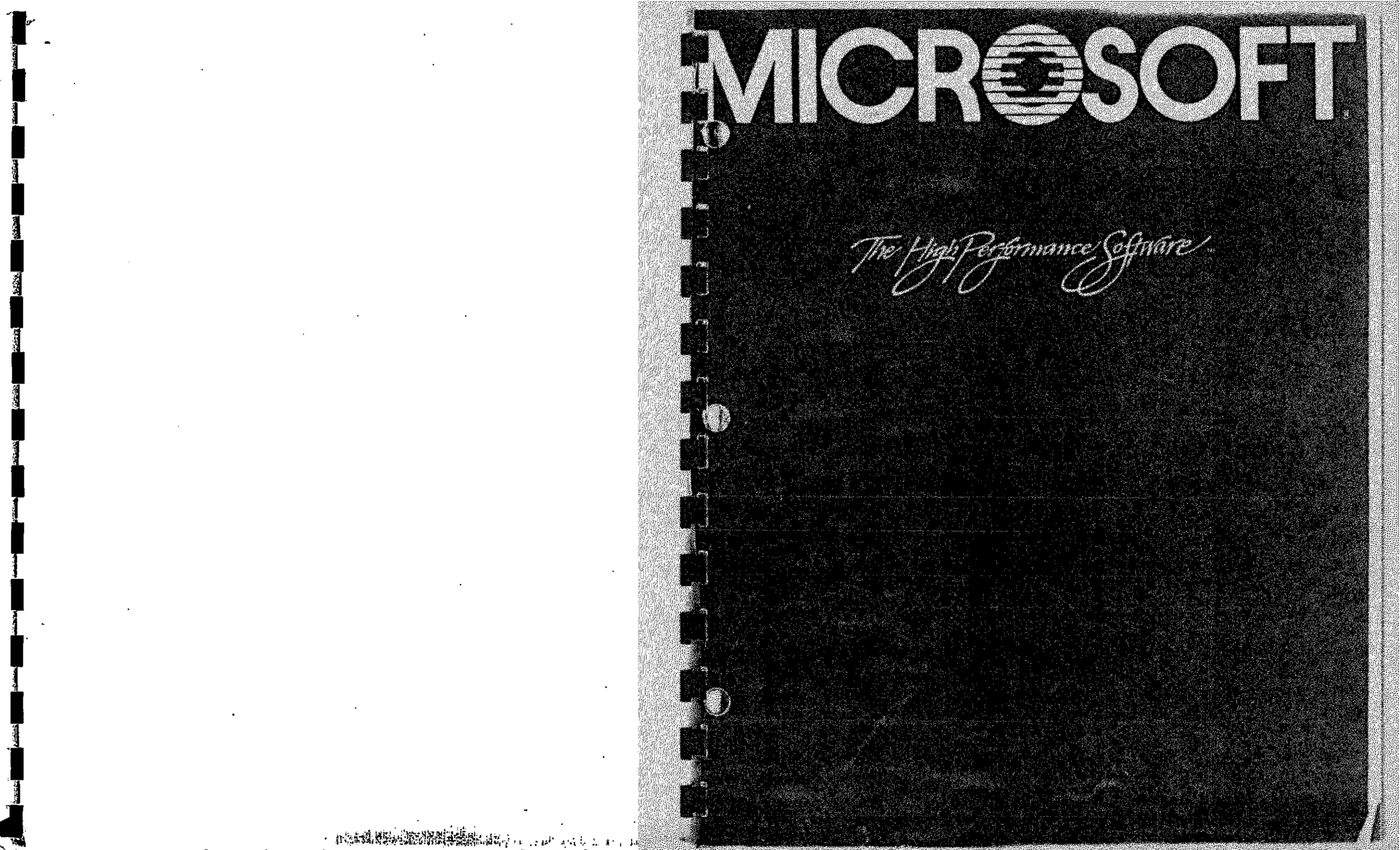

MICROSOFT
The High Performance Software

About the Microsoft C Compiler

Welcome to the Microsoft® C Compiler for MS®-DOS. Microsoft C is a full implementation of the C language, a language known for its efficiency, economy, and portability. The Microsoft C Compiler provides a simple command structure with a flexible set of options to accommodate all levels of programming experience.

With the Microsoft C Compiler, you can take advantage of C's strengths. Three different memory models are defined to let you set up your program in the most efficient way, taking advantage of the segmented architecture of the Intel® 8086 family of processors. In addition, the Microsoft C Compiler lets you combine features from different memory models in "mixed model" programs. Mixed model programs let you change addressing conventions for one or more program items without changing the addressing conventions for the program as a whole.

Included with your C compiler is a set of more than 200 run-time library routines that provides you with an extensive base of built-in functions for use in your C programs. The MS-DOS C run-time library is designed to make writing portable programs easier by providing compatibility with the XENIX® run-time library for 80286 systems.

In fact, compatibility with the 286 XENIX operating system is a built-in feature of the Microsoft C Compiler for MS-DOS. This compiler shares its design with the 286 XENIX C compiler, written by Microsoft Corporation and chosen by IBM® for its Personal Computer AT.

The Microsoft C Compiler offers advanced optimizing capabilities. Optimizing is performed automatically whenever you compile. Command line options are available to select alternative optimizing procedures or to turn off optimizing in the early stages of program development.

The compiler generates a broad range of error and warning messages to help you locate errors and potential problems. A special command line option lets you adjust the level of warning messages to suit your own needs.

Package Contents

Your Microsoft C Compiler package contains the following programs, stored on floppy disks.

- The compiler software
- The Microsoft LINK utility
- The Microsoft LIB utility
- EXEPACK, the executable file compression utility
- EXEMOD, the executable file header modification utility

Two documentation binders are included with the package.

System Requirements

To use the Microsoft C Compiler, your machine must run MS-DOS Version 2.0 or later. You must have two double-sided disk drives and a minimum of 256K (kilobytes) of memory (one kilobyte is 1,024 bytes). You must use Microsoft LINK Version 3.0 or later (included in this package). You cannot use earlier versions of Microsoft LINK with the compiler.

About These Manuals

The two documentation binders in your Microsoft C Compiler package hold the three manuals listed below.

Microsoft C Compiler User's Guide

The *C User's Guide* gives you the information you need to set up and operate the Microsoft C Compiler on your computer and explains how to compile, link, and run your C programs. Refer to the *C User's Guide* when you have questions about invoking the compiler and linker or about this particular implementation of C on MS-DOS.

Microsoft C Language Reference

The *C Language Reference* defines the C language as implemented by Microsoft. Use the *C Language Reference* when you have questions about the rules and behavior of the C language.

Microsoft C Run-Time Library Reference

The *C Library Reference* describes the run-time library routines provided for use in your C programs. The first part of the *C Library Reference* gives an overview of the run-time library, while the second section presents the routines in alphabetical order for quick reference.

3.1
1.1

User's Guide

Microsoft® C Compiler

User's Guide

for the MS®-DOS Operating System

Microsoft Corporation

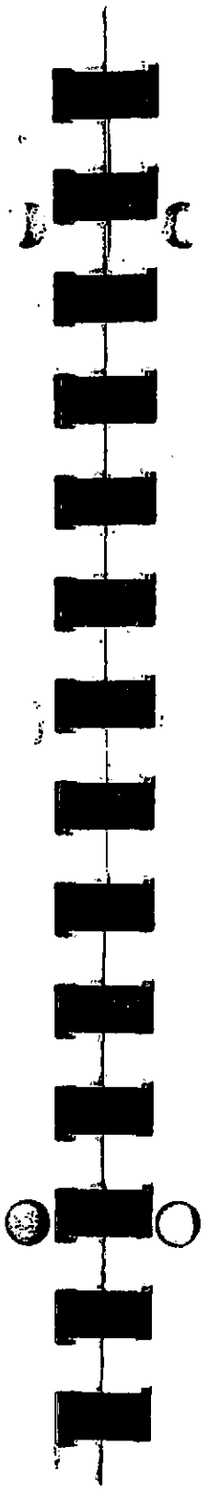
If you have comments about the software or this manual, complete the Software Problem Report at the back of this manual and return it to Microsoft Corporation.

Document Number 8415L-300-02
Part Number 048-014-025

Contents

Figures

Tables



Chapter 1
Introduction

1.1 Overview

The C language is a powerful general-purpose programming language that is capable of generating efficient, compact and portable code. The Microsoft® C Compiler for the MS®-DOS operating system is a full implementation of the C language as defined by its authors, Brian W. Kernighan and Dennis M. Ritchie, in *The C Programming Language*. Microsoft Corporation is actively involved in the development of the ANSI (American National Standards Institute) standard for the C language; this version of Microsoft C attempts to anticipate and conform to the forthcoming standard.

Microsoft C offers several important features to help you increase the efficiency of your C programs. You can choose between three standard memory models (small, medium and large) to set up the combination of data and code storage that best suits your program. For flexibility and even greater efficiency, the Microsoft C Compiler allows you to "mix" memory models by using special declarations in your program.

The C language does not provide such standard features as input and output capabilities and string manipulation features. These capabilities are provided as part of the run-time library of functions that accompanies the C installation. Because the functions that require interaction with the operating system (for example, input and output) are logically separate from the language itself, the C language is especially suited for producing portable code.

The portability of your Microsoft C programs is increased by the use of a common run-time library for MS-DOS and XENIX® installations. Using the routines in this library you can easily transport programs from a XENIX development environment to an MS-DOS machine, or vice versa. See the *Microsoft C Run-Time Library Reference* (included in this package) for more information on the common library for MS-DOS and XENIX.

Compared to other programming languages, C is extremely flexible about data conversions and nonstandard constructions. The Microsoft C Compiler offers several levels of warnings to help you control this flexibility. Programs in an early stage of development can be processed using the full warning capabilities of the compiler to catch mistakes and unintentional data conversions. The experienced C programmer can use a lower warning level for programs that contain intentionally nonstandard constructions.

1.2 About This Manual

This manual explains how to use the Microsoft C Compiler to compile, link, and run C programs on your MS-DOS system. The manual assumes that you are familiar with the C language and with MS-DOS and that you are able to create and edit a C language source file on your system. If you have questions about the C language, turn to the *Microsoft C Language Reference*, included in this package. The *Microsoft C Run-Time Library Reference* documents the run-time library routines you can use in your C programs. For more information about C, refer to Section 1.4, "Learning More About C." A brief description of the remaining chapters of the *C User's Guide* is given below.

Chapter 2, "Getting Started," covers installing and organizing the compiler software. This chapter explains how to set up an operating environment for the compiler by defining environment variables and includes a practice session to acquaint you with the Microsoft C Compiler.

Chapter 3, "Compiling," discusses the process of compiling a program using the basic compiler command MSC. This chapter contains a detailed description of the options most commonly used to control preprocessing, compilation, and output of files. The standard memory models (small, medium, and large) are discussed in this chapter.

Chapter 4, "Linking," describes the Microsoft LINK Object Code Linker and the options available to control its operation. This chapter includes a discussion of the special requirements that apply when linking C program files.

Chapter 5, "Running Your C Program," explains how to run your executable program file and how to pass data to a program at execution time.

Chapter 6, "Managing Libraries," describes the Microsoft LIB Library Manager. This utility enables you to create and maintain your own libraries of useful functions. You can use these libraries to customize the run-time support available to your programs.

Chapter 7, "Advanced Topics," describes additional command line options for the experienced programmer and gives the technical information necessary to use them. "Mixed model" programming (combining features from the three standard memory models) is discussed in this chapter, as well as the EXEPACK and EXEMOD utilities, which can be used to modify executable files.

Chapter 8, "Interfaces with Other Languages," covers two main topics: the interface between assembly language routines and C routines, and declarations of Microsoft Pascal and FORTRAN routines in C programs.

The appendices at the end of this manual contain useful reference material. Appendix A, "ASCII Character Codes," gives the ASCII decimal, octal, and hexadecimal equivalents for characters.

Appendix B, "Command Summary," provides a complete list of command line options for the MSC command and summarizes characteristics of the small, medium and large memory models. It also summarizes command characters and options for Microsoft LINK, Microsoft LIB, and the EXEPACK and EXEMOD utilities.

Appendix C, "The CL Command," describes an alternate command for invoking the compiler, the CL command. This command provides an interface that is similar to the XENIX and UNIX® "cc" command.

Appendix D, "Converting from Previous Versions of the Compiler," summarizes the differences between this version of the Microsoft C Compiler and previous versions. This appendix gives instructions for converting your existing programs to work under Version 3.0 and later.

Appendix E, "Error Messages," lists and describes the error messages generated by the compiler, the linker, and the library manager. It also lists and explains run-time error messages.

Appendix F, "Working with Microsoft Products," gives an overview of Microsoft products included in the C compiler package and in other Microsoft language packages, and explains how these products work together. In particular, it demonstrates how to use SYMDEB, the Microsoft Symbolic Debug Utility (not included in this package) with C programs.

Appendix G, "Microsoft LINK Technical Summary," is a technical discussion of the linker's operation.

1.3 Notational Conventions

The following notational conventions are used throughout this manual.

italics

Italics mark the places in command line and option specifications and in the text where specific terms appear in an actual command. For example, in

/W number

number is italicized to indicate that this is a general form for the /W option. In an actual command, the user supplies a particular number for the placeholder *number*.

Italics are also used when referring to specific identifiers supplied for functions, variables, types, and labels. For instance, when a program example such as

```
pc = &count;
```

is provided, the variable names *pc* and *count* are italicized in the discussion of the example.

Occasionally, italics are used to emphasize particular words in the text.

[brackets]

Brackets enclose optional fields in command line and option specifications. For example, in

/D identifier[=[*string*]]

the brackets around the phrase "= [*string*]" indicate that you are not required to supply this phrase when you use the /D option. Furthermore, within this phrase, *string* is enclosed in brackets. Thus, when you give an equal sign (=), the *string* is optional. Notice, however, that you may not give a *string* without first giving the equal sign.

The C language also uses brackets for array declarations and subscript expressions. In examples, brackets have the meaning specified by C. For instance,

```
a[10]
```

is an example showing a C subscript expression.

ellipses...

Ellipses following an item indicate that more items having the same form may appear. For example, in

LIB *library* [*/pagesize*] *operations*...

the ellipses indicate that one or more *operations* are allowed.

Vertical ellipses are also used in program examples to indicate that a portion of the program is omitted. For instance, in the following excerpt, three statements are shown. The ellipses between the statement indicate that intervening program lines occur but are not shown.

```
count = 0;
   .
   .
   .
*pc++;
   .
   .
   .
count = 0;
```

CAPITALS

Capital letters are used for the names of files, directories, environment variables, and manifest constants and macros. Commands typed at the MS-DOS level are also capitalized. These commands include built-in MS-DOS commands such as SET, as well as the MSC, LINK, and LIB commands, which invoke the compiler, linker, and library manager programs. You are not required to use capital letters when you actually enter these commands.

For example, in the command

```
SET TMP=B:\SCRATCH
```

the MS-DOS command SET is capitalized, as is the environment variable name TMP and the directory name to which it is set, B:\SCRATCH.

SMALL CAPITALS — Small capital letters are used for the names of keys and key sequences, such as RETURN and CONTROL-C.

"quotation marks" — Quotation marks set off terms defined in the text. For example, the term "far" appears in quotation marks the first time it is defined.

Quotation marks are also used to set off program fragments and to refer to command line prompts. For example, Microsoft LINK prompts you for the name of the executable file; this prompt is referred to as the "Run File" prompt.

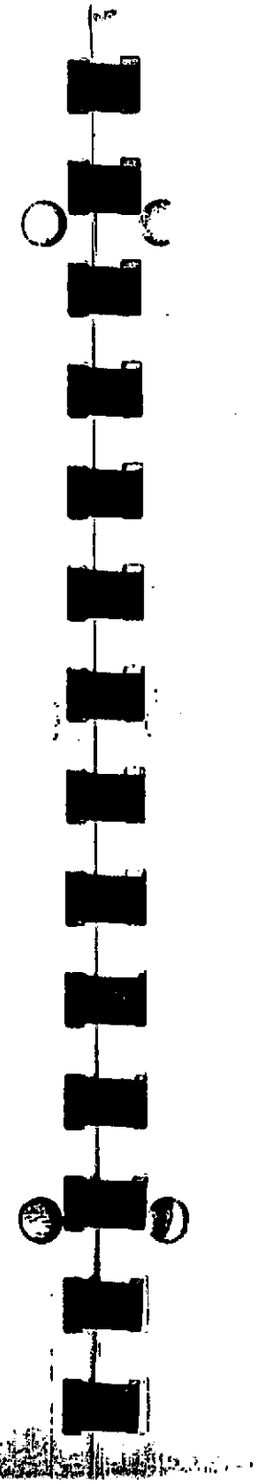
Some C constructs require quotation marks. Quotation marks required by the language have the form " " rather than “ ”. For example,

```
"abc"
```

is a C string.

keywords — C keywords, such as **goto** and **char**, are set in a different type font (the Helvetica font) to distinguish them from ordinary identifiers and text.

