both instructions must be on the same line), the instructions are repeated until the specified repeat condition is false, or the **CX** register is equal to zero. The repeat instruction decrements **CX** by one for each execution.

Example

```
mov     cx,10
rep     scasb
```

In this example, **SCASB** is repeated 10 times.

A.3 8087 Instructions

The 8087 instructions are listed below. **MASM** assembles 8087 instructions by default.

Syntax	Action
F2XM 1	Calculate 2^{X}-
FABS	Take absolute value of top of stack
FADD	Add real
FADD *mem*	Add real from memory
FADD ST, ST(*i*)	Add real from stack
FADD ST(*i*),ST	Add real to stack
FADDP ST(*i*),ST	Add real and pop stack
FBLD *mem*	Load 10-byte packed decimal on stack
FBSTP *mem*	Store 10-byte packed decimal and pop
FCHS	Change sign on the top stack element
FCLEX	Clear exceptions after **WAIT**
FCOM	Compare real
FCOM ST	Compare real with top of stack
FCOM ST(*i*)	Compare real with stack
FCOMP	Compare real and pop stack
FCOMP ST	Compare real with top of stack and pop
FCOMP ST(*i*)	Compare real with stack and pop stack
FCOMPP	Compare real and pop stack twice

FDECSTP	Decrement stack pointer
FDISI	Disable interrupts after WAIT
FDIV	Divide real
FDIV *mem*	Divide real from memory
FDIV ST,ST(*i*)	Divide real from stack
FDIV ST(*i*),ST	Divide real in stack
FDIVP ST(*i*),ST	Divide real and pop stack
FDIVR	Reversed real divide
FDIVR *mem*	Reversed real divide from memory
FDIVR ST,ST(*i*)	Reversed real divide from stack
FDIVR ST(*i*),ST	Reversed real divide in stack
FDIVRP ST(*i*),ST	Reversed real divide and pop stack twice
FENI	Enable interrupts after **WAIT**
FFREE	Free stack element
FFREE ST	Free top-of-stack element
FFREE ST(*i*)	Free *i*th stack element
FIADD *mem*	Add 2- or 4-byte integer
FICOM *mem*	2- or 4-byte integer compare
FICOMP *mem*	2- or 4-byte integer compare and pop stack
FIDIV *mem*	2- or 4-byte integer divide
FIDIVR *mem*	Reversed 2- or 4-byte integer divide
FILD *mem*	Load 2-, 4-, or 8-byte integer on stack
FIMUL *mem*	2- or 4-byte integer multiply
FINCSTP	Increment stack pointer
FINIT	Initialize processor after **WAIT**
FIST *mem*	Store 2- or 4-byte integer
FISTP *mem*	Store 2-, 4-, or 8-byte integer and pop stack

FISUB *mem*	2- or 4-byte integer subtract
FISUBR *mem*	Reversed 2- or 4-byte integer subtract
FLD *mem*	Load 4-, 8-, or 10-byte real on stack
FLD1	Load +1.0 onto top of stack
FLDCW *mem*	Load control word
FLDENV *mem*	Load 8087 environment (14 bytes)
FLDL2E	Load $\log_2 e$ onto top of stack
FLDL2T	Load $\log_2 10$ onto top of stack
FLDLG2	Load $\log_{10} 2$ onto top of stack
FLDLN2	Load $\log_e 2$ onto top of stack
FLDPI	Load pi onto top of stack
FLDZ	Load +0.0 onto top of stack
FMUL	Multiply real
MUL *mem*	Multiply real from memory
FMUL ST,ST(*i*)	Multiply real from stack
FMUL ST(*i*),ST	Multiply real to stack
FMULP ST(*i*),ST	Multiply real and pop stack
FNCLEX	Clear exceptions with no **WAIT**
FNDISI	Disable interrupts with no **WAIT**
FNENI	Enable interrupts with no **WAIT**
FNINIT	Initialize processor, with no **WAIT**
FNOP	No operation
FNSAVE *mem*	Save 8087 state (94 bytes) with no **WAIT**
FNSTCW *mem*	Store control word with no **WAIT**
FNSTENV *mem*	Store 8087 environment with no **WAIT**
FNSTSW *mem*	Store 8087 status word with no **WAIT**
FPATAN	Partial arctangent function
FPREM	Partial remainder