`programming examples` — Programming examples are displayed without proportional spacing so that they look similar to the programs you create with a text editor.



1.4 Learning More About C

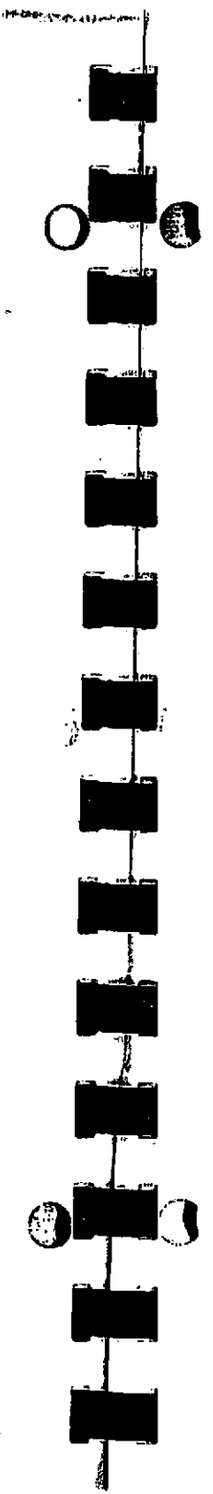
The three manuals in this documentation package provide a complete programmer's reference for Microsoft C. They do not, however, teach you how to program in C. If you are new to C or to programming, you may want to familiarize yourself with the language by reading one of the following books.

Hancock, Les and Morris Krieger. *The C Primer*. New York: McGraw-Hill Book Co., Inc., 1982.

Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1978.

Kochan, Stephen. *Programming in C*. Hasbrouck Heights, New Jersey: Hayden Book Company, Inc., 1983.

Plum, Thomas.. *Learning to Program in C*. Cardiff, New Jersey: Plum Hall, Inc., 1983.



Chapter 2
Getting Started

2.1 Introduction

This chapter explains how to install the compiler software and set up an operating environment for the compiler. It describes the files that make up your compiler package and suggests methods for organizing the files.

Several MS-DOS procedures are mentioned in this chapter. In particular, the MS-DOS SET and PATH commands are used to give values to "environment variables," which control the compiler environment. If you are unfamiliar with the SET and PATH commands or with other MS-DOS procedures mentioned in this chapter, consult your MS-DOS manual for instructions.

This chapter includes a sample disk setup for your files and a practice session to introduce you to the process of compiling and linking a program with the Microsoft C Compiler and Microsoft LINK utility. The practice session, while not required, allows you to confirm that your files are set up properly and provides a quick overview of the MSC and LINK commands.

2.2 Backing Up Your Disks

The first thing you should do when you have unwrapped your system disks is to make working copies, using the COPY or DISKCOPY utility supplied with MS-DOS. Save the original disks for backup.

2.3 Disk Contents

When you first open your compiler package, you may want to verify that you have a complete set of software. You should find the following files on your disks.

Executable Files

Filename	Description
MSC.EXE	Control program for the compiler
P0.EXE	Preprocessor
P1.EXE	Language parser
P2.EXE	Code generator
P3.EXE	Optimizer, link text emitter, and assembly listing generator
LINK.EXE	Linker utility, Microsoft LINK
LIB.EXE	Library manager utility, Microsoft LIB
EXEPACK.EXE	File compression utility
EXEMOD.EXE	File header modification utility
CL.EXE	Alternate control program for the compiler

Include Files

Filename	Description
ASSERT.H	Defines *assert* macro
CONIO.H	Declares console I/O functions
CTYPE.H	Defines character classification macros
DIRECT.H	Declares directory control functions
DOS.H	Defines data types and macros for DOS interface functions and declares DOS interface functions
ERRNO.H	Defines system-wide error numbers
FCNTL.H	Defines flags used in *open* functions
IO.H	Declares functions that work on file handles ("low-level" functions)
MALLOC.H	Declares memory allocation functions
MATH.H	Declares math functions and defines related constants
MEMORY.H	Declares buffer manipulation functions
PROCESS.H	Declares process control functions and defines flags for *spawn* functions
SEARCH.H	Declares searching and sorting functions
SETJMP.H	Declares and sets up storage for *setjmp* and *longjmp* functions
SHARE.H	Defines flags for file sharing

SIGNAL.H	Declares *signal* function and defines related constants
STDIO.H	Declares stream functions and defines related macros, constants, and types
STDLIB.H	Declares all functions from the C run-time library that are not declared in other include files
STRING.H	Declares string manipulation functions
TIME.H	Declares time functions and defines structure type used by time functions
V2TOV3.H	Defines macros to aid in converting programs from Microsoft C Versions 2.03 and earlier
SYS\LOCKING.H	Defines flags for file locking
SYS\STAT.H	Declares *stat* and *fstat* functions and defines *stat* structure type and related constants
SYS\TIMEB.H	Declares *ftime* function and defines the *timeb* structure type
SYS\TYPES.H	Defines types used for file status and time information
SYS\UTIME.H	Declares *utime* function and defines the *utimbuf* structure type

Library Files

Filename	Description
SLIBC.LIB	Small model standard C library
SLIBFP.LIB	Small model floating-point math library
SLIBFA.LIB	Small model alternate math library
MLIBC.LIB	Medium model standard C library
MLIBFP.LIB	Medium model floating-point math library
MLIBFA.LIB	Medium model alternate math library
LLIBC.LIB	Large model standard C library
LLIBFP.LIB	Large model floating-point math library
LLIBFA.LIB	Large model alternate math library
EM.LIB	Emulator floating-point library
87.LIB	8087/80287 floating-point library

Other Files

Filename	Description
BINMODE.OBJ	Routine for processing binary data.
SSETARGV.OBJ	Small model routine for processing wild card characters.
MSETARGV.OBJ	Medium model routine for processing wild card characters.
LSETARGV.OBJ	Large model routine for processing wild card characters.
DEMO.C	Sample C program.
README.DOC	Documentation of most recent changes and additions not appearing in this manual. If you see files on your disks that do not appear in the above list, they will be explained in the README.DOC file. Your release of the software may not include a README.DOC file, so don't be alarmed if you are unable to find this file on your disks.

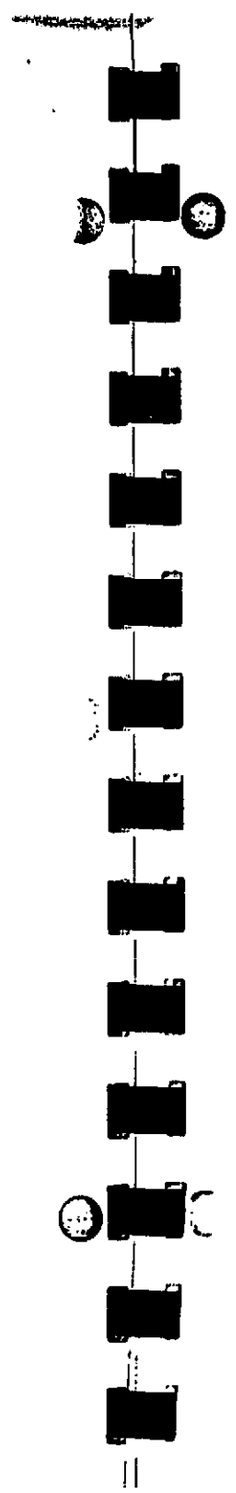
2.4 Understanding the Compiler Software

The software for the Microsoft C Compiler consists of three main categories of files: compiler executable files, include files, and library files. These files are listed in Section 2.3, "Disk Contents." Sections 2.4.1, 2.4.2, and 2.4.3, respectively, describe each of the three file categories in more detail. A number of additional files do not fall into the three main categories and are discussed separately in Section 2.4.4, "Other Files."

2.4.1 Executable Files

Executable files have an ".EXE" extension. MSC.EXE, the control program for the compiler, is an executable file. To run the compiler, invoke MSC.EXE by typing "MSC" or "msc".

P0.EXE, P1.EXE, P2.EXE, and P3.EXE are the four stages, or "passes," of the compiler. They are executed in order when you process a file using the compiler control program (MSC.EXE or CL.EXE).



The file LINK.EXE is the linker utility, Microsoft LINK. You invoke the linker by typing "LINK", after you have compiled a file or files. The linker produces an executable program file from your compiled files.

The library manager program, LIB.EXE, is used to create and organize libraries of object modules. You invoke this utility by typing "LIB".

EXEPACK.EXE and EXEMOD.EXE are special programs you can use to modify your executable program files. They are discussed in Sections 7.9.1 and 7.9.2, respectively, of Chapter 7, "Advanced Topics."

CL.EXE is an alternate control program for the compiler. It is provided for those users who are familiar with the cc command from XENIX or UNIX systems. Like MSC.EXE, CL.EXE invokes the four passes of the compiler for you. You can also invoke the linker through CL.EXE.

2.4.2 Include Files

Include files are text files you can incorporate into your program by using the C preprocessor directive **#include**. These files contain definitions used by run-time library routines.

By convention, some include files are stored in a subdirectory named SYS. This convention originated in the practice of storing files that define "system-level" constants and types in a separate "system" subdirectory on UNIX and XENIX systems. However, not all the include files that are traditionally stored in the SYS subdirectory contain system-level definitions, and some of the include files *not* in the SYS subdirectory contain system-level definitions. Since many programs, particularly those created under the XENIX and UNIX operating systems, rely on the SYS subdirectory convention, Microsoft continues to recognize this convention to maintain compatibility with existing programs.

2.4.3 Library Files

Library files contain compiled run-time library routines to be linked with your program. A separate set of library files is included for each standard memory model: small, medium, and large. The terms "small model," "medium model," and "large model" refer to standard memory models you can choose for your program, based on its storage requirements for code and data.

You do not have to choose a memory model in order to process and run your program. The small model is appropriate for most programs, and the compiler uses the small model and the small model library files by default.

Two additional library files, EM.LIB and 87.LIB, are model-independent; they can be used with all three memory models. EM.LIB is the floating-point emulator, used to perform floating-point operations. 87.LIB is the 8087/80287 floating-point library. This library provides minimal floating-point support and can only be used when an 8087 or 80287 coprocessor is present. The compiler uses the emulator (EM.LIB) by default, but you can override the default to use 87.LIB (if you have a coprocessor) or the alternate math library, described below. Floating-point options are described in more detail in Section 3.7 of Chapter 3, "Compiling," and in Section 7.7 of Chapter 7, "Advanced Topics."

The library files beginning with an "S" belong to the small model library set. SLIBC.LIB is the standard run-time library. SLIBC.LIB contains all the routines included in the Microsoft C run-time library except for math routines that require floating-point support.

SLIBC.LIB also contains an object module named CRT0.OBJ, which is the start-up routine for small model programs. The start-up routine performs several important tasks. It allocates the stack for your program and initializes the segment registers. It sets up the "argv," "argc," and "envp" variables to allow command-line arguments and environment settings to be passed to the program. The start-up routine is responsible for setting up and maintaining the operating environment for the program. The start-up routine also initializes the emulator, if loaded.

SLIBFP.LIB is the floating-point math library. It is required whenever your program uses EM.LIB or 87.LIB.

SLIBFA.LIB is the alternate floating-point library. You can use SLIBFA.LIB instead of EM.LIB and SLIBFP.LIB when speed is more important than precision in floating-point calculations. See the discussion of floating-point operations in Section 3.7 of Chapter 3, "Compiling," and in Section 7.7 of Chapter 7, "Advanced Topics," for details on this option.

When you compile a source file using MSC.EXE or CL.EXE, the compiler places the names of the standard library (SLIBC.LIB) and the floating-point libraries (EM.LIB and SLIBFP.LIB are the default) in the object file for the linker. Thus, LINK is able to link these libraries with your program automatically. If you compile using one of the /FP options, you can control which floating-point libraries are specified in the object files. You can also override the default at link time by substituting the name of a

different floating-point library for the library name in the object file. These options are discussed in Section 3.7 of Chapter 3, "Compiling," and in Section 7.7 of Chapter 7, "Advanced Topics."

The files beginning with an "M" are medium model library files, and the files beginning with "L" are large model library files. The organization and content of these files are analogous to that of the small model library set. MLIB.LIB and LLIBC.LIB, like SLIBC.LIB, each contain a start-up routine named CRT0.OBJ.

If you specify the medium or large model when you process your program, the compiler uses the appropriate standard library (MLIBC.LIB or LLIBC.LIB) and floating-point libraries (by default, EM.LIB plus MLIBFP.LIB or LLIBFP.LIB) when placing information in the object file for the linker. Otherwise, the compiler uses the small model files.

2.4.4 Other Files

The object file BINMODE.OBJ is provided for modifying the default mode for data files from text mode to binary mode. The same file can be used with all three memory models (see Section 7.10, "Controlling Binary and Text Modes," of Chapter 7, "Advanced Topics," for details on BINMODE.OBJ).

The SSETARGV.OBJ file provides a routine that expands the MS-DOS wild card characters "?" and "*" in filename arguments passed to C programs from the command line. SSETARGV.OBJ is the small model version of the routine; MSETARGV.OBJ and LSETARGV.OBJ are the medium and large model versions. Wild card expansion is performed only if you explicitly link with the appropriate SETARGV file. See Section 5.3, "Expanding Wild Card Arguments," in Chapter 5, "Running Your C Program," for details.

The README.DOC file, if present, contains documentation of recent changes that may not be included in this manual. If a README.DOC file is included on your disks, be sure to read the file before trying to use the software, since the file may contain information that affects how the compiler operates. In case of conflict between the manual and the README.DOC file, the README.DOC file takes precedence.

2.5 Setting Up the Environment

Before you compile and link a program using MSC.EXE and LINK.EXE, you must make sure that the programs can locate all the files they need to process your program. The required files are listed below.

Executable files	These are the files the control program executes as it processes your program. The names of these files are P0.EXE, P1.EXE, P2.EXE, and P3.EXE. When using CL.EXE, the alternate control program, LINK.EXE may also be executed by the control program. Notice that MSC.EXE and CL.EXE are also executable files.
Include files	If your program uses the preprocessor directive **#include**, the compiler attempts to find the given text file and include it in your program at compile time. Your program cannot be compiled if the given include file is not found.
Library files	At link time, LINK.EXE attempts to find the library files that are specified in the object file or on the link command line and link them with your program.

When you invoke the compiler or linker, it determines whether you have defined certain "standard places" to search for the necessary files. You can define these places by using environment variables. Environment variables are defined at the MS-DOS command level using the MS-DOS commands SET and PATH. They are called environment variables because they are effective throughout the environment in which a program is executed.

Although environment variables are usually helpful, you are not required to set them. If you do not set these variables, the current working directory is used to search for files and to create temporary files. An error is produced if the files are not found in the current working directory or if insufficient space is available in the directory for creating temporary files.

MSC.EXE looks for three environment variables: PATH, INCLUDE, and TMP. LINK.EXE uses one environment variable, LIB. The alternate control program, CL.EXE, uses all four environment variables.

PATH tells the compiler where to look for executable files, and INCLUDE tells it where to look for include files. The LIB environment variable tells LINK.EXE where to find any library files it needs.

The TMP environment variable has a slightly different function. The compiler creates a number of temporary files as it processes a program. The TMP environment variable tells the compiler where to create these files. The temporary files are removed by the time the compiler finishes processing. The space required for the temporary files is typically two times the size of the source file. It is often helpful to create the temporary files on another disk to avoid running out of space on your default disk.

Note

If you have a memory-based disk emulator (for example, the Microsoft RAMCard Memory Board), you can speed up processing by assigning it to the TMP variable.

To define the environment variables INCLUDE, LIB, and TMP, use the SET command to assign a directory specification or specifications to the variable. You must set PATH, INCLUDE, and TMP *before* invoking the compiler if you want the variables to be effective while the compiler is running. Similarly, you must set LIB before the linking stage.