FPTAN	Partial tangent function
FRNDINT	Round to integer
FRSTOR *mem*	Restore 8087 state (94 bytes)
FSAVE *mem*	Save 8087 state (94 bytes) after **WAIT**
FSCALE	Scale
FSQRT	Square root
FST	Store real
FST ST	Store real from top of stack
FST ST(*i*)	Store real from stack
FSTCW *mem*	Store control word with **WAIT**
FSTENV *mem*	Store 8087 environment after **WAIT**
FSTP *mem*	Store 4-, 8-, or 10-byte real and pop stack
FSTSW *mem*	Store 8087 status word after **WAIT**
FSUB	Subtract real
FSUB *mem*	Subtract real from memory
FSUB ST,ST(*i*)	Subtract real from stack
FSUB ST(*i*),ST	Subtract real to stack
FSUBP ST(*i*),ST	Subtract real and pop stack
FSUBR	Reversed real subtract
FSUBR *mem*	Reversed real subtract from memory
FSUBR ST,ST(*i*)	Reversed real subtract from stack
FSUBR ST(*i*),ST	Reversed real subtract in stack
FSUBRP ST(*i*),ST	Reversed real subtract and pop stack
FTST	Test top of stack
FWAIT	Wait for last 8087 operation to complete
FXAM	Examine top-of-stack element
FXCH	Exchange contents of stack element
FFREE ST	Exchange top-of-stack element

FFREE ST(*i*)	Exchange top-of-stack and *i*th element
FXTRACT	Extract exponent and significand
FYL2X	Calculate Y $\log_2 x$
FYL2PI	Calculate Y $\log_2(x+1)$

A.4 80186 Instruction Mnemonics

The 80186 instruction set consists of all 8086 instructions plus the following instructions. The **.186** directive must be used to enable these instructions.

Syntax	**Action**
BOUND *reg, mem*	Detect value out of range
ENTER *immed16,immed8*	Enter procedure
IMUL *reg,immed*	Integer multiply register by immediate
IMUL *reg,r/m,immed*	Integer multiply general operand by immediate and store result in register
INS *mem*,**DX**	Input string from port **DX**
INSB *mem*,**DX**	Input byte string from port **DX**
INSW *mem*,**DX**	Input word string from port **DX**
LEAVE	Leave procedure
OUTS DX,*mem*	Output byte/word string to port **DX**
OUTSB DX,*mem*	Output byte string to port **DX**
OUTSW DX,*mem*	Output word string to port **DX**
POPA	Pop all registers
PUSH *immed*	Push immediate data onto stack
PUSHA	Push all registers
RCL *r/m,immed*	Rotate left through carry by immediate
RCR *r/m,immed*	Rotate right through carry by immediate
ROL *r/m,immed*	Rotate left by immediate

ROR *r/m,immed*	Rotate right by immediate
SAL *r/m,immed*	Shift arithmetic left by immediate
SAR *r/m,immed*	Shift arithmetic right by immediate
SHL *r/m,immed*	Shift left by immediate
SHR *r/m,immed*	Shift right by immediate

A.5 80286 Nonprotected Instructions

The 80286 nonprotected instruction set consists of all 8086 instructions plus the following instructions. The **.286c** directive must be used to enable these instructions.

Syntax	**Action**
BOUND *reg,mem*	Detect value out of range
ENTER *immed16,immed8*	Enter procedure
IMUL *reg,immed*	Integer multiply register by immediate
IMUL *reg,r/m,immed*	Integer multiply general operand by immediate and store result in register
INS *mem*,**DX**	Input string from port **DX**
INSB *mem*,**DX**	Input byte string from port **DX**
INSW *mem*,**DX**	Input word string from port **DX**
LEAVE	Leave procedure
OUTS DX,*mem*	Output byte/word string to port **DX**
OUTSB DX, *mem*	Output byte string to port **DX**
OUTSW DX, *mem*	Output word string to port **DX**
POPA	Pop all registers
PUSH *immed*	Push immediate data onto stack
PUSHA	Push all registers
RCL *r/m,immed*	Rotate left through carry by immediate
RCR *r/m,immed*	Rotate right through carry by immediate

ROL *r/m,immed*	Rotate left by immediate
ROR *r/m,immed*	Rotate right by immediate
SAL *r/m,immed*	Shift arithmetic left by immediate
SAR *r/m,immed*	Shift arithmetic right by immediate
SHL *r/m,immed*	Shift left by immediate
SHR *r/m,immed*	Shift right by immediate

A.6 80286 Protected Instruction Mnemonics

The 80286 protected instruction set consists of all 8086 and 80286 non-protected instructions plus the following instructions. The **.286p** directive must be used to enable these instructions.

Syntax	**Action**
ARPL *mem,reg*	Adjust requested privilege level
CLTS	Clear task-switched flag
LAR *reg,mem*	Load access rights
LGDT *mem*	Load global-descriptor table (8 bytes)
LIDT *mem*	Load interrupt-descriptor table (8 bytes)
LLDT *mem*	Load local-descriptor table
LMSW *mem*	Load machine-status word
LSL *reg, mem*	Load segment limit
LTR *mem*	Load task register
SGDT *mem*	Store global-descriptor table (8 bytes)
SIDT *mem*	Store interrupt-descriptor table (8 bytes)
SLDT *mem*	Store local-descriptor table
SMSW *mem*	Store machine-status word
STR *mem*	Store task register
VERR *mem*	Verify read access

VERW *mem*	Verify write access

A.7 80287 Instruction Mnemonics

The 80287 instruction set consists of all 8087 instructions plus the following additional instructions. The **.287** directive must be used to enable these instructions.

FSETPM	Set protected mode
FSTSW AX	Store status word in **AX** (wait)
FNSTSW AX	Store status word in **AX** (no-wait)

Appendix B
Directive Summary

B.1 Introduction

Directives give the assembler directions and information about input and output, memory organization, conditional assembly, listing and cross-reference control, and definitions. Table B.1 lists all directives.

Table B.1

Directives

.186	ENDP	IF1	ORG
.286c	ENDS	IF2	%OUT
.286p	EQU	IFB	PAGE
.287	.ERR	IFDEF	PROC
.8086	.ERR1	IFDIF	PUBLIC
.8087	.ERR2	IFE	PURGE
=	.ERRB	IFIDN	.RADIX
ASSUME	.ERRDEF	IFNB	RECORD
COMMENT	.ERRDIF	IFNDEF	REPT
.CREF	.ERRE	INCLUDE	.SALL
DB	.ERRIDN	IRP	SEGMENT
DD	.ERRNB	IRPC	.SFCOND
DQ	.ERRNDEF	LABEL	STRUC
DT	.ERRNZ	.LALL	SUBTTL
DW	EVEN	.LFCOND	.TFCOND
ELSE	EXITM	.LIST	TITLE
END	EXTRN	LOCAL	.XALL
ENDIF	GROUP	MACRO	.XCREF
ENDM	IF	NAME	.XLIST

Any combination of upper- and lowercase letters can be used when giving directive names in a source file.

B.2 MASM Directives

The directives you can use in MASM source code are listed below with the syntax and function of each. This list is for reference only. See the appropriate chapters in this manual for details.

.186

Enables assembly of 80186 and 8086 instructions.

.286c

Enables assembly of 80286 nonprotected instructions and 8086 instructions.

.286p

Enables assembly of 80286 protected instructions and 8086 instructions.

.287

Enables assembly of 80287 and 8087 instructions.

.8086

Enables assembly of 8086 instructions (and the identical 8088 instructions) while disabling assembly of instructions available only with 80186 and 80286. This is the default mode.

.8087

Enables assembly of 8087 instructions while disabling assembly of instructions available only with 80287. This is the default mode.

name = *expression*

Assigns the numeric value of *expression* to *name.*

ASSUME *segmentregister:segmentname,,,*

Selects *segmentregister* to be the default segment register for all symbols in the named segment or group. If *segmentname* is NOTHING, no register is selected.