The TMP variable can only be assigned one pathname. The INCLUDE and LIB variables can contain more than one pathname. Each pathname is separated from the next pathname by a semicolon (;). The compiler or linker searches through all directories specified, in order of their appearance, until it finds the file it needs. This means that include files and library files can be separated and placed in different directories.

For example, you can tell the compiler where to look for include files by setting the INCLUDE variable. Here is an example.

```
SET INCLUDE=B:\INCLUDE;B:\CUSTOM
```

The compiler will first look for include files on Drive B in the directory named INCLUDE; then, if necessary, the compiler will search the CUSTOM directory.

Use the PATH command instead of the SET command to define the PATH variable. (Although it is permissible to define the PATH variable with the SET command, using this method under versions of MS-DOS earlier than 3.0 can cause the PATH variable to work incorrectly for some directory specifications using lowercase letters.) To define the PATH variable using the PATH command, simply give the PATH command followed by a space (or an equal sign) and one or more directory specifications separated by semicolons. For example,

```
PATH A:\BIN;A:\LINKER
```

tells the compiler to search for executable files on Drive A in the directory named BIN, then, if necessary, in the LINKER directory.

MSC.EXE searches through all directories specified, in order of their appearance, until it finds the executable file it needs. Thus, executable files can be separated and placed in different directories, as long as the pathname of each directory containing an executable file appears in the PATH specification.

The MS-DOS operating system also uses the PATH setting to locate executable files. For example, when you invoke MSC.EXE (by typing "MSC"), the MS-DOS system locates MSC.EXE by looking in your default directory and in the directories specified in the PATH setting. If you include the pathname of the directory containing MSC.EXE (or CL.EXE) in your PATH setting, you can execute the control program from any directory.

Once you have set an environment variable, it remains effective until you reset it to a different value (or to an empty value) or until you turn off the machine. If you frequently set up your compiler files in a standard way, you should place SET and PATH commands in your AUTOEXEC.BAT file. Then you will be ready to use the compiler each time you boot your machine.

You can also use SET and PATH commands in an MS-DOS batch file to define the environment for a particular program or programs. If you frequently switch back and forth between different environments, you can save time by setting up batch files that contain the SET and PATH commands for each environment. Then you can just execute a batch file each time you want to switch to a new environment.

Certain command line options available with the compiler override the effect of environment variables. For example, the /X option (described in Section 3.5.6 of Chapter 3, "Compiling") tells the compiler not to automatically search the standard places for include files. The result is that the compiler does not search for include files in the directories specified by the INCLUDE variable.

2.6 Setting Up Your CONFIG.SYS File

Before you can run the compiler you must make sure that your CONFIG.SYS file allows the compiler to open at least 10 files. Check this by looking in your CONFIG.SYS file for the line

```
files=n
```

where *n* is some integer. If *n* is less than 10, edit CONFIG.SYS to set *n* to 10 (or greater). If you do not currently have a CONFIG.SYS file, create a file by that name and insert the following line.

```
files=10
```

It is recommended, but not required, that you also set the number of buffers allowed in your CONFIG.SYS file. Check your CONFIG.SYS for the line

```
buffers=n
```

where *n* is an integer. If *n* is not already set, 10 is a reasonable number.

After you have edited or created your CONFIG.SYS file, reboot the system so the new settings will take effect.

2.7 Using an 8087 or 80287 Coprocessor

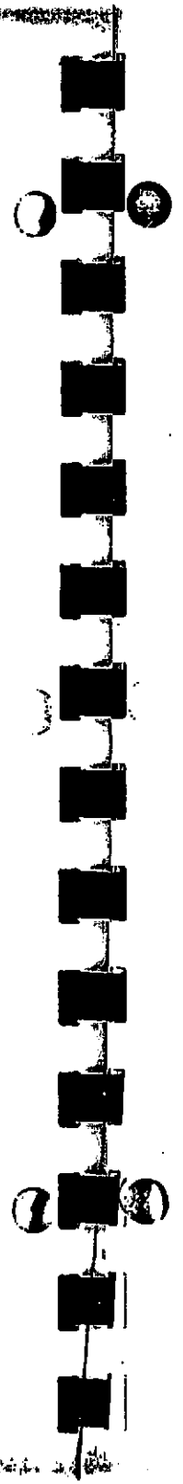
If you have an 8087 or 80287 coprocessor, you should read Section 3.7, "Selecting Floating-Point Options," in Chapter 3, "Compiling." With an 8087 or 80287, you can perform fast, efficient floating-point operations. You may want to select one of the 8087 options described in Section 3.7.1, "If You Have an 8087 or 80287 Coprocessor," to take maximum advantage of your processor's capabilities.

2.8 Using an 80186, 80188, or 80286 Processor

You can use the compiler with an 80186, 80188, or 80286 processor without taking any special steps. However, to take advantage of your processor's capabilities you will probably want to use the /G1 or /G2 option when you compile your programs. These options enable the instruction set for the 80186/80188 and 80286 processors respectively (see Section 3.8 of Chapter 3, "Compiling").

2.9 Converting Existing C Programs

If you own the Microsoft C Compiler Version 2.03 or earlier, or if you have programs written for that compiler, turn to Appendix D, "Converting from Previous Versions of the Compiler," for a discussion of differences between this compiler and earlier versions.



2.10 Organizing Your Software

Before you begin using the compiler, you will probably want to spend some time organizing the files on your disks. The optimal arrangement of files depends on your specific needs and on how you most frequently use the compiler, as well as your machine configuration. You can also take advantage of the compiler's use of environment variables to determine search paths for various pieces of the software.

It is recommended that you create a separate directory for each type of file: executable, include, and library. (Refer to your MS-DOS documentation if you are unfamiliar with the procedures for setting up and maintaining directories.) The "system-level" include files are conventionally placed in a separate subdirectory of the include file directory named SYS, but this is not required.

If you use the SYS subdirectory convention, you should give the subdirectory name along with the filename when you use a "system-level" include file in your program. For example, the line

```
#include <sys\timeb.h>
```

causes the compiler to look for the subdirectory named "sys" in the directory specified by the INCLUDE variable, and to include the file *timeb.h* from that subdirectory. On the other hand, if you do not use the SYS convention, the line

```
#include <timeb.h>
```

is sufficient.

Notice that, although case is significant within C programs, case is not significant to MS-DOS. The names "sys" and "SYS" are equivalent when used as MS-DOS directory names.

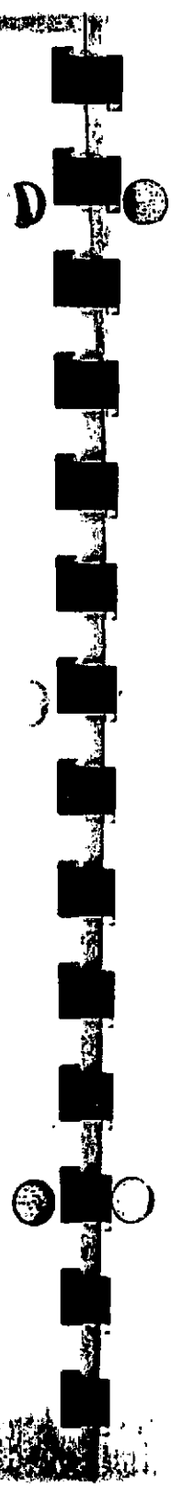
Sample setups for hard disk systems and floppy disk systems are given below. It is not necessary to read both sections; just read the one that applies to your system.

2.10.1 Sample Hard Disk Setup

The following sample setup is suitable for a hard disk system. The setup includes only the small model library files. If all your programs are small model, or if you are not concerned with memory models at all, then the small model library files are the only ones you need. On the other hand, if you use more than one memory model in your programming, you will probably want to add the appropriate library files to the LIB directory.

The 8087/80287 floating-point library and the alternate math library are not included in the sample setup because you do not need both the regular floating-point library and the other floating-point libraries at the same time. If you want to use one of the other floating-point libraries, you can substitute it or add it to the LIB directory. Similarly, only the MSC.EXE control program is included in this setup. If you prefer to use CL.EXE, add it to the BIN directory or substitute it for MSC.EXE.

```
C:\BIN\MSC.EXE
C:\BIN\P0.EXE
C:\BIN\P1.EXE
C:\BIN\P2.EXE
C:\BIN\P3.EXE
C:\BIN\LINK.EXE
C:\BIN\LIB.EXE

C:\INCLUDE\ASSERT.H
C:\INCLUDE\CONIO.H
C:\INCLUDE\CTYPE.H
C:\INCLUDE\DIRECT.H
C:\INCLUDE\DOS.H
C:\INCLUDE\ERRNO.H
C:\INCLUDE\FCNTL.H
C:\INCLUDE\IO.H
C:\INCLUDE\MALLOC.H
C:\INCLUDE\MATH.H
C:\INCLUDE\MEMORY.H
C:\INCLUDE\PROCESS.H
C:\INCLUDE\SEARCH.H
C:\INCLUDE\SETJMP.H
C:\INCLUDE\SHARE.H
C:\INCLUDE\SIGNAL.H
C:\INCLUDE\STDIO.H
C:\INCLUDE\STDLIB.H
C:\INCLUDE\STRING.H
C:\INCLUDE\TIME.H
C:\INCLUDE\V2TOV3.H

C:\INCLUDE\SYS\LOCKING.H
```


```
C:\INCLUDE\SYS\STAT.H
C:\INCLUDE\SYS\TIMEB.H
C:\INCLUDE\SYS\TYPES.H
C:\INCLUDE\SYS\UTIME.H

C:\LIB\SLIBC.LIB
C:\LIB\SLIBFP.LIB
C:\LIB\EM.LIB
```

Using this setup, your environment variables can be given the values shown below.

```
PATH C:\BIN
SET INCLUDE=C:\INCLUDE
SET LIB=C:\LIB
SET TMP=C:\
```

Notice that the TMP setting simply specifies the root directory of Drive C. The temporary files created by the compiler are removed by the time processing is completed, so you don't need to create a separate directory to store them. MSC.EXE deletes the temporary files automatically; you are not responsible for removing them.

With this sample setup you can run the compiler and linker from any directory or disk. If you use the EXEPACK.EXE and EXEMOD.EXE utilities, put them in the BIN directory. Then you can execute the programs from any directory.

If you use one of the SETARGV files (SSETARGV, MSETARGV, or LSETARGV, depending on the memory model) to enable wild card expansion, or the BINMODE.OBJ file to change the default text processing mode, you can place the file either in your C program file directory or in the LIB directory. Notice, however, that the LIB environment variable is not used to find the SETARGV or BINMODE file; if it is not in your current working directory, you must specify a pathname at link time.

2.10.2 Sample Floppy Disk Setup

You will need at least two floppy disks to set up the files so that you can run the compiler. The sample setup given below uses two disks and assumes that you will swap these two disks in and out of Drive A as necessary. You can develop your programs and create listing files on a separate disk in Drive B.

This sample setup includes only the small model library files. You can save space by keeping only one set of library files on a disk, since any given program uses only one set (either the small, the medium, or the large model set). If all your programs are small model, or if you are not concerned with memory models at all, then the small model library files are the only ones you need.

The 8087/80287 floating-point library and the alternate math library are not included in this sample setup because you do not need both the regular floating-point library and the other floating-point libraries at the same time. If you want to use one of the other floating-point libraries, you can substitute it. Similarly, only the MSC.EXE control program is included in this setup. If you prefer to use CL.EXE instead, substitute it for MSC.EXE.

```
Disk 1:
        BIN\MSC.EXE
        BIN\P0.EXE
        BIN\P1.EXE
        BIN\P2.EXE
        BIN\P3.EXE

        INCLUDE\ASSERT.H
        INCLUDE\CONIO.H
        INCLUDE\CTYPE.H
        INCLUDE\DIRECT.H
        INCLUDE\DOS.H
        INCLUDE\ERRNO.H
        INCLUDE\FCNTL.H
        INCLUDE\IO.H
        INCLUDE\MALLOC.H
        INCLUDE\MATH.H
        INCLUDE\MEMORY.H
        INCLUDE\PROCESS.H
        INCLUDE\SEARCH.H
        INCLUDE\SETJMP.H
        INCLUDE\SHARE.H
        INCLUDE\SIGNAL.H
        INCLUDE\STDIO.H
        INCLUDE\STDLIB.H
        INCLUDE\STRING.H
        INCLUDE\TIME.H
        INCLUDE\V2TOV3.H

        INCLUDE\SYS\LOCKING.H
        INCLUDE\SYS\STAT.H
        INCLUDE\SYS\TIMEB.H
        INCLUDE\SYS\TYPES.H
```



```
        INCLUDE\SYS\UTIME.H

Disk 2:
        BIN\LINK.EXE
        BIN\LIB.EXE

        LIB\SLIBC.LIB
        LIB\SLIBFP.LIB
        LIB\EM.LIB
```

With this setup, all the files required in the compiling stage are on Disk 1, and all the files required in the linking stage are on Disk 2. This organization allows you to change disks easily when you have finished compiling a file and are ready to link.

Using this sample setup, your environment variables can be given the values shown below, assuming that Disks 1 and 2 will be swapped in and out of Drive A.

```
PATH A:\BIN
SET INCLUDE=A:\INCLUDE
SET LIB=A:\LIB
SET TMP=B:\
```

With this setup, you should create and store your programs on a separate disk on Drive B. You should also run the compiler from Drive B, so that B is the default drive for output files (the object file, listing file, map file, and executable program file). If you try to run the compiler from Drive A, there may be insufficient space available for creating these files. For the same reason, the TMP variable is assigned to Drive B.

Notice that the TMP setting simply specifies the root directory of Drive B. The temporary files created by the compiler are removed by the time processing is completed, so you don't need to create a separate subdirectory to store them. MSC.EXE deletes the temporary files automatically; you are not responsible for removing them.

If you use the EXEPACK.EXE and EXEMOD.EXE utilities, put them in the BIN directory on Disk 2. Then you can execute them from your program disk in Drive B.

If you use one of the SETARGV files (SSETARGV, MSETARGV, or LSETARGV, depending on the memory model) to enable wild card expansion, or the BINMODE.OBJ file to change the default text processing mode, you can place the file either in the directory with your C program files or in the LIB directory. Notice, however, that the LIB environment

variable is not used to find the SETARGV or BINMODE file; when it is not in your current working directory, you must specify a pathname at link time.

If you use more than one memory model in your programming, you will probably want to set up a separate library disk for each model. Notice that the files stored on Disk 1 (the compiler passes and the include files) do not change with the memory model, so you can use the same disk in the compiling stage for all three models.

On each separate library disk you will have the library files for that model, plus a copy of the LINK and LIB utilities. Although the LINK and LIB utilities do not change with the memory model, it is convenient to have a copy on each disk so that you can invoke LINK and LIB without changing back to your small model disk.

Use the same directory structure on all three disks (small, medium, and large). That way, you will not have to change the values of your environment variables when you change disks. For example, to process a medium model program using the alternate math library instead of the emulator, you could set up a disk like the following, to be used in Drive A.

```
BIN\LINK.EXE
BIN\LIB.EXE

LIB\MLIBC.LIB
LIB\MLIBFA.LIB
```

This organization is identical to the setup for Disk 2 given earlier, except that the medium model standard library file replaces the small model file, and the medium model alternate math library (MLIBFA.LIB) is used instead of EM.LIB and SLIBFP.LIB. The PATH setting (A:\BIN) and TMP setting (B:\) used above are valid for this disk as well, since it is organized with the same directory structure. Notice that you must use the same disk drive, Drive A, when you change from the small model disk to the medium model disk. Otherwise, your environment settings become invalid.

2.11 Practice Session

This section shows you the steps involved in compiling and linking a program using the Microsoft C Compiler. By following these steps you can produce and run an executable program file.

The source file used for this practice session is the sample source file DEMO.C, which is included with your compiler software. DEMO.C is a very simple C program that contains only one function, the *main* function. The *main* function is designed to print on your terminal any command line arguments you pass to the program at execution time. It will also print out the current value of environment settings. You can examine the DEMO.C source file to see how it accomplishes this. For a full discussion of passing command line data to programs, accessing the program environment from within a program, and declaring the "argc," "argv," and "envp" parameters, see Chapter 5, "Running Your C Program."

This practice session assumes that you are using the sample disk setup and environment that is appropriate for your system. The sample setup and compiler environment was discussed in Section 2.10, "Organizing Your Software."

The first thing you should do is verify that the compiler environment is set up correctly. You can do this by typing SET. When you give the SET command without an argument, it lists all environment variables and their current settings. Make sure the PATH, INCLUDE, TMP, and LIB variables are in the list and that they are set appropriately for your system, as shown below.