COMMENT *delimiter text delimiter*

Treats as a comment all *text* between the given pair of delimiters *delimiter.*

.CREF

Restores listing of symbols in the cross-reference listing file.

⟦*name*⟧ **DB** *initialvalue,,,*

Allocates and initializes a byte (8 bits) of storage for each *initialvalue.*

⟦*name*⟧ **DW** *initialvalue,,,*

Allocates and initializes a word (2 bytes) of storage for each *initialvalue.*

⟦*name*⟧ **DD** *initialvalue,,,*

Allocates and initializes a doubleword (4 bytes) of storage for each *initialvalue.*

⟦*name*⟧ **DQ** *initialvalue,,,*

Allocates and initializes a quadword (8 bytes) of storage for each *initialvalue.*

⟦*name*⟧ **DT** *initialvalue,,,*

Allocates and initializes 10 bytes of storage for each given *initialvalue.*

ELSE

Marks the beginning of an alternate block within a conditional block.

END ⟦*expression*⟧

Marks the end of the module and, optionally, sets the program entry point to *expression.*

ENDIF

Terminates a conditional block.

ENDM

Terminates a macro or repeat block.

name **ENDP**

Marks the end of a procedure definition.

name **ENDS**

Marks the end of a segment or of a structure-type definition.

name **EQU** *expression*

Assigns *expression* to *name.*

.ERR

Generates error.

.ERR1

Generates error on Pass 1 only.

.ERR2

Generates error on Pass 2 only.

.ERRB <*argument*>

Generates error if the *argument* is blank.

.ERRDEF *name*

Generates error if *name* is a previously defined label, variable, or symbol.

.ERRDIF <*string1*>,<*string2*>

Generates error if the strings are different.

.ERRE *expression*

Generates error if the *expression* is false (0).

.ERRIDN <*string1*>,<*string2*>

Generates error if the strings are identical.

.ERRNB <*argument*>

Generates error if the *argument* is not blank.

.ERRNDEF *name*

Generates error if *name* has not yet been defined.

.ERRNZ *expression*

Generates error if *expression* is true (nonzero).

EVEN

If necessary, increments the location counter to an even value and generates one **NOP** instruction (90h).

EXITM

Terminates expansion of the current repeat or macro block and begins assembly of next statement outside the block.

EXTRN *name:type,,,*

Defines an external variable, label, or symbol called *name* whose type is *type*.

name **GROUP** *segmentname,,,*

Associates a group name *name* with one or more segments.

IF *expression*

Grants assembly if *expression* is true (nonzero).

IF1

Grants assembly on Pass 1 only.

IF2

Grants assembly on Pass 2 only.

IFB <*argument*>

Grants assembly if *argument* is blank.

IFDEF *name*

Grants assembly if *name* is a previously defined label, variable, or symbol.

IFDIF <*argument1*>,<*argument2*>

Grants assembly if the arguments are different.

IFE *expression*

Grants assembly if *expression* is false (0).

IFIDN <*argument1*>,<*argument2*>

Grants assembly if the arguments are identical.

IFNB <*argument*>

Grants assembly if *argument* is not blank.

IFNDEF *name*

Grants assembly if *name* has not yet been defined.

INCLUDE *filename*

Inserts source code from the source file given by *filename* into the current source file during assembly.

IRP *dummyname*, <*parameter,,,*>

Marks start of a block that will be repeated for as many parameters as are given, with the current *parameter* replacing the placeholder *dummyname* on each repetition.

IRPC *dummyname*,<*string*>

Marks start of a block that will be repeated for as many characters as there are in *string*, with the current character replacing the placeholder *dummyname* on each repetition.

name **LABEL** *type*

Creates a new variable or label by assigning the current location-counter value and the given *type* to *name*.

.LALL

Lists all statements in a macro.

.LFCOND

Restores the listing of conditional blocks.

.LIST

Restores listing of statements in the program listing.

LOCAL *dummyname,,,*

Declares *dummyname* within a macro as a placeholder for an actual name to be created when the macro is expanded.

name **MACRO** *dummyparameter,,,*

Marks the beginning of macro *name* and establishes each item called *dummyparameter* as a placeholder for the expressions passed when the macro is called.

NAME *modulename*

Sets the name of the current module to *modulename.*

PURGE *macroname,,,*

Deletes the named macros.

ORG *expression*

Sets the location counter to *expression.*

%OUT *text*

Displays *text* at the user's terminal.

name **PROC** *type*

Marks the beginning of procedure *name*, of specified *type.*

PAGE *length,width*

Sets line *length* and character *width* of the program listing.

PAGE +

Increments section-page numbering.

PAGE

Generates a page break in the listing.

PUBLIC *name,,,*

Makes each variable, label, or absolute symbol specified as *name* available to all other modules in the program.

.RADIX *expression*

Sets the input radix for numbers in the source file to *expression.*

recordname **RECORD** *fieldname:width*⟦*=expression*⟧*,,,*

Defines a record type for an 8- or 16-bit record that contains one or more fields.

REPT *expression*

Marks the start of a block that is to be repeated *expression* number of times.

.SALL

Suppresses listing of all macro expansions.

name **SEGMENT** ⟦*align*⟧ ⟦*combine*⟧ ⟦'*class*'⟧

Marks the beginning of a program segment called *name* and having segment attributes *align*, *combine*, and *class*.

.SFCOND

Suppresses listing of any subsequent conditional blocks whose **IF** condition evaluates to false (0).

name **STRUC**

Marks the beginning of a type definition for a structure.

SUBTTL ⟦*text*⟧

Defines the listing subtitle.

.TFCOND

Sets the default mode for listing of conditional blocks.

TITLE *text*

Defines the program listing title.

.XALL

Lists only those macro statements that generate code or data.

.XCREF ⟦*name,,,*⟧

Suppresses the listing of symbols in the cross-reference listing file.

.XLIST

Suppresses listing of subsequent source lines to the program listing.

B.3 MASM Operators

The operators recognized by MASM are listed by precedence in Table B.2. Operations of highest precedence are performed first. Operations of equal precedence are performed from left to right. This default order can be overridden using enclosing parentheses.

Table B.2

Operator Precedence

Precedence	Operators
(Highest)	
1	LENGTH, SIZE, WIDTH, MASK, (), [], <>
2	. (structure field name operator)
3	:
4	PTR, OFFSET, SEG, TYPE, THIS
5	HIGH, LOW
6	+,– (unary)
7	*, /, MOD, SHL, SHR
8	+, – (binary)
9	EQ, NE, LT, LE, GT, GE
10	NOT
11	AND
12	OR, XOR
13	SHORT, .TYPE
(Lowest)	

The syntax of each operator is shown in the following list:

expression1 * *expression2*

Multiply *expression1* by *expression2*.

expression1 / *expression2*

Divide *expression1* by *expression2*.

expression1 + *expression2*

Add *expression1* to *expression2*.

expression1 — expression2

Subtract *expression2* from *expression1*.

+*expression*

Retain the current sign of *expression*.

—*expression*

Reverse the sign of *expression*.

segmentregister:*expression*

Override the default segment of *expression* with *segmentregister*.

segmentname:*expression*

Override the default segment of *expression* with *segmentname*.

groupname:*expression*

Override the default segment of *expression* with *groupname*.

variable.*field*

Add the offset of *field* to the offset of *variable*.

expression1[*expression2*]

Add the value of *expression1* to the value of *expression2*.

&*dummyparameter*

Replace *dummyparameter* with its actual parameter value.

dummyparameter&

Replace *dummyparameter* with its actual parameter value.

<*text*>

Treat *text* as a single literal element.

!*character*

Treat *character* as a literal character rather than as an operator or symbol.

%text

Treat *text* as an expression and compute its value rather than treating it as a string.