Hard Disk Settings	Floppy Disk Settings
PATH=C:\BIN	PATH=A:\BIN
INCLUDE=C:\INCLUDE	INCLUDE=A:\INCLUDE
LIB=C:\LIB	LIB=A:\LIB
TMP=C:\	TMP=B:\

If your settings do not match these, turn back to the previous section to review the disk setup and environment settings.

Once you have set up the environment, you are ready to begin processing DEMO.C using steps 1-13 below.

1. First, set up a directory to hold program files. The directory can be on the hard disk or on a floppy disk. You can give the directory any name you like; for this session, the name PROG will be used. Next, copy DEMO.C into the PROG directory.
2. Now you are ready to begin compiling. Make sure that the PROG directory is your current working directory (use the CD command to change directories if necessary.) Then give this command:

   ```
   MSC
   ```

   The MSC command invokes MSC.EXE, the compiler control program. MSC.EXE displays prompts on your screen to guide you through the compiling process.
3. The first message to appear on your screen is

   ```
   Microsoft C Compiler  Version 3.xx
   (C)Copyright Microsoft Corp 1984 1985
   Source filename [.C]:
   ```

   Following the "Source filename" prompt, specify the name of the file or files to be compiled. If you don't include the filename extension, MSC.EXE assumes that the extension is ".C" (or ".c"). Type

   ```
   DEMO
   ```

   in response to this prompt.
4. The next prompt is

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

   This prompt allows you to supply a name for the object file. Instead of typing a name, respond to this prompt by pressing the RETURN key, causing MSC.EXE to use the default response for the prompt. The default response for the "Object filename" prompt is to name the object file DEMO.OBJ. The object file is created in the current working directory, which is the PROG directory.
5. The next prompt is

   ```
   Object listing [NUL.COD]:
   ```


This prompt lets you create a listing of your object file, containing the machine instructions that correspond to your C instructions. Type

```
DEMO
```

in response to this prompt. MSC.EXE appends the default extension ".COD" and creates a listing named DEMO.COD. The listing file is created in the current working directory (PROG).

6. MSC.EXE now begins to compile your program. If your program has errors, they will be displayed as the compiler operates. (DEMO.C does not have errors.) When the compilation process is finished, the MS-DOS prompt reappears.

   You now have an object file named DEMO.OBJ and a listing file named DEMO.COD in your current working directory.
7. Next you need to link your program.

Important

If you are using a floppy disk setup, you should change the disk in Drive A at this point. Remove the disk containing the compiler files and include files, then insert the disk containing the LINK utility and the library files.

To link your file, simply type

```
LINK
```

The LINK command invokes the linker. You will see the following message on your screen.

```
Microsoft 8086 Object Linker
```

The message is followed by a version number and copyright notice.

8. The first linker prompt is

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

   You have only one object file to link, so just type

```
DEMO
```

in response to this prompt. Microsoft LINK appends the ".OBJ" extension to find your file on the disk. Since the file is in the current working directory, you do not have to specify a pathname for LINK to find it.

9. The next prompt is

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

This prompt lets you name the executable program file. Press the RETURN key in response to this prompt. The linker uses the default name shown in brackets for the executable file if you don't supply a different name. The executable file is created in the current working directory (PROG).

10. The next prompt is

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

If you give a filename following this prompt, the linker creates a map file listing all the external symbols in your program and their locations. Type the response

```
DEMO /MAP
```

This response tells the linker to create a listing file named DEMO.MAP. The ".MAP" extension is used because you did not supply your own extension. The map file is created in the PROG directory by default. The /MAP option causes global symbols to be listed at the end of DEMO.MAP.

11. The final prompt is

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

The names of the standard C and floating-point libraries are provided in the object file, and the LIB environment variable tells the linker where to find the given library files. Therefore, you do not need to give any library names following this prompt. Just press the RETURN key.

12. Microsoft LINK now proceeds to link your file. If any errors are found, they are displayed on your screen. When the MS-DOS system prompt reappears, the linker has finished processing your file. You now have an executable file named DEMO.EXE in your directory, plus an object listing named DEMO.MAP.

    You may want to examine the object listing (DEMO.COD) and map file (DEMO.MAP) to familiarize yourself with their formats. These files are especially useful for debugging programs. However, the listing and map files are not required for running the program, so you can delete them if you like.

    You can also delete the object file (DEMO.OBJ); since you have the executable program file, it is no longer needed. Chapter 6, "Managing Libraries," discusses how to use the Microsoft LIB program to organize object files into libraries of useful functions.

13. You can run the sample program simply by typing "DEMO". However, since the sample program is designed to take command line arguments and print them, you will probably want to give command line arguments when you run the program. For instance, you can run the program and pass three arguments by typing:

```
DEMO ONE TWO THREE
```

The program name is displayed on your screen, followed by the arguments ONE, TWO, and THREE and a listing of all current environment settings. The environment settings include PATH, LIB, INCLUDE, and TMP, as well as any other settings that are currently in effect (whether or not they apply to the C program or to the compilation and linking process).

Note

Under versions of MS-DOS earlier than 3.0, the program name is not available and will not be displayed.

This practice session used the simplest form of the MSC and LINK commands to show you their basic operation. The chapters that follow describe alternate forms and explain how to specify options with the MSC, LINK, and LIB commands. Notice that the CL command, described in Appendix C, "The CL Command," can be used to perform the same tasks as MSC and LINK.

2.12 Using Batch Files

You can create an MS-DOS batch file to set up the compiler environment and invoke the compiler. Creating and using batch files is discussed in full in your MS-DOS manual. This section is intended simply to demonstrate possible uses of the MSC command in a batch file.

A batch file is a text file containing a series of executable MS-DOS commands. Batch files always have the extension ".BAT". You execute a batch file by typing the filename without the ".BAT" extension. This causes MS-DOS to execute the series of commands the file contains.

Batch files are especially useful with the MSC command because they allow you to set up an environment before using the command. The examples below use the command line method of invoking MSC and Microsoft LINK. The command line method lets you give all responses to the prompts on a single line instead of waiting for the individual prompts. The command line method is discussed in Section 3.2.8 of Chapter 3, "Compiling," and in Section 4.4.10 of Chapter 4, "Linking."

For example, the following batch file, MYCOMP.BAT, could be used to create a program from a C source file in an environment set up for that purpose.

```
SET INCLUDE=B:\TOP\MYINC
MSC %1;
IF ERRORLEVEL 0 LINK %1,,%1;
```

The value given to INCLUDE in the first line alters the environment for the MSC command. Since no value is given for PATH, TMP, or LIB, their current values, if set, are unaffected by the batch file.

The symbol "%1" tells MS-DOS to look for an argument on the command line when you execute the batch file. When you type

```
MYCOMP THIS
```

the filename THIS is substituted for "%1", and THIS.C is compiled, producing the object file THIS.OBJ.

The line

```
IF ERRORLEVEL 0 LINK %1,,%1;
```


ensures that linking is only attempted if the source file was successfully compiled. The MSC and CL control programs return an exit code to allow testing for successful compilation. The exit code 0 indicates success; for information on the other codes, see Section 3.9.3, "Compiler Exit Codes," in Chapter 3, "Compiling." The MS-DOS batch command IF ERRORLEVEL is used to test the exit code; see your MS-DOS documentation for more on this command.

If compilation is successful, the object file THIS.OBJ is linked to produce THIS.EXE (the default name, since none is supplied). The name THIS is also supplied (by means of the symbol "%1") for the map file prompt, so a map file named THIS.MAP is produced.

Notice that the value given to INCLUDE when you execute the batch file remains in effect until you explicitly change it or until you reboot your machine. To restore your usual environment settings, you can create a batch file that resets the environment variables to the directories you most frequently use. For example, the following lines might be placed in a file called RESET.BAT, to be executed by typing RESET whenever you want to restore your usual environment settings.

```
PATH A:\HOME\BIN
SET INCLUDE=A:\INCLUDE
SET LIB=A:\LIB
SET TMP=B:\
```

Chapter 3
Compiling

3.1 Introduction

One basic command, MSC, is all you need to compile your C source files with the Microsoft C Compiler. The MSC command takes care of executing the four compiler passes for you.

By drawing on the large set of MSC options, you can control and modify the tasks performed by the command. For example, you can direct MSC to create an object listing file or a preprocessed listing. Options also let you give information that applies to the compilation process, such as the definitions for manifest (symbolic) constants and macros and the kinds of warning messages you want to see.

The MSC command automatically optimizes your program. You never have to give an optimizing instruction, unless you want to change the way that MSC optimizes or you want to disable optimization altogether. See Section 3.11, "Optimizing," for more on these choices.

This chapter explains how to run the compiler using the MSC command and discusses commonly used MSC options in detail.

Additional MSC options are covered in Chapter 7, "Advanced Topics." A summary of the MSC command and all available options is provided in Section B.2 of Appendix B, "Command Summary," of this guide. Appendix C, "The CL Command," is a summary of the CL command, an alternative to the MSC command. The CL command is similar to the cc command on XENIX and UNIX systems, and is included for users who are comfortable with the XENIX cc command.

This chapter assumes that you know how to create, edit, and debug C program files on your system. For questions relating to the definition of the C language, see the *Microsoft C Language Reference*.

3.2 Running the Compiler

MSC requires two types of input: a command to start the compiler and responses to command prompts. Start the compiler by typing

```
MSC
```

at the MS-DOS command level. MSC prompts for the input it needs by displaying the following three messages, one at a time.

```
Source filename [.C]:
Object filename [.OBJ]:
Object listing  [NUL.COD]:
```

The responses you make to each prompt are explained below.

If you want to stop the compiling session for any reason, type CONTROL-C at any time. You will be returned to the MS-DOS command level, where you can restart MSC from the beginning.

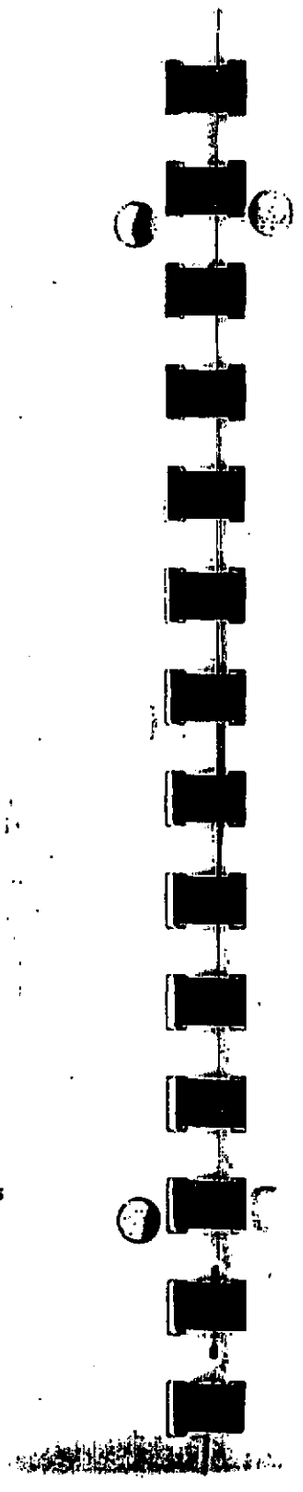
3.2.1 Filename Conventions

You can use uppercase, lowercase, or a combination of both for the filenames you give in response to the prompts. For example, the following three filenames are considered equivalent.

```
abcde.fgh
AbCdE.FgH
ABCDE.fgh
```

You can include spaces before or after filenames, but not within them. Options (see Section 3.2.9) can appear anywhere spaces can appear.

MSC uses the default file extensions ".C", ".OBJ", and ".COD" when you do not supply extensions with your filenames. You can override the default extension for a particular prompt by specifying a different extension. To enter a filename that has no extension, type the name followed by a period. For example, typing "ABC." in response to a prompt tells MSC that the specified file has *no* extension, while typing just "ABC" tells MSC to use the default extension for that prompt.



You can override any defaults by typing all or part of the name. For example, if the currently logged drive is B and you want the output file to be written to the disk in Drive A, type just the response "A:". The output file is written on Drive A with the default filename.

Notice that if you type any part of a legal pathname following the "Object listing" prompt, MSC produces a listing file. The default name is the "basename" of the source file with the extension ".COD". The basename of a file is the portion of the name preceding the period (.). For example, if you compile a file named TEST.C and type "A:" following the "Object listing" prompt, MSC produces a listing file on Drive A with the name A:TEST.COD.

3.2.2 Special Filenames

You can use the following MS-DOS device names as filenames with the MSC command. This allows you to direct files to your terminal or to a printer. Notice that you cannot use these names for ordinary filenames.

Name	Device
AUX	Refers to an auxiliary device (such as a printer or disk drive).
CON	Refers to the console (terminal).
PRN	Refers to the printer device.
NUL	Specifies a "null" (nonexistent) file. Giving NUL as a filename means that no file is created.

Even if you add device designations or filename extensions to these special filenames, they remain associated with the devices listed above. For example, A:CON.XXX still refers to the console and is not the name of a disk file.

Note

Object files contain machine code and are not printable. When responding to the "Object filename" prompt, do not give a filename that refers to a printer or console.

3.2.3 Source Filename Prompt

Following the "Source filename" prompt, give the name of the source file you want to compile. If you do not supply an extension, MSC automatically looks for a file with the ".C" extension.

Pathnames are allowed with the source filename. Therefore, you can specify the pathname of a source file in another directory or on another disk.

You may compile only one file at a time, so only one response to this prompt is allowed. There is no default response; MSC displays an error message if you do not supply a source filename.

3.2.4 Object Filename Prompt

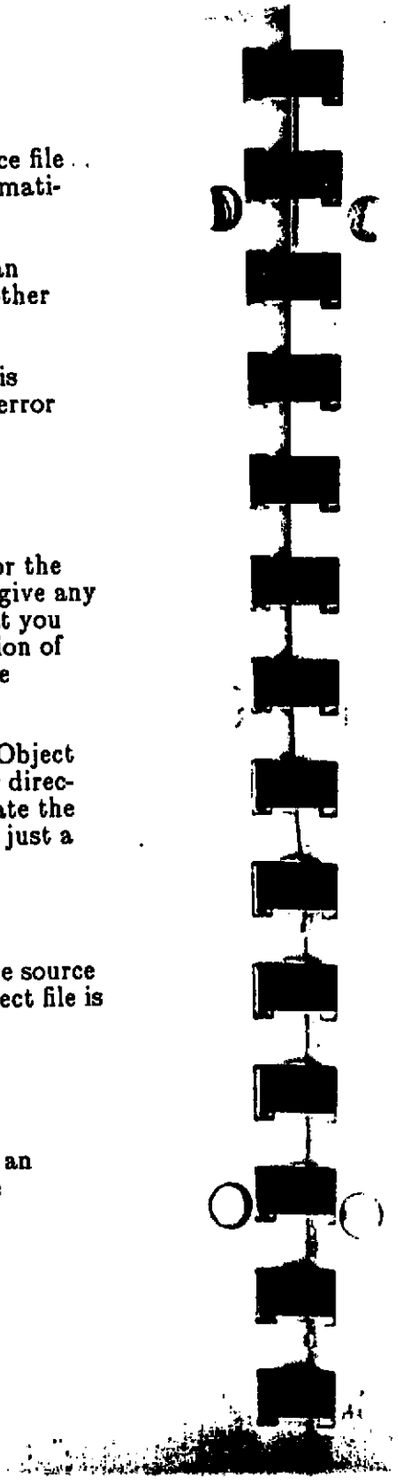
Following the "Object filename" prompt, you can supply a name for the object file produced by compiling your source file. You are free to give any name and any extension you like. However, it is recommended that you use the conventional ".OBJ" extension because it simplifies operation of Microsoft LINK and Microsoft LIB, both of which use ".OBJ" as the default extension when processing object files.

If you supply just a drive or directory specification following the "Object filename" prompt, MSC creates the object file in the given drive or directory and uses the default filename. You can use this option to create the object file in another directory or on another disk. When you give just a directory specification, the directory specification must end with a backslash (\) so that MSC can distinguish between a directory specification and a filename.

The default name supplied for the object file is the basename of the source file with an ".OBJ" extension. If no pathname is supplied, the object file is created in the current working directory.

3.2.5 Object Listing Prompt

Following the "Object listing" prompt you can tell MSC to create an object listing for the compiled file. The object listing contains the machine instructions and assembled code for your program.



If you supply any filename following this prompt, MSC creates an object listing, using the filename you supply. By convention, these listings are given the extension ".COD", but you are free to choose any extension you like.

When you do not supply a filename, the default is the special name NUL.COD, which tells MSC *not* to create a listing.

The MSC command optimizes by default, so the object listing reflects the optimized code. Since optimization may involve rearrangement of code, the correspondence between your source file and the machine instructions may not be clear. To produce a listing without optimizing, use the /Od option, discussed in Section 3.10, "Preparing for Debugging."

To produce a combined source and assembly code listing, use the /Fc option, described in Section 3.4, "Producing Listing Files."

3.2.6 Selecting Default Responses

To select the default response to the current prompt, press the RETURN key without giving any other response. The next prompt will appear.

To select default responses to all remaining prompts, use a single semicolon (;) after the filename following the "Source filename" or "Object filename" prompt. Once the semicolon has been entered, you cannot respond to any of the remaining prompts for that compiling session. Any text appearing after the semicolon (such as an option) is ignored. Use the semicolon to save time when the default responses are acceptable.

There is no default for the first prompt, "Source filename". The default for the "Object filename" is the basename of the source file with an ".OBJ" extension. The default for the "Object listing" prompt is the special name NUL.COD, which tells MSC *not* to create an object listing file.

3.2.7 Swapping Disks

MSC suspends execution and displays a prompt whenever it cannot find one or more of the executable files that make up the compiler: P0.EXE, P1.EXE, P2.EXE, and P3.EXE. This behavior lets you store the compiler files on different disks if necessary and swap disks when MSC prompts you.

If you respond to the "Source filename" prompt with a nonexistent filename, or to the "Object filename" or "Object listing" prompts with an invalid pathname, MSC displays an error message and terminates. You must restart MSC with the correct information.

3.2.8 Using the Command Line

Once you understand how the MSC prompts and responses work, you can use the command line method of running the compiler. With this method you type all the filenames on the line used to start MSC. The command line method has the following form.

```
MSC sourcename [,[objectname]] [,[listingname]] [;]
```

The entries following MSC are responses to the command prompts.

You can include spaces before or after filenames, but not within them. Options (described in Section 3.2.9) can appear anywhere spaces can appear.

You can leave the *objectname* and *listingname* fields blank to cause MSC to select the default responses. The semicolon (;) character has the same effect in the command line as it does with the MSC prompts. When MSC sees a semicolon on the command line, it uses the default responses to the remaining prompts. Any text after the semicolon on the command line is ignored.

The comma character serves as a separator and also has a special function in the command line. If you place a comma after the *objectname* field in the command line (whether or not an *objectname* is actually given), the default for the listing field is changed from NUL.COD to the basename of the source file plus ".COD". For example, the following two command lines are equivalent.

```
MSC TEST, TEST, TEST;
MSC TEST, , ;
```

In the first command line, the name TEST is explicitly specified for all three prompts, so TEST.C is compiled and two files are produced: TEST.OBJ and TEST.COD.

In the second command line, only the source filename is supplied. The default name (TEST.OBJ) is used for the object filename, since none is specified. The comma following the object filename field causes the default for the listing file to be changed to TEST.COD. Since no alternative name is supplied in the command line, a listing file named TEST.COD is created.

By contrast, the line

```
MSC TEST;
```

creates an object file named TEST.OBJ, but does not create a listing file, since no comma is present in the command line to change the default from NUL.COD to TEST.COD.

You can combine the prompt method and command line methods by giving MSC a partial command line. It prompts you for the fields you do not supply. You can end a partial command line with any of the items listed in the first column below. The second column describes the results.

semicolon (;)	MSC uses the default responses for the remaining prompts.
filename	MSC prompts you for the remaining responses, if any.
comma	If you give just a source filename followed by a comma, MSC prompts for both object filename and object listing name, as usual. However, if you supply both a source filename and an object filename, and then terminate the command line with a comma, MSC changes the default object listing name from NUL.COD to the basename of the source file plus ".COD". MSC then prompts you for an object listing name to allow you to override the default. (You can give the name NUL.COD to suppress the creation of an object listing.)

Options can also appear at the end of a partial command line, as discussed in the next section. The following examples demonstrate partial command lines.

Examples

1. `MSC ASK.C, TELL.OBJ`
2. `MSC ASK, TELL;`
3. `MSC ASK.C, TELL.OBJ,`
4. `MSC ASK`

Example 1 causes MSC to prompt with

```
Object listing[NUL.COD]
```

since you supplied the source filename and object filename but not the listing filename.

Notice the difference between Example 1 and Example 2, which tells MSC to use the default response (no file) for the object listing. No further prompts appear in this case.

In Example 3, the trailing comma (after TELL.OBJ) has a special meaning. It causes MSC to prompt as follows.

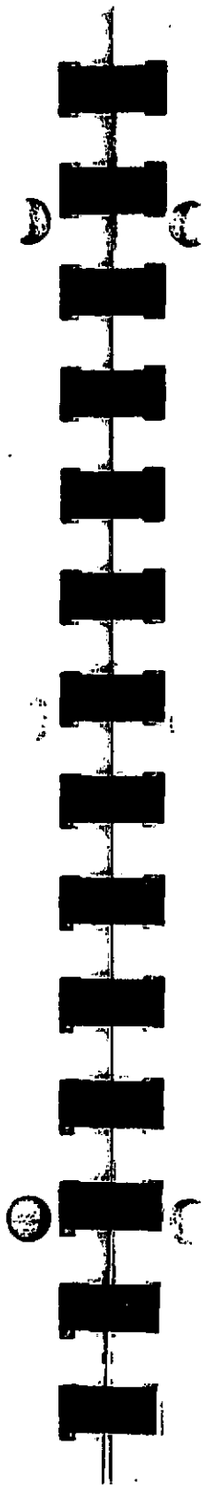
```
Object listing[TEST.COD]:
```

Notice that the default name in brackets is TEST.COD rather than NUL.COD. In this case an object listing is created by default, unless you override the default to specify a different listing name (or the name NUL.COD, to suppress the listing).

In Example 4, MSC starts prompting with the "Object filename" prompt, since only the source filename is supplied.

3.2.9 Options

The MSC command offers a large number of command options to control and modify the compiler's operation. Options begin with the forward slash character (/) and contain one or more letters. The hyphen character (-) can be used instead of the forward slash if you prefer. For example, /Zg and -Zg are both acceptable forms of the Zg option.



Important

Although filenames can be given in either uppercase or lowercase, *options must be given exactly as shown in this manual.* For example, /W and /w are two different options.

Options can appear anywhere a space can appear when you give the MSC command, except that options following a semicolon are ignored. Thus, options can go before or after any of the three filenames (source filename, object filename, and object listing.) The options apply to the entire compilation process, not just to the line on which they appear.

Some options take arguments, such as filenames, strings, or numbers. In most of these cases, spaces are allowed between the option letter and the argument. For example, these are both acceptable forms of the /W option:

```
/W 3
/W3
```

The /Gt option and /F family of options (/Fa, /Fc, /Fl, and /Fo, plus /Fe and /Fm with the CL command) form the only exceptions to this rule. The /Gt option accepts an optional numerical argument, while the /F options accept an optional pathname or partial pathname argument. When you supply an argument to one of these options, no spaces may appear between the option and the argument. For example,

```
/FcMINGLE
```

is acceptable, but

```
/Fc MINGLE
```

is not.

Some options consist of more than one letter. For example, the /F options mentioned above are two-letter options. No spaces are allowed between the letters of an option. Thus, in the above example, no spaces can appear between "F" and "c".

The order of the options is not important, and they can be given following any prompt or in any command line field. The default for the prompt is still used if you supply an option but no filename in response to the prompt.

The compiler options and the tasks they perform are discussed in the remainder of this chapter and in Chapter 7, "Advanced Topics." The command line form of the MSC command is used for the examples of options in this manual. Remember that you can use options with the prompts as well, as shown below.

Examples

```
1.  MSC
    Source filename [.C]:  A:\LOAD.C
    Object filename [LOAD.OBJ]: OUT
    Object listing  [NUL.COD]: /Oas /Fc

2.  MSC A:\LOAD.C /Oas /FoOUT /Fc;
```

The two examples above produce exactly the same effect. The source file LOAD.C on Drive A is compiled. The object file is named OUT.OBJ. The /Fc option produces a combined source and assembly code listing; since no argument was given with the /Fc option, the listing is given the default name LOAD.COD, formed by appending ".COD" to the basename of the source file. The object file and combined listing are both created on the default drive, since no drive was specified. The /Oas option tells the compiler how to optimize the object file. The Fc and Oas options are discussed in detail in Section 3.4, "Producing Listing Files," and Section 3.11, "Optimizing," respectively.

3.3 Naming the Object File

Option

*/Fo**objectname***

You can name the object file produced by compiling your source file using the /Fo option. Using this option has the same effect as giving a filename at the "Object filename" prompt. When using the /Fo option, the *objectname* argument must appear immediately after the option, with no intervening spaces.

You are free to supply any name and any extension you like for the *objectname*. However, it is recommended that you use the conventional ".OBJ" extension because it simplifies operation of Microsoft LINK and Microsoft LIB, both of which use ".OBJ" as the default extension when processing object files. If you give an object filename without an extension, MSC automatically appends the ".OBJ" extension.

If you give just a drive or directory specification following the /Fo option, MSC creates the object file in the given drive or directory and uses the default filename (the basename of the source file plus ".OBJ"). You can use this option to create the object file in another directory or on another disk. When you give just a directory specification, the directory specification must end with a backslash (\) so that MSC can distinguish between a directory specification and a filename.

If you give a name following the "Object filename" prompt and also use the /Fo option, the name you give after the /Fo option overrides the name you give following the prompt.

Examples

```
1.  MSC THIS, B:\OBJECT\;

2.  MSC THIS /FoB:\OBJECT\;
```

The two examples above produce exactly the same effect. The source file THIS.C is compiled; the resulting object file is named THIS.OBJ (by default). The directory specification "B:\OBJECT\" tells MSC to create THIS.OBJ in the given directory on Drive B.

3.4 Producing Listing Files

Options

/Fl [listingname]
/Fa [listingname]
/Fc [listingname]

You can create a listing of your compiled source file by responding to the "Object listing" prompt when you run MSC or by using the /Fl option. The object listing shows the machine instructions and assembled code for your program.

The /Fa listing produces an assembly listing of your program. The assembly listing contains the assembly code corresponding to your C file. The listing is suitable as input to the Microsoft Macro Assembler, also called Microsoft MASM.

To produce a listing that shows your source program along with the assembly code, use the /Fc option. This option produces a line-by-line combined source and assembly code listing, showing one line of your source program followed by the corresponding line (or lines) of machine instructions.

When using the /Fa, /Fc, and /Fl options, the *listingname*, if given, must follow the option immediately, with no intervening spaces. The *listingname* can be any one of the items listed in the first column below. The second column describes the results. If the *listingname* does not include an extension, the default extension is used. The default extension is ".COD" for the /Fc and /Fl options and ".ASM" for the /Fa option.

Filename	MSC uses the given filename, appending the default extension if the filename has no extension. The filename can include a path to tell MSC where to create the listing.
Directory specification	MSC creates the object listing in the given directory, using the default listing name, which is formed by appending the default extension to the basename of the source file. The directory specification must end with a backslash (\) so that MSC can distinguish between a directory specification and a filename.

Omitted	When no *listingname* is given with the /Fl, /Fa, or /Fc option, MSC uses the default listing name (basename of the source file plus the default extension) and creates the listing in the current working directory.

The MSC command optimizes by default, so listing files reflect the optimized code. Since optimization may involve rearrangement of code, the correspondence between your source file and the machine instructions may not be clear, especially when you use the /Fc option to mingle the source and assembly code. To produce a listing without optimizing, use the /Od option (discussed in Section 3.11, "Optimizing") along with the listing option.

When you examine a listing file, you will notice that the names of globally visible functions and variables begin with an underscore. The Microsoft C compiler automatically prefixes an underscore to all global names to preserve compatibility with XENIX C compilers. If you write assembly language routines to interface with your C program, this naming convention is important; see Section 8.1.6, "Naming Conventions," in Chapter 8, "Interfaces with Other Languages."

In the listing file, you may also see names that begin with more than one underscore. Identifiers with more than one leading underscore are reserved for internal use by the compiler. You should not attempt to use these identifiers in your program. Moreover, you should avoid creating global names that begin with an underscore in your C source files. Since the compiler automatically adds another leading underscore, these names end up with two leading underscores, possibly causing conflicts with the names reserved by the compiler.

At most one listing file is produced each time you compile. The /Fc option overrides other listing options; whenever you use /Fc, a combined listing is produced.

Example

```
MSC HELLO.C, , /FcHELLO.LST;
```

The example creates a combined source and assembly code listing file named HELLO.LST and an object file named HELLO.OBJ from the source file HELLO.C.

3.5 Controlling the Preprocessor

The MSC command provides a number of options that give you control over the operation of the C preprocessor. You can define macros and manifest (symbolic) constants from the command line, change the search path for include files, and stop compilation of a source file after the preprocessing stage to produce a preprocessed source file listing. The options that perform these tasks are described below.

The C preprocessor recognizes only preprocessor directives. It treats the source file as a text file, processing substitutions and definitions as directed. The preprocessor can be run on a file at any stage of development, whether or not the file is a complete C source file. In fact, the preprocessor is not restricted to processing C files; it can be run on any kind of file. See the *Microsoft C Language Reference* for a complete discussion of C preprocessor directives.

3.5.1 Defining Constants and Macros

Option

/Didentifier[=[*string*]]

The /D option lets you define a constant or macro used in your source file. The *identifier* is the name of the constant or macro and the *string* is its value or meaning.

If you leave out both the equal sign and the *string*, the given constant or macro is assumed to be defined, and its value is set to 1. For example, /DSET is sufficient to define SET.

If you give the equal sign with an empty string, the given constant or macro is considered defined; its definition is the empty string. This definition effectively removes all occurrences of the identifier from the source file. For example, "/Dregister=" removes all occurrence of "register" from the source file. Notice that the identifier "register" is still considered to be defined.

The effect of using the /D option is the same as using a preprocessor `#define` directive at the beginning of your source file. The identifier is defined throughout the source file being compiled.

You can supply a command line definition for an identifier that is also defined within the source file. The command line definition holds up to the point of the redefinition in the source file.

Up to 16 definitions may appear on the command line, each preceded by the /D option. If you need to define more than 16 identifiers, see the discussion of the /U and /u options in Section 3.5.3, "Removing Definitions of Predefined Identifiers."

Example

```
MSC MAIN.C /D NEED=2;
```

The example defines the manifest constant NEED in the source file MAIN.C. Notice that spaces are permitted (but not required) between /D and the identifier. This definition is equivalent to placing the directive

```
#define NEED    2
```

at the top of the source file.

The /D option is especially useful with the `#if` directive. You can use the option to control compilation of statements in the source file. For example, suppose a source file named OTHER.C contains the following fragment.

```
#if defined(NEED)
    .
    .
#endif
```

Suppose further that OTHER.C does not explicitly define NEED (that is,

no #define directive for NEED is present). Then all statements between the #if and the #endif directives are compiled only if you supply a definition of NEED by using /D. For instance, the command

```
MSC MAIN.C /DNEED;
```

is sufficient to compile all statements following the #if directive. Notice that NEED does not have to be set to a specific value to be considered defined. The following command, on the other hand, causes the statements in the #if block to be ignored (not compiled).

```
MSC MAIN.C;
```

3.5.2 Predefined Identifiers

The compiler defines four identifiers that are useful in writing portable programs. You can use these identifiers to compile code sections conditionally, depending on the current processor and operating system. The predefined identifiers and their functions are listed below.

Identifier	Function
MSDOS	Always defined. Identifies target operating system as MS-DOS.
M_I86	Always defined. Identifies target machine as a member of the I86 family.
M_I86*x*M	Always defined. Identifies memory model, where *x* is either S (small model), M (medium model), or L (large model). Small model is the default. Memory models are discussed later in this chapter.
SS_NE_DS	Defined only when memory model options are used to set up separate stack and data segments. Memory model options are discussed in Section 3.13, "Working with Memory Models," of this chapter and in Section 7.11, "Mixed Model Programming," of Chapter 7, "Advanced Topics."

3.5.3 Removing Definitions of Predefined Identifiers

Options

/U identifier
/u

The /U (for "undefine") option can be used to turn off the definition of one or more of the predefined identifiers discussed in the previous section. The /u option turns off all four definitions.