;;text

Make *text* into a comment that will not be listed in expanded macros.

expression1 **AND** *expression2*

Do a bitwise Boolean **AND** on *expression1* and *expression2*.

count **DUP** (*initialvalue*)

Specify *count* number of declarations of *initialvalue*.

expression1 **EQ** *expression2*

Return true (0FFFFh) if *expression1* equals *expression2*, or return false (0) if it does not.

expression1 **GE** *expression2*

Return true (0FFFFh) if *expression1* is greater than or equal to *expression2*, or return false (0) if it is not.

expression1 **GT** *expression2*

Return true (0FFFFh) if *expression1* is greater than *expression2*, or return false (0) if it is not.

HIGH *expression*

Return the high byte of *expression*.

expression1 **LE** *expression2*

Return true (0FFFFh) if *expression1* is less than or equal to *expression2*, or return false (0) if it is not.

LENGTH *variable*

Return the length of *variable* in the size in which the variable was declared.

LOW *expression*

Return the low byte of *expression*.

expression1 **LT** *expression2*

Return true (0FFFFh) if *expression1* is less than *expression2*, or return false (0) if it is not.

MASK *recordfieldname*

Return a bit mask in which the bits for *recordfieldname* are set and all other bits are not set.

MASK *record*

Return a bit mask in which the bits used in *record* are set and all other bits are not set.

expression1 **MOD** *expression2*

Return the remainder of dividing *expression1* by *expression2*.

expression1 **NE** *expression2*

Return true (0FFFFh) if *expression1* does not equal *expression2*, or return false (0) if it does.

NOT *expression*

Reverse all bits of *expression*.

OFFSET *expression*

Return the offset of *expression*.

expression1 **OR** *expression2*

Do a bitwise Boolean **OR** on *expression1* and *expression2*.

type **PTR** *expression*

Force the *expression* to be treated as having the specified *type*.

SEG *expression*

Return the segment of *expression*.

expression **SHL** *count*

Shift the bits of *expression* left *count* number of bits.

SHORT *label*

Set type of label to short (having a distance less than 128 bytes from the current location-counter value).

expression **SHR** *count*

Shift the bits of *expression* right *count* number of bits.

SIZE *variable*

Return the total number of bytes allocated for *variable.*

THIS *type*

Create an operand of specified *type* whose offset and segment values are equal to the current location-counter value.

TYPE *expression*

Return the type of *expression.*

.TYPE *expression*

Return a byte defining the mode and scope of *expression.*

WIDTH *recordfieldname*

Return the width in bits of the current *recordfieldname.*

WIDTH *record*

Return the width in bits of the current *record.*

expression1 **XOR** *expression2*

Do a bitwise Boolean **XOR** on *expression1* and *expression2.*

Appendix C
Segment Names for High-Level Languages

C.1 Introduction

This appendix describes the naming conventions used to form assembly-language source files compatible with object modules produced by recent Microsoft language compilers. Compilers that use these conventions include the following:

Microsoft C Version 3.0 or later

Microsoft Pascal Version 3.3 or later

Microsoft FORTRAN Version 3.3 or later

High-level-language modules have the following four predefined segment types:

Type	Use
TEXT	For program code
DATA	For program data
BSS	For uninitialized space
CONST	For constant data

Any assembly-language source file to be assembled and linked to a high-level-language module must use these segments, as described in Sections C.2–C.6.

High-level-language modules also have three different memory models:

Model	Use
Small	For single code and data segments
Middle	For multiple code segments, but a single data segment
Large	For multiple code and multiple data segments

Assembly-language source files to be assembled for a given memory model must use the naming conventions detailed in Sections C.2–C.6.

C.2 Text Segments

Syntax

[[*prefix*]]_**TEXT SEGMENT byte public 'CODE'**
  ASSUME cs:[[*prefix*]]_**TEXT**
statements
[[*prefix*]]_**TEXT ENDS**

A text segment defines a module's program code. It contains *statements* that define instructions and data within the segment. A text segment must have the name *prefix*_**TEXT**, where *prefix* can be any valid string. For middle- and large-model programs, the module's own name is recommended. For small-model programs, *prefix* is omitted; the segment must be called _**TEXT**.