These options are useful if you want to give more than 16 definitions on the command line, or if you have other uses for the predefined identifiers. For each definition of a predefined identifier you remove, you can substitute a definition of your own on the command line. When the definitions of all four predefined identifiers are removed, you can specify up to 20 command line definitions.

Example

```
MSC WORK /U MSDOS /U M_I86 /U M_I86SM;
```

This example removes the definitions of three predefined identifiers. Notice that the /U option must be given three times to do this.

3.5.4 Producing a Preprocessed Listing

Options

/P
/E
/EP

The /P, /E, and /EP options produce listings of preprocessed files. These options allow you to examine the output of the C preprocessor.

The preprocessed listing file is identical to the original source file except that all preprocessor directives are carried out, macro expansions are performed, and comments are removed. All three options suppress compilation; no object file or listing is produced, even if you supply a name following the "Object filename" or "Object listing" prompt.

The /P option writes the preprocessed listing to a file with the same basename as the source file but with a ".I" extension.

The /E option copies the preprocessed listing to the standard output (usually your terminal), and places a #line directive in the output at the beginning and end of each included file. You can save this output by redirecting it to a file, using the MS-DOS redirection symbol ">" or ">>" (see your MS-DOS manual for a description of these symbols).

The /E option is useful when you want to resubmit the preprocessed listing for compilation. The #line directives renumber the lines of the preprocessed file so that errors generated in later stages of processing refer to the original source file rather than the preprocessed file.

Using the /EP option combines features of the /E and /P options: the file is preprocessed and copied to the standard output, but no #line directives are added.

Examples

```
1.  MSC MAIN.C /P;

2.  MSC ADD.C /E ;  > PREADD.C

3.  MSC ADD.C /EP ;
```

The first example creates the preprocessed file MAIN.I from the source file MAIN.C. The second command creates a preprocessed file with inserted #line directives from the source file ADD.C. The output is redirected to the file PREADD.C. The third command produces the same preprocessed output as the second example without the #line directives. The output appears on the screen.

3.5.5 Preserving Comments

Option

/C

Normally comments are stripped from a source file in the preprocessing stage, since they do not serve any purpose in later stages of compiling. The /C (for "comment") option preserves comments during preprocessing. The /C option is valid only when the /E, /P, or /EP option is also used.

Example

```
MSC SAMPLE.C /P /C;
```

The example produces a listing named SAMPLE.I. The listing file contains the original source file, including comments, with all preprocessor directives expanded or replaced.

3.5.6 Searching for Include Files

Options

/I directory
/X

The /I and /X options temporarily override or change the effects of the environment variable INCLUDE. These options let you give a particular file special handling without changing the compiler environment you normally use. (See Section 2.5, "Setting Up the Environment," in Chapter 2, "Getting Started," for a discussion of environment variables.)

You can add to the standard places for include files by using the /I (for "include") option. This option causes the compiler to search the directory you specify before searching the standard places given by the INCLUDE environment variable. You can add more than one include directory by giving the /I option more than once in the MSC command. The directories are searched in order of their appearance in the command line.

The directories are searched only until the specified include file is found. If the file is not found in the given directories or the standard places, the compiler prints an error message and stops processing. When this occurs you must restart compilation with a corrected directory specification.

You can prevent the C preprocessor from searching the standard places for include files by using the /X (for "exclude") option. When MSC sees the /X option, it considers the list of standard places to be empty. This option is often used with the /I option to define the location of include files that have the same names as include files found in other directories, but that contain different definitions. See the second example, below.

Examples

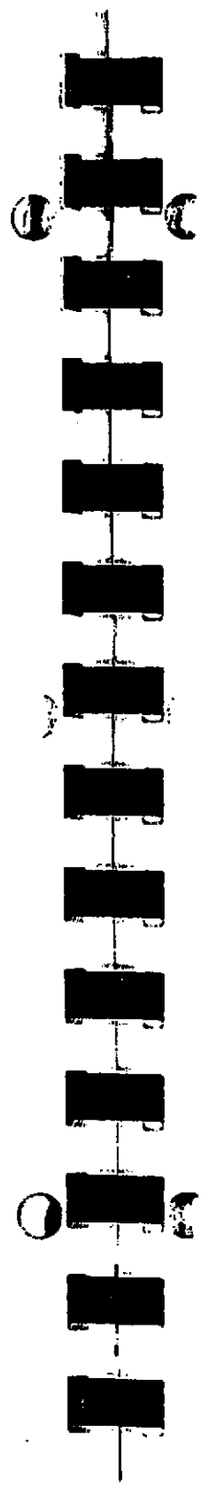
```
1.  MSC MAIN.C /I A:\INCLUDE /IB:\MY\INCLUDE;

2.  MSC MAIN.C /X /I B:\ALT\INCLUDE;
```

The first command directs the compiler to search for include files requested by MAIN.C first in the directory A:\INCLUDE, second in the directory B:\MY\INCLUDE, and finally in the directory or directories assigned to the INCLUDE environment variable.

In the second example, the compiler looks for include files only in the directory B:\ALT\INCLUDE. First the /X option tells MSC to consider the list of standard places empty; then the /I option specifies one directory to be searched.

3.6 Syntax Checking

The options described in this section are useful in the early stages of program development. They allow you to identify syntax errors and argument type mismatches quickly, without having to compile the source file.



3.6.1 Identifying Syntax Errors

Option

/Zs

The /Zs option causes the compiler to perform a syntax check only. No code is generated and no object file is produced. If the source file has syntax errors, error messages will be displayed.

This option provides a quick way to locate and correct syntax errors before attempting to compile a source file.

Example

```
MSC /Zs PRELIM.C;
```

This command causes the compiler to perform a syntax check on PRELIM.C, displaying messages about any errors it finds.

3.6.2 Generating Function Declarations

Option

/Zg

The /Zg option generates a function declaration for each function defined in the source file. The function declaration includes the function return type and an argument type list created from the types of the formal parameters of the function. Any function declarations already present in the source file are ignored.

The generated list of declarations is written to the standard output. It can be saved in a file using the MS-DOS redirection symbol ">" or ">>".

When the /Zg option is used, the source file is not compiled. As a result, no object file or listing is produced.

The list of declarations is helpful for verifying that actual arguments and formal parameters of a function are compatible. You can save the list and include it in your source file to cause the compiler to perform type-checking. The presence of a declared argument type list for a function "turns on" the compiler's type-checking between actual arguments to a function (given in the function call) and the formal parameters of a function.

This type-checking can be a helpful feature in writing and debugging C programs, especially when working with older C programs. Argument type-checking is a recent addition to the C language, so existing C programs do not have argument type lists. See the *Microsoft C Language Reference* for details on function declarations and argument type lists.

You can use the /Zg option even if your source program already contains some function declarations. The compiler accepts more than one occurrence of a function declaration, as long as the declarations do not conflict. No conflict occurs when one declaration has an argument type list and another declaration of the same function does not, as long as the declarations are identical otherwise.

Your program may include calls to Microsoft C run-time library routines. The include files provided with the Microsoft C run-time library contain function declarations so that you can enable type-checking on library calls. The declarations are enclosed in preprocessor #ifdef blocks and are included only if you define the special identifier LINT_ARGS. You can define LINT_ARGS either with a #define directive in your program or by using the /D option when you compile.

Example

```
MSC FILE.C /Zg;
```

The above command causes the compiler to generate argument type lists for functions defined in FILE.C.

3.7 Selecting Floating-Point Options

Options

/FPa	Generates floating-point calls and selects alternate math library
/FPc	Generates floating-point calls and selects emulator library
/FPc87	Generates floating-point calls and selects 8087/80287 library
/FPi	Generates in-line instructions and selects emulator library
/FPi87	Generates in-line instructions and selects 8087/80287 library

The Microsoft C compiler offers several methods of handling floating-point operations. This section provides an overview of the floating-point options available and discusses the default floating-point behavior. For more detailed information on the floating-point libraries, plus a discussion of overriding floating-point options at link time and using the NO87 environment variable, see Section 7.7, "Controlling Floating-Point Operations," in Chapter 7, "Advanced Topics."

The Microsoft C Compiler can use an 8087 or 80287 coprocessor if one is present and can emulate 8087 operation through the use of an emulator library if not. The emulator library (EM.LIB) provides a large subset of the functions of an 8087/80287 in software. The emulator can perform basic operations to the same degree of accuracy as an 8087/80287. However, the emulator routines used for transcendental math functions differ slightly from the corresponding 8087/80287 functions, causing a slight difference (usually within 2 bits) in the results of these operations when performed with the emulator instead of with an 8087/80287.

By default, the Microsoft C compiler handles floating-point operations by making calls to the emulator library (this is the /FPc option). The emulator library is loaded, but if an 8087 or 80287 coprocessor is present at run time, the coprocessor will be used instead of the emulator. This method of handling floating-point operations always works, whether or not you have a coprocessor installed. Thus, you do not have to give a floating-point option at compile time unless you want to use one of the other options described below.

When you compile a source file using one of the floating-point options, the name of the required floating-point library (or libraries) is placed in the object file. At link time, the linker refers to the names in the object file to link with the appropriate libraries. You can override the library name given in the object file at link time and link with a different library instead; see Section 7.7.1, "Changing Libraries at Link Time," in Chapter

7, "Advanced Topics," for details. The only restriction on overriding at link time is that you are not allowed to change to the alternate math library after you have compiled using the /FPi or /FPi87 option.

3.7.1 If You Have an 8087 or 80287 Coprocessor

The /FPi87 option is the fastest and smallest option available for floating-point operations. It generates in-line instructions for an 8087/80287 coprocessor and selects the 8087/80287 library (87.LIB), plus SLIBFP.LIB, MLIBFP.LIB, or LLIBFP.LIB, depending on the memory model. An 8087 or 80287 *must* be present at run-time if the /FPi87 option is used.

The /FPc87 option generates function calls to routines in the 8087/80287 library (87.LIB) that perform the corresponding 8087/80287 instructions. The 8087/80287 library (87.LIB) plus SLIBFP.LIB, MLIBFP.LIB, or LLIBFP.LIB, depending on the memory model, are selected. The /FPc87 option is slower than /FPi87 because it makes function calls instead of using in-line instructions. However, /FPc87 is more flexible. Using the /FPc87 option allows you to change your mind at link time (without recompiling the file) and use either the emulator or the alternate math library instead of relying on an 8087/80287 coprocessor. This is possible because the calls to 8087/80287 instructions are interchangeable with calls to the emulator and the alternate math library. See Section 7.7.1 of Chapter 7, "Advanced Topics," for instructions on changing libraries at link time.

Both the /FPi87 and /FPc87 options select the 8087/80287 library (87.LIB), which provides minimal floating-point support. Whenever 87.LIB is used, an 8087 or 80287 coprocessor must be present at run time. If no coprocessor is present, the program will not run and the message

```
Floating point not loaded
```

will appear.

The /FPi option generates in-line instructions for an 8087/80287 and selects the emulator library (EM.LIB), plus SLIBFP.LIB, MLIBFP.LIB, or LLIBFP.LIB, depending on the memory model. If an 8087/80287 coprocessor is present at run time, it will be used. If not, the emulator is used.

Loading the emulator requires approximately 7K of additional space, so programs that use the /FPi option are larger than programs that use /FPi87. However, /FPi is a particularly useful option when you do not know in advance whether an 8087 or 80287 coprocessor will be available at run time.

In some cases, you may not want to use an 8087 or 80287 coprocessor, even though one is present. For example, you may be developing programs to run on systems that lack coprocessors. Conversely, you may want to write programs that can take advantage of an 8087/80287 at run time, even though you don't have one installed. There are several ways to control the use of an 8087 or 80287:

1. Use the /FPc (default) or /FPi option to specify use of an 8087/80287 if present, and use of the emulator if not. To use the emulator even when an 8087 or 80287 is present, set the NO87 environment variable, as discussed in Section 7.7.2 of Chapter 7, "Advanced Topics."
2. Use the /FPc87 or /FPi87 option if you always want to use a coprocessor. Programs compiled with these options will fail if a coprocessor is not present at run time.

3.7.2 If You Don't Have a Coprocessor

The /FPi option generates in-line instructions for an 8087/80287 coprocessor and selects the emulator library (EM.LIB), plus SLIBFP.LIB, MLIBFP.LIB, or LLIBFP.LIB, depending on the memory model. If an 8087/80287 is present at run time, it will be used. If not, the emulator library, which mimics the operation of an 8087, is used. Because this option uses in-line instructions, it is the most efficient way to get maximum precision in floating-point operations without a coprocessor.

The /FPc option is the default when you do not specify a floating-point option. It generates floating-point calls to the emulator library and selects the emulator library (EM.LIB), plus SLIBFP.LIB, MLIBFP.LIB, or LLIBFP.LIB, depending on the memory model. The /FPc option is slower than /FPi because it makes function calls instead of using in-line instructions. However, /FPc is more flexible than /FPi. Using the /FPc option allows you to change your mind at link time (without recompiling the file) and use an 8087/80287 coprocessor or the alternate math library instead of using the emulator. This is possible because the same function call interface is provided in all three libraries: the 8087/80287 library, the alternate math library, and the emulator library. See Section 7.7.1 of

Chapter 7, "Advanced Topics," for instructions on changing libraries at link time.

The /FPa option generates floating-point calls and and selects the alternate math library (SLIBFA.LIB, MLIBFA.LIB, or LLIBFA.LIB, depending on the memory model). The alternate math library uses a subset of the IEEE (Institute of Electrical and Electronics Engineers, Inc.) standard format numbers, sacrificing some accuracy for speed and simplicity. (Infinities, NaN's, and denormal numbers are not used.) Calls to this library provide your fastest and smallest option if you do not have an 8087 or 80287 coprocessor. With this option, as with the /FPc option, you can change your mind at link time and use the emulator or an 8087/80287 instead; see Section 7.7.1 of Chapter 7, "Advanced Topics," for details.

In some cases, you may want to write programs that will be able to take advantage of an 8087 or 80287 at run time, even though you don't have one installed. See Section 3.7.1, "If You Have an 8087 or 80287 Coprocessor," for a description of the appropriate options.

3.7.3 Compatibility Between Floating-Point Options

Each time you compile a source file, you can specify a floating-point option. When you link more than one source file together to produce an executable program file, you are responsible for ensuring that floating-point operations are handled in a consistent way and that the environment is set up properly to allow the linker to find the required libraries. See Chapter 4, "Linking," for a detailed discussion of linking.

Note

If you are building libraries of C routines that contain floating-point operations, the /FPc (default) floating-point option is recommended for all compilations. The /FPc option offers the greatest amount of flexibility.

Whenever a file is compiled using the /FPi or /FPi87 option, in-line instructions are generated. In the case of the /FPi87 option, the library file 87.LIB and either SLIBFP.LIB, MLIBFP.LIB, or LLIBFP.LIB, depending on the memory model, must be present at link time, and an

8087/80287 coprocessor must be present at run time. For /FPi, the emulator library (EM.LIB) plus SLIBFP.LIB, MLIBFP.LIB, or LLIBFP.LIB must be present at link time, and either the emulator or an 8087/80287 must be present at run time. As long as these requirements are satisfied, object files produced using the /FPi and /FPi87 options can be linked together without compatibility problems. Such object files can also be linked with object files produced using /FPa, /FPc, or /FPc87.

Whenever a file is compiled with the /FPa, /FPc, or /FPc87 options, floating-point function calls are generated. Each option places the name of the appropriate library file or files in the object file. However, when linking several such object files together, you must be aware of the process used to resolve the function calls.

Since floating-point calls to the emulator, the alternate math library, and 8087/80287 coprocessor instructions are interchangeable, only one library is used at link time to resolve the calls. In other words, you must choose one of these libraries per program; the same program cannot make calls to more than one library.

You can control which library is used in one of two ways:

1. At link time, as the *first* name in the list of object files to be linked, give an object file that contains the name of the desired library. For example, if you want to use the alternate math library, give the name of an object file compiled using the /FPa option. All floating-point calls will refer to the alternate math library.
2. At link time, give the /NOD (no default library search) option and then give the name of the floating-point library file or files you want to use in the "Libraries" field. This library overrides the names in the object files, and all floating-point calls will refer to the named library. Since the /NOD option causes all default libraries to be ignored, you must also specify the name of the standard C library (SLIBC.LIB, MLIBC.LIB, or LLIBC.LIB). Always give the names of the floating-point libraries *before* the name of the standard C library in the "Libraries" field.

3.8 Using 80186, 80188, or 80286 Processors

Options

```
/G0
/G1
/G2
```

If you have an 80186, 80188, or 80286 processor, you can use the /G1 or /G2 option to enable the instruction set for your processor. Use /G1 for 80186 and 80188 processors; use /G2 for an 80286. Although it is usually advantageous to enable the appropriate instruction set, you are not required to do so. If you have an 80286 processor, for example, but you want your code to be able to run on an 8086, you should not use the 80286 instruction set.

The /G0 option enables the instruction set for the 8086/8088 processor. You do not have to specify this option explicitly since the 8086/8088 instruction set is used by default.

3.9 Understanding Error Messages

The C compiler generates a broad range of error and warning messages to help you locate errors and potential problems in programs. The following sections describe the form and meaning of the compiler error messages and warning messages you can encounter while using the MSC command. For a list of actual error messages, see Appendix E, "Error Messages."

Error messages produced by the compiler are sent to the standard output, which is usually your console. You can redirect the messages to a file or printer by using an MS-DOS redirection symbol, ">" or ">>". This is especially useful in batch file processing. For example, the following command redirects error messages to the printer device (designated by PRN).

```
MSC ALPHA.C; > PRN
```

Note that only output that ordinarily goes to the console screen is redirected. The object file is given the name ALPHA.OBJ and is created in the current working directory.

3.9.1 C Compiler Messages

The C compiler displays messages about syntactic and semantic errors, such as misplaced punctuation, illegal use of operators, and undeclared variables, in a source file. It also displays warning messages about statements containing potential problems caused by data conversions or the mismatch of types. If you give invalid or incompatible command line options, the compiler will notify you of the error.

The error messages produced by the C compiler fall into five categories: warning messages, fatal error messages, compilation error messages, command line messages, and compiler internal error messages.

Warning messages are informational only; they do not prevent compilation and linking. These messages alert you to potential problems such as type mismatches, data conversions, redeclarations, and overflow conditions. The conditions described by warning messages are not necessarily illegal or undesirable, but you should examine the messages carefully to verify that your program produces these conditions intentionally. Otherwise, your program may not operate as you expect. You can control the level of warnings generated by the compiler by using the /W option, as described in Section 3.9.2.

Fatal error messages indicate a severe problem, one that prevents the compiler from processing your program. Fatal errors can be caused by problems such as insufficient disk space or malformed preprocessor commands. After printing out a message about the fatal error, the compiler terminates without producing an object file or checking for further errors.

Compilation error messages identify actual program errors. No object file is produced for a source file that has such errors. When the compiler encounters a nonfatal program error, it attempts to recover from the error. If possible, the compiler continues to process the source file and produce error messages. If errors are too numerous or too severe, the compiler terminates processing.

Command line messages give you information about invalid or inconsistent command line options. If possible, the compiler continues operation, printing a warning message to indicate which command line options are in effect and which are disregarded. In some cases, command line errors are fatal, and the compiler terminates processing.

Compiler internal error messages indicate an error on the part of the compiler rather than your program. See Section E.3, "Compiler Error Messages," in Appendix E, "Error Messages," for instructions on how to notify Microsoft about internal compiler errors.

Error messages of all types have the same basic form:

filename (*linenumber*) : *message*

where *filename* is the name of the source file being compiled, *linenumber* identifies the line of the file containing the error, and *message* is a self-explanatory description of the error or warning. For warning messages and fatal errors, the word "warning" or "fatal" appears at the beginning of the message, followed by a colon.

The messages for each category are listed in alphabetical order and described in more detail in Appendix E, "Error Messages."

3.9.2 Setting the Warning Level

Option

/W number
/w

You can set the level of warning messages produced by the compiler by using the /W (for "warning") option. This option directs the compiler to display messages about statements that may not be compiled as the programmer intends. Warnings indicate potential problems rather than actual errors.

To use the /W option, choose one of the warning levels described in Table 3.1 and specify the corresponding *number* after the option. The /w option provides a shorter way to say /W 0 and has the same effect.

Table 3.1
Warning Levels

Level	Warning
0	Suppresses all warning messages. Only messages about actual syntactic or semantic errors are displayed.
1	Warns about potentially missing statements, unsafe conversions, and other structural problems. Also, warns about overt type mismatches.
2	Warns about all type mismatches (strong typing).
3	Warns on all automatic data conversions.

The default is level 1, so you do not need to give the /W option when you want level 1.

The higher option levels are especially useful in the earlier stages of program development when messages about potential problems are most helpful. The lower levels are best for compiling programs whose questionable statements are intentionally designed.

Example

```
1.  MSC /W 3 MAIN.C;

2.  MSC /w MAIN.C;
```

The first command directs the compiler to perform the highest level of checking, and produces the greatest number of warning messages. The second command causes MAIN.C to be compiled at the lowest level of checking, with no warning messages. Note that the /w option has the same effect as /W 0.

3.9.3 Compiler Exit Codes

The MSC control program returns an exit code of 0, 2, or 4 to indicate the status of the compilation. The exit code is useful with the MS-DOS batch command IF ERRORLEVEL; it allows you to test for the success or failure of the compilation before proceeding with other tasks in the batch file. The exit codes listed in the first column below have the meanings described in the second column.

0	Successful compilation. Notice that compilation can be successful even if warning messages are produced.
2	Unsuccessful due to program errors.
4	Unsuccessful due to system-level errors (such as insufficient disk space) or compiler internal errors. Compiler internal errors indicate an error on the part of the compiler rather that your program.

See Appendix E, "Error Messages," for details about specific error messages.

3.10 Preparing for Debugging

Options

`/Zd`
`/Od`

The /Zd option produces an object file containing line number records that correspond to the line numbers of the source file. The /Zd option is useful when you want to pass an object file to a symbolic debugger. The debugger can use the line numbers to refer to program locations. (See Section F.5 of Appendix F, "Working with Microsoft Products," for information on using SYMDEB, the symbolic debugger for Microsoft Macro Assembler, with C programs.)

The /Od option tells the compiler *not* to perform optimization. Without the /Od option, the default is to optimize. You may want to use this option when you plan to use a symbolic debugger with your object file. Optimization can involve rearrangement of instructions. If you optimize before debugging, it may be difficult to recognize and correct your code.

Other optimization options are discussed in Section 3.11, "Optimizing."

Example

```
MSC TEST.C, /Zd /Od, TEST.COD;
```

This command produces an object file named TEST.OBJ that contains line numbers corresponding to the line numbers of TEST.C. A listing file, TEST.COD, is also created. No optimization is performed.

3.11 Optimizing

Option

/Ostring

The optimizing procedures available with the Microsoft C Compiler can reduce the storage space and execution time required for a compiled program by eliminating unnecessary instructions and rearranging code. The compiler performs some optimization by default. You can use the /O (for "optimize") options to exercise greater control over the optimizations performed. Some additional advanced optimizing procedures are discussed in Section 7.8 of Chapter 7, "Advanced Topics."

The *string* after the /O option lets you influence how the compiler performs optimization. The string is formed from the following characters.

Character	Optimizing Procedure
s	Favor code size during optimization
t	Favor execution time during optimization
d	Disable optimization
a	Relax alias checking

The letters can appear in any order: /Oat and /Ota have the same effect. The letter "x" is also available with the /O option to perform maximum optimization, as discussed in Section 7.8.2 of Chapter 7, "Advanced Topics."

When you do not give an /O option to the MSC command, it automatically uses /Os, meaning that code size is favored in the optimization. Wherever the compiler has a choice between producing smaller (but perhaps less efficient) and larger (but perhaps more efficient) code, the compiler chooses to generate smaller code. To cause the compiler to favor execution time instead (generating more efficient but perhaps larger code), use the /Ot option.

The /Od option turns off optimization. This option is useful in the early stages of program development to avoid optimizing code that will be changed. Because optimization may involve rearrangement of instructions, you may also want to specify the /Od option when you use a debugger with your program or when you want to examine an object file listing. If you optimize before debugging, it can be difficult to recognize and correct your code.

The "a" option can be used with either the "s" or the "t" option to relax alias checking. The compiler performs alias checking to make sure that it does not eliminate instructions incorrectly when you refer to the same memory location by more than one name. You should include the "a" option only when you are sure that your program does not use aliases.

For example, consider the following code fragment.

```
int count, *pc;
pc = &count;
count = 0;
   .
   .
   .
(*pc)++;
   .
   .
   .
count = 0;
```

The reference to *count* through a pointer, (**pc*), is known as an "alias" for *count* because it provides another way to access the same memory location. When the compiler performs alias checking, it detects the indirect reference to *count* through *pc* and does *not* eliminate the second instruction that assigns zero to *count*.

When you use the "a" option, you are telling the compiler that your program does not use aliases. Therefore, the compiler does not check for indirect references, such as the reference to *count* through a pointer. It would be an error to use the "a" option with the above example. The

compiler would see only that the same value, 0, is assigned to *count* twice, without any intervening assignments that change its value. The second assignment would be considered redundant and would be eliminated in the optimization stage, possibly causing the program to give incorrect results.

Example

```
MSC FILE.C /Ota;
```

This command tells the compiler to relax alias checking and to optimize for faster execution time when it compiles FILE.C.

3.12 Compiling Large Programs

If you are compiling a program or file with more than 64K (kilobytes) of data or with more than 64K of code you should use one of the memory models described in Section 3.13, "Working with Memory Models." (A kilobyte is 1,024 bytes.) You can use map files to determine data and code sizes for each individual program file.

The compiler uses a small memory model by default. The small memory model allocates one segment each, up to 64K in size, for the code and data of your program. (The code segment of a program may also be referred to as the "text" segment.) MSC produces an error message, like the one listed below, if an individual file exceeds these limits.

```
Illegal allocation of segment-type segment > 64K
```

The *segment-type* will specify either "data" or "code," depending on which limit was exceeded.

Even if no individual file exceeds the small model restrictions, you may exceed the 64K limit when you link several compiled files together to form a large program. If this occurs you must recompile the files using a larger memory model. Using a medium memory model allows you to create programs with more than 64K of code (the 64K-restriction on data still applies, however). In large model programs, code and data can both exceed 64K (although no single data item can be larger than 64K).

If your program exceeds the 64K limit on data but has less than 64K of code, you may want to use the far keyword for one or more data items. See Section 7.11.1 of Chapter 7, "Advanced Topics," for a discussion of this option.

Even in medium and large model programs you cannot exceed the limit of 64K of code per program file compiled. The total code size for the program may be greater than 64K, but each individual program file (or "compiland") must contain less than 64K of code.

3.13 Working with Memory Models

Options

/Aletter
/Astring

The MSC command lets you create programs of a variety of sizes and purposes using the /A options. The /A option has two forms. In the first form, you give a single capital letter after /A to select one of the three standard memory models (small, medium, and large) defined by MSC. The compiler uses the small model by default. If your program is too large to compile with the default small model, use a medium or large model, as described below.

In the second form, you supply a string of three lowercase letters that tells the compiler the attributes of the memory model you want to use. The second form is more flexible but it requires a thorough understanding of the memory model attributes: code pointer size, data pointer size, and the stack and data segment setup.

The remainder of this chapter discusses the first form of the memory model option (*/Aletter*). For a discussion of the second form of the memory model option (*/Astring*) and the **near** and **far** keywords, see Section 7.11, "Mixed Model Programming," in Chapter 7, "Advanced Topics."

The memory model options allow you to set up memory in the optimal way for your program. They also determine how the system loads the program for execution.

C programs in memory consist of the actual machine code created from the program's source statements and the data storage created for the program's variables. The data storage, in most cases, also contains the stack used by the program for temporary storage during execution.

The MS-DOS system loads the code and data into "segments" in physical memory. Each segment is up to 64K bytes long. Separate segments are always allocated for the program code and data, so the minimum number of segments allocated for a program is two. Depending on its size (and the use of **near** and **far**, as discussed in Section 7.11.1 of Chapter 7, "Advanced Topics"), a program may require more segments for its code or data. Note that the program size includes any data and code required for library routines.

Three commonly used memory models are defined for you by the MSC command: the small model, the medium model, and the large model. Library support is provided for each of these standard models. Each model defines a different type of program structure and storage.

Small model programs are typically C programs that are short or have a limited purpose. Code and data for these programs each occupy one segment, so they are limited to 64K bytes each (128K bytes maximum total).

Medium model programs are typically C programs that have a large number of program statements but a relatively small amount of data. Program code can occupy any amount of space and is given as many segments as needed. However, no single module (program file) can exceed 64K bytes of code. Program data must not exceed 64K bytes total.

Large model programs are typically very large C programs that use a large amount of data storage during normal processing. Program code can occupy any amount of space, as long as no single module exceeds 64K bytes. Program data may have any size, except that the program must not contain any single data item exceeding 64K bytes.

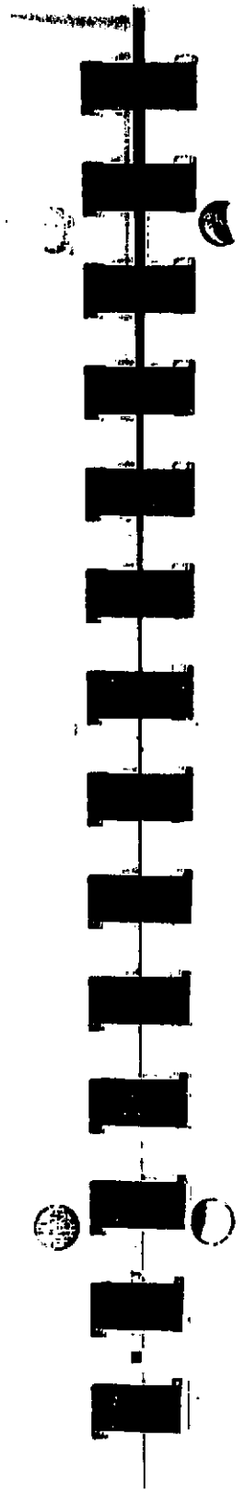
The limitation on data item size in the large model program allows the compiler to perform address arithmetic on just 16 bits (the offset portion) of the address to refer to individual elements or members of the item. This is much more efficient than using a full 32-bit address and is possible because all elements or members of an item are known to reside in the same segment.

When you choose one of these memory models, the compiler operates with certain assumptions about the addresses of code and data for your program. In a small model program, all code is stored in a single segment and all data are stored in a single segment. Because the segment addresses are constant for all code items and all data items, the segment address is not required each time an item is addressed. Instead, any items in the program can be addressed with just an offset from the appropriate segment address. Only 16 bits are required to store an offset from an address, as opposed to 32 bits for a full segmented address. Thus, the compiler generates 16-bit, or "near", pointers for use in small model programs. This is the smallest and fastest option.

A medium model program uses multiple segments for code and a single segment for data. The address of a function, for example, in a medium model program must include the address of the appropriate code segment plus the offset of the beginning of the function from the base of that segment. Full 32-bit, or "far" pointers, are generated by the compiler to access code items in a medium model program. However, just an offset is sufficient for data items, since all data reside in one segment. Data items are accessed with "near" pointers in a medium model program. The medium model provides a useful trade-off of speed and space, since most programs refer more frequently to data items than to code.

A large model program requires the compiler to produce "far" pointers for both code and data items, since multiple segments are allotted for both code and data. Although this is the slowest option, the large model is useful because it can accommodate very large programs.

To provide additional flexibility within the standard memory models, the Microsoft C Compiler allows you to override the default addressing conventions for individual program items by using the special `near` and `far` keywords. These keywords let you access an item with either a "near" or a "far" pointer. This is particularly useful when you have a very large or infrequently used data item that you want to access from a small or medium model program.



3.13.1 Creating Small Model Programs

Option

`/AS`

The small model option tells the compiler to create a program that occupies two segments, one each for code and data, when loaded into physical memory.

The compiler creates small model programs by default when you do not otherwise specify a program model. Thus, the /AS option is never required.