A segment can contain any combination of instructions and data statements. These statements must appear in an order that creates a valid program. All instructions and data addresses in a text segment are relative to the **CS** segment register. Therefore, the **ASSUME** statement must appear at the beginning of the segment. This statement ensures that each label and variable declared in the segment will be associated with the **CS** segment register (see Section 3.7).

Text segments should have **byte** align type and **public** combine type, and must have the class name **'CODE'**. These define loading instructions to be passed to the linker. Although other segment attributes are available, they should not be used. For a complete description of the attributes, see Sections 3.4.1, 3.4.2, and 3.4.3.

The following formats are used for each of the different memory models:

Model	Requirements
Small model	Only one text segment is allowed. The segment must not exceed 64K. All procedure and statement labels should have the **NEAR** type.

Example

```
_TEXT         SEGMENT byte public 'CODE'
              ASSUME cs:_TEXT
_main         PROC near
              .
              .
              .
_main         ENDP
_TEXT         ENDS
```

Middle or large model

Multiple text segments are allowed. However, no segment can exceed 64K. To distinguish one segment from another, each should have its own name. Since most modules contain only one text segment, the module's name is often used as part of the text segment's name. All procedure and statement labels should have the **FAR** type, unless they will only be accessed from within the same segment.

Example

```
SAMPLE_TEXT   SEGMENT byte public 'CODE'
              ASSUME cs:SAMPLE_TEXT
_main         PROC far
              .
              .
              .
_main         ENDP
SAMPLE_TEXT   ENDS
```

C.3 Data Segments – Near

Syntax

DGROUP GROUP _DATA
ASSUME ds:DGROUP
_DATA SEGMENT word public 'DATA'
statements
_DATA ENDS

A near data segment defines initialized data in the segment pointed to by the **DS** segment register when the program starts execution. The segment is **NEAR** because all data in the segment are accessible without giving an explicit segment value. All programs have exactly one near data segment. Only large-model programs can have additional data segments.

A near data segment's name must be **_DATA**. The segment can contain any combination of data *statements* defining variables to be used by the program. The segment must not exceed 64K of data. All data addresses in the segment are relative to the predefined group **DGROUP**. Therefore, the **GROUP** and **ASSUME** statements must appear at the beginning of the segment. These statements ensure that each variable declared in the data segment will be associated with the **DS** segment register and **DGROUP** (see Sections 3.6 and 3.7).

Near data segments must have **word** align type, **public** combine type, and must have the class name **'DATA'**. These define loading instructions that are passed to the linker. Although other segment attributes are available, they must not be used. For a complete description of the attributes, see Sections 3.4.1–3.4.3.

Example

```
DGROUP  GROUP   _DATA
        ASSUME  ds:DGROUP

_DATA   SEGMENT word public 'DATA'
count   DW      0
array   DW      10 dup(1)
string  DB      "Type CANCEL then press RETURN", 0Ah, 0
_DATA   ENDS
```

C.4 Data Segments – Far

Syntax

*prefix_***DATA SEGMENT word public 'FAR_DATA'**
statements
*prefix_***DATA ENDS**

A far data segment defines data or data space that can be accessed only by specifying an explicit segment value. Only large-model programs can have far data segments.

A far data segment's name must be *prefix_***DATA**, where *prefix* can be any valid string. The name of the first variable declared in the segment is recommended. The segment can contain any combination of data *statements* defining variables to be used by the program. The segment must not exceed 64K of data. All data addresses in the segment are relative to the **ES** segment register. When accessing a variable in a far data segment, the **ES** register must be set to the appropriate segment value. Also, the segment override operator (**:**) must be used with the variable's name (see Section 5.3.7).

Far data segments must have **word** align type, **public** combine type, and should have the class name **'FAR_DATA'**. These define loading instructions that are passed to the linker. Although other segment attributes are available, they must not be used. For a complete description of the attributes, see Sections 3.4.1–3.4.3.