3.13.2 Creating Medium Model Programs

Option

`/AM`

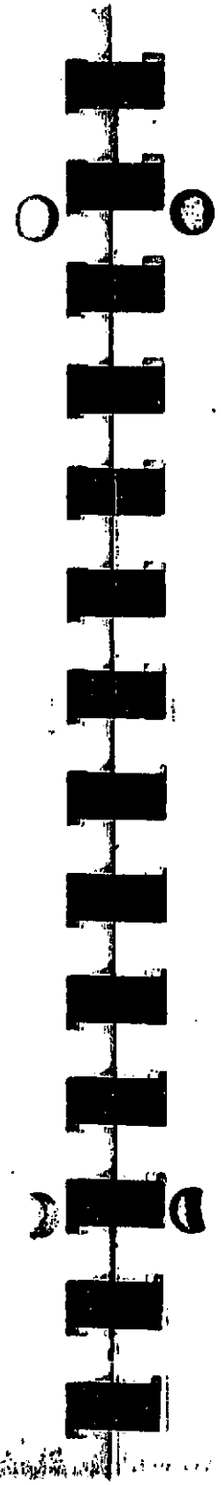
The medium model option creates one segment for the data of the program and one code segment per file compiled.

3.13.3 Creating Large Model Programs

Option

`/AL`

The large model option directs the compiler to create multiple segments, as needed, for both instructions and data. However, no single module can exceed 64K bytes of code, and no single data item can exceed 64K bytes.



Chapter 4
Linking

4.1 Introduction

The Microsoft LINK Object Code Linker links object files compiled on 8086/8088 machines and produces an executable run file as output. The format of input to Microsoft LINK is a subset of the Intel® object module format standard.

The output file from Microsoft LINK (the run file) is not bound to specific memory addresses. It can, therefore, be loaded and executed by the operating system at any convenient address. Microsoft LINK can produce executable files containing up to 1 megabyte of code and data.

4.2 How the Linker Works

Microsoft LINK performs the following steps to combine object modules and produce a run file.

1. Reads the object modules you submit
2. Searches the given libraries, if necessary, to resolve external references
3. Assigns addresses to segments
4. Assigns addresses to public symbols
5. Reads data in the segments
6. Reads all relocation references in object modules
7. Performs fix-ups
8. Outputs a run file (executable image) and relocation information

Microsoft LINK produces a list file that shows how external references are resolved and prints any error messages.

The "executable image" contains the code and data that make up the executable file. The "relocation information" is a list of references, relative to the start of the program, each of which changes when the executable image is loaded into memory and an actual address for the entry point is assigned.

Microsoft LINK uses available memory for the link session. If the files to be linked create an output file that exceeds available memory, Microsoft LINK creates a temporary disk file to serve as memory. The temporary file is created in the current working directory and is named VM.TMP. When this happens, you will see the following message.

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

After this message appears, you must not remove the disk from the given drive (*d*) until the link session ends. If the disk is removed, the operation of Microsoft LINK is unpredictable, and you may see the following message.

```
Unexpected end of file on VM.TMP
```

When this happens you must restart the link session from the beginning.

VM.TMP is a working file only. Microsoft LINK deletes it at the end of the link session.

Warning

Do not give any of your own files the name VM.TMP. If you have a file named VM.TMP on the default drive when Microsoft LINK needs to create a temporary file, LINK deletes the existing VM.TMP file and creates a new one. The contents of the old VM.TMP are lost.

4.3 Linking C Program Files

A number of special considerations should be kept in mind when using Microsoft LINK to process C files. These considerations are discussed below.

4.3.1 The Main Function

When linking C programs, one (and only one) of the object files you submit to Microsoft LINK must have a function named "main". The start-up object module in the standard C library contains a call to the "main" function to begin program execution. If none of the object files you submit contains a "main" function, Microsoft LINK will display an error message informing you that the reference to "main" is unresolved or that the program has no starting address.

4.3.2 Default Libraries and the Library Search Path

Object files created using the Microsoft C Compiler are encoded with the names of the default C libraries for the appropriate memory model. The default C libraries are the standard C library and the floating-point library or libraries selected at compile time. This encoded information enables Microsoft LINK to search for the default library files and link them with your C program.

You do not have to give the names of the default library files when you link. However, you must specify the directory or directories where the library files reside. You can do this by giving directory specifications following the Microsoft LINK "Libraries" prompt, by setting the LIB environment variable, or by combining the two methods.

You can give zero or more directory specifications following the Microsoft LINK "Libraries" prompt. Each directory specification must end with a backslash (\) so Microsoft LINK can recognize the specification as a directory name rather than a library name.

The LIB variable can contain one or more directory specifications. See Section 2.5 of Chapter 2, "Getting Started," for a detailed discussion of environment variables.

To locate library files, Microsoft LINK goes through the following procedure.

1. First, the current working directory is searched.
2. Next, if the library files have not been found, Microsoft LINK searches any directories specified following the Microsoft LINK "Libraries" prompt. The directories are searched in order of their appearance on the line.

3. Finally, if the library files have not been found, Microsoft LINK searches the libraries specified by the LIB environment variable. The directories are searched in order until the given libraries are found.

Note that you can separate the library files and store them in different directories, since Microsoft LINK searches as many of the specified directories as necessary to find the files.

If you want to link with additional libraries, give the library names following the "Libraries" prompt. Microsoft LINK uses the same procedure to search for additional libraries as it does for the default libraries. However, if you give a library name that includes a pathname, Microsoft LINK searches just that pathname for the library; no other directory specifications apply.

4.3.3 Changing the Default Libraries

If you use the /FPa, /FPc87, or /FPc (the default) option when you compile, you are allowed to switch to a different floating-point library at link time. You can do this by giving the name of the library or libraries you want to use following the "Libraries" prompt. See Section 7.7.1 of Chapter 7, "Advanced Topics," for details.

If you do not want to use the standard C library (SLIBC.LIB, MLIBC.LIB, or LLIBC.LIB), you must give the /NOD (for "no default library") option when you link. This option tells Microsoft LINK to ignore the encoded information in the C object files. This option should be used with caution; see the discussion of the /NOD option in Section 4.5.4 for details.

4.3.4 LINK Options to Avoid

Some of the options available with the Microsoft LINK utility are not suitable for use with Microsoft C programs. They include the /HIGH option, the /NOGROUPASSOCIATION option, and the /DSALLOCATE option. Overlays are permitted with C programs, but the /OVERLAYINTERRUPT option (to change the default interrupt number) should not be used.

These options are documented in this chapter along with the other LINK options because you may need them if you use Microsoft LINK to link files written in other languages. The discussion of each option that is not suitable for C programs includes a warning note to that effect.

Using the /DOSSEG option with C programs is not prohibited, but it is never necessary. The segment order specified by the /DOSSEG option is the default segment order for C programs, so the option has no effect on C programs.

4.4 Running the Linker

Microsoft LINK requires two types of input: a command to start Microsoft LINK and responses to command prompts. Start Microsoft LINK by typing

```
LINK
```

at the MS-DOS command level. Microsoft LINK prompts you for the input it needs by displaying the following four lines, one at a time. Microsoft LINK waits for you to respond to each prompt before printing the next one.

```
Object Modules [.OBJ]:
Run File [.EXE]:
List File [NUL.MAP]:
Libraries [.LIB]:
```

The responses you can make to each prompt are explained below.

Once you understand the Microsoft LINK prompts and operations, you can use the two alternate methods of running Microsoft LINK. These alternate methods are described in detail below. The command line method lets you type all commands, options, and filenames on the line used to start Microsoft LINK. With the response file method, you create a file that contains all the necessary commands, and then tell Microsoft LINK where to find that file.

Both of the alternate methods require that you understand how Microsoft LINK works and what your responses to its prompts mean. It is recommended that you allow Microsoft LINK to prompt you for responses until you are comfortable with its commands and operations.

You can also invoke LINK through the CL command. See Section C.3 of Appendix C, "The CL Command," for a discussion of linking with the CL command.

4.4.1 Filename Conventions

You can use either uppercase, lowercase, or a combination of both for the filenames you give in response to the prompts. For example, the following three filenames are considered equivalent.

```
abcde.fgh
AbCdE.FgH
ABCDE.fgh
```

Microsoft LINK uses the default file extensions ".OBJ", ".EXE", ".MAP", and ".LIB" when you do not supply extensions with your filenames. You can override the default extension for a particular prompt by specifying a different extension. To enter a filename that has no extension, type the name followed by a period. For example, typing "ABC." in response to a prompt tells Microsoft LINK that the given file has *no* extension, while typing just "ABC" tells Microsoft LINK to use the default extension for that prompt.

4.4.2 Object Modules Prompt

At the "Object Modules" prompt, list the names of the object files you want to link. For C programs, one (and only one) of the object files must contain a "main" function to serve as the entry point for the program. You must respond to this prompt. There is no default.

Microsoft LINK automatically supplies the ".OBJ" extension when you give a filename without an extension. If your object file has a different extension, you must give the full name, with the extension, for the file to be found.

Pathnames are allowed with the object filenames. This means that you can give Microsoft LINK the pathname of an object file in another directory or on another disk. If Microsoft LINK cannot find a given object file, it displays a message and waits for you to change disks.

Each object filename must be separated from the next by blank spaces or a plus sign (+). If a plus sign is the last character typed on the line, the "Object Modules" prompt reappears on the next line, allowing you to give more object files.

4.4.3 Run File Prompt

The "Run File" prompt lets you supply a name for the executable program file. You can give any filename you like; however, it is recommended that you use an ".EXE" or ".COM" extension, since MS-DOS will only execute files having those extensions.

You are allowed to skip this prompt by typing a carriage return without giving a name. By default Microsoft LINK gives the executable file the name of the first ".OBJ" file listed at the previous prompt, with an ".EXE" extension replacing the ".OBJ" extension of the object file.

4.4.4 List File Prompt

At the "List File" prompt you can tell Microsoft LINK to create a listing file. A listing file contains the names of all segments in order of their appearance in the load module. By adding the /MAP option (discussed in Section 4.5.2.1) you can also list all external (public) symbols and their addresses.

If you give a filename without an extension, Microsoft LINK provides the ".MAP" extension. The ".MAP" extension is not required, so you can give another extension if you like. Microsoft LINK creates the listing file in the current working directory unless you give a different pathname.

You can skip this prompt by hitting a carriage return without giving a name. The default response is the special filename NUL.MAP, which tells Microsoft LINK *not* to create a listing file.

4.4.5 Libraries Prompt

Following the "Libraries" prompt you can give zero or more entries, separated by blank spaces or a plus sign (+). If the plus sign is the last character typed, the "Libraries" prompt reappears on the next line, allowing you to type in additional entries. Each entry can be either a directory specification or a library name. Directory specifications must end with a backslash (\) so that Microsoft LINK can distinguish the directory names from the library names.

When you give a directory specification or specifications, Microsoft LINK uses the specifications to search for the default libraries and for any other libraries without a pathname on the same line. To locate the default libraries, Microsoft LINK searches in the following order.

1. In the current working directory
2. In the directories listed following the "Libraries" prompt (in the same order in which they are listed)
3. In the libraries specified by the LIB environment variable

When you give a library name, Microsoft LINK searches for the given library and links it with your program. If the library name includes a directory specification, Microsoft LINK searches only that directory for the library. If just a library name is given (no directory specification), Microsoft LINK uses the search path described above to locate the given library file.

You can give any combination of directory specifications and library names. Notice that you are not required to give any entries; in this case your program will be linked only with the default libraries, and Microsoft LINK will search for the default libraries in the current working directory and in the directories specified by the LIB variable.

Microsoft LINK automatically supplies the ".LIB" extension if you omit it from a library filename. If you want to link a library file with a different extension, be sure to specify the extension.

Microsoft LINK searches all libraries in order of their appearance on the line and searches only until the first definition of a symbol is found. The default libraries are searched after libraries given on the command line are searched. The default floating-point library (or libraries) is searched before the standard C library.

If you do not want to link with the default floating-point library, you can give the name of a different floating-point library instead, provided that you compiled with a "calls" option (/FPc, /FPc87, or /FPa). See Section 3.7 of Chapter 3, "Compiling," for a discussion of floating-point options. If you do not want to link with the standard C library, you must use the /NOD option, discussed in Section 4.5.4.

4.4.6 Separating Entries

Use the plus sign (+) or one or more space characters to separate filename entries in a list of object files or libraries.

4.4.7 Extending Lines

To extend a line, type the plus sign (+) as the last character of a line to be continued. (This is valid only for the "Object Files" and "Libraries" prompts.) The prompt will reappear on the next line, and you can add more entries. Do not type the plus sign in the middle of a filename entry; the plus sign may only be used after complete filenames.

Example

```
Object Modules [.OBJ]: FUN TEXT TABLE CARE+<RETURN>
Object Modules [.OBJ]: YOYO+FLIPFLOP+JUNQUE+<RETURN>
Object Modules [.OBJ]: CORSAIR<RETURN>
```

4.4.8 Selecting Default Responses

To select the default response to the current prompt, type a carriage return without giving a filename. The next prompt will appear.

To select default responses to the current prompt and all remaining prompts, use a single semicolon (;) followed immediately by a carriage return. Once the semicolon has been entered, you cannot respond to any of the remaining prompts for that link session. Use this option to save time when the default responses are acceptable. Note, however, that the semicolon character is not allowed with the first prompt, "Object Modules [.OBJ]", because there is no default response for that prompt.

Defaults for the other linker prompts are as follows.

1. The default for the "Run File" prompt is the name of the first object file submitted for the previous prompt, with the ".EXE" extension replacing the ".OBJ" extension.
2. The default for the "List File" prompt is the special filename NUL.MAP, which tells Microsoft LINK *not* to create a listing file.

3. The default for the "Libraries" prompt for C programs is the floating-point library and the standard C library for the appropriate memory model (small model is the default).

4.4.9 Terminating the Link Session

To terminate the link session at any time, use CONTROL-C. If you type an erroneous response, you may have to type CONTROL-C to exit Microsoft LINK before you can restart the program.

4.4.10 Using the Command Line

To invoke the linker on the command line, give your responses to the command prompts discussed in the last section on a single line, following the LINK command. The responses to the prompts must be separated by commas, as shown below.

LINK *object-list*[, *executable-name*] [, *map-name*] [, *library-list*] [*/option* . . .] [;]

Here *object-list* is a list of object modules, separated by plus signs or blanks. The *executable-name* is the name of the file to receive the executable output. The *map-name* names the file to receive the map listing. The *library-list* consists of libraries and directories to be searched, separated by plus signs or blanks.

The */option* argument or arguments shown at the end of the line are optional. You do not have to give any options when you run the linker. When you do specify options, you can put them after any of the responses in the line. The options available with Microsoft LINK are described in Section 4.5, "Controlling the Linker."

You can select the default response for any prompt by omitting the filename or list before the comma.

You can also select default responses by using the semicolon. The semicolon tells Microsoft LINK to use the default responses for all remaining prompts.

Example

```
LINK FUN+TEXT+TABLE+CARE, ,FUNLIST,COBLIB.LIB
```

Microsoft LINK loads and links the object modules FUN.OBJ, TEXT.OBJ, TABLE.OBJ, and CARE.OBJ, searching for unresolved references in the library file COBLIB.LIB. By default, the executable file produced is named FUN.EXE. A list file named FUNLIST.MAP is also produced.

4.4.11 Using a Response File

To operate the linker with a response file, you must set up the response file, and then type

LINK @*filename*

Here *filename* gives the name of the response file, possibly preceded by a directory specification. You can name the response file anything you like. Microsoft LINK does not impose any naming restrictions for the response file.

A response file contains responses to the Microsoft LINK prompts. Options may be appended to any of the responses or given on a separate line or lines. The responses must be in the same order as the Microsoft LINK prompts discussed above. Each new response must start a new line; however, you can extend long responses across more than one line by typing a plus sign (+) as the last character of each incomplete line.

Options and command characters are used in the response file in the same way they are used for responses typed at the keyboard. For example, if you type a semicolon on the line of the response file corresponding to the "Run File" prompt, Microsoft LINK uses the default responses for the executable file and for the remaining prompts.

When you give the Microsoft LINK command with a response file, each Microsoft LINK prompt is displayed on your screen with the corresponding response from your response file. If the response file does not contain answers for all the prompts (in the form of filenames, the semicolon command character, or carriage returns), Microsoft LINK displays the missing prompts and waits for you to enter responses. When you type an acceptable response, Microsoft LINK continues the link session.

Example

```
FUN TEXT TABLE CARE
/PAUSE /MAP
FUNLIST
COBLIB.LIB
```

This response file tells Microsoft LINK to load the four object modules FUN, TEXT, TABLE, and CARE. Two output files are produced, named FUN.EXE and FUNLIST.MAP. The /PAUSE option causes Microsoft LINK to pause before producing the executable file (FUN.EXE). This permits you to swap disks if necessary. See the discussion of the /PAUSE and /MAP options in Section 4.5, "Controlling the Linker," for more on these options.