Example

```
ARRAY_DATA      SEGMENT word public 'FAR_DATA'
array    DW     0
         DW     1
         DW     2
         DW     4
table    DW     1600 DUP(?)
ARRAY_DATA      ENDS
```

C.5 BSS Segments

Syntax

DGROUP GROUP _BSS
ASSUME ds:DGROUP
_BSS SEGMENT word public 'BSS'
statements
_BSS ENDS

A **BSS** segment defines uninitialized data space. A **BSS** segment's name must be **_BSS**. The segment can contain any combination of data *statements* defining variables to be used by the program. The segment must not exceed 64K. All data addresses in the segment are relative to the predefined group **DGROUP**. Therefore, the **GROUP** and **ASSUME** statements must appear at the beginning of the segment. These statements ensure that each variable declared in the **BSS** segment will be associated with the **DS** segment register and **DGROUP** (see Sections 3.6 and 3.7).

Note

The group name DGROUP must not be defined in more than one **GROUP** directive in a source file. If a source file contains both a DATA and a BSS segment, the directive

```
DGROUP  GROUP  _DATA,_BSS
```

should be used.

A **BSS** segment must have **word** align type, **public** combine type, and must have the class name **'BSS'**. These define loading instructions that are passed to the linker. Although other segment attributes are available, they must not be used. For a complete description of the attributes, see Sections 3.4.1–3.4.3.

Example

```
DGROUP  GROUP    _BSS
        ASSUME ds:DGROUP

_BSS    SEGMENT word public 'BSS'
```

```
count   DW      ?
array   DW      10 DUP(?)
string  DB      30 DUP(?)
_BSS    ENDS
```

C.6 Constant Segments

Syntax

DGROUP GROUP CONST
ASSUME ds:DGROUP
CONST SEGMENT word public 'CONST'
statements
CONST ENDS

A constant segment defines constant data that will not change during program execution. Constant segments are typically used in large-model programs to hold the segment values of far data segments.

The constant segment's name must be **CONST**. The segment can contain any combination of data *statements* defining constants to be used by the program. The segment must not exceed 64K. All data addresses in the segment are relative to the predefined group **DGROUP**. Therefore, the **GROUP** and **ASSUME** statements must appear at the beginning of the segment. These statements ensure that each variable declared in the constant segment will be associated with the **DS** segment register and **DGROUP** (see Sections 3.6 and 3.7).

Note

The group name DGROUP must not be defined in more than one **GROUP** directive in a source file. If a source file contains a DATA, BSS, and CONST segment, the directive

```
DGROUP GROUP _DATA,_BSS,CONST
```

should be used.

A constant segment must have **word** align type, **public** combine type, and must have the class name **'CONST'**. These define loading instructions that are passed to the linker. Although other segment attributes are available, they must not be used. For a complete description of the attributes, see Sections 3.4.1–3.4.3.

Example

```
DGROUP  GROUP   CONST
        ASSUME ds:DGROUP

CONST   SEGMENT word public 'CONST'
seg1    DW      ARRAY_DATA
seg2    DW      MESSAGE_DATA
CONST   ENDS
```

In this example, the constant segment receives the segment values of two far data segments: ARRAY_DATA and MESSAGE_DATA. These data segments must be defined elsewhere in the module.

Index (Reference Manual)

MICROSOFT®

16011 NE 36th Way, Box 97017, Redmond, WA 98073-9717

Software Problem Report

Name ____

Street ____

City ____________________ State __________ Zip ________

Phone ___________________________ Date _____________

Instructions

Use this form to report software bugs, documentation errors, or suggested enhancements. Mail the form to Microsoft.

Category

______ Software Problem

______ Software Enhancement

______ Documentation Problem (Document #__________)

______ Other

Software Description

Microsoft Product ______________________________________

Rev. __________ Registration # ____________________________

Operating System ____

Rev. __________ Supplier _________________________________

Other Software Used _______________________________________

Rev. __________ Supplier _________________________________

Hardware Description

Manufacturer ______________ CPU __________ Memory ______ KB

Disk Size ______″ Density: Sides:

Single ______ Single ______

Double ______ Double ______

Peripherals ____

Problem Description

Describe the problem. (Also describe how to reproduce it, and your diagnosis and suggested correction.) Attach a listing if available.

Microsoft Use Only

Tech Support ____________

Date Received ____________

Routing Code ____________

Date Resolved ____________

Report Number ____________

Action Taken: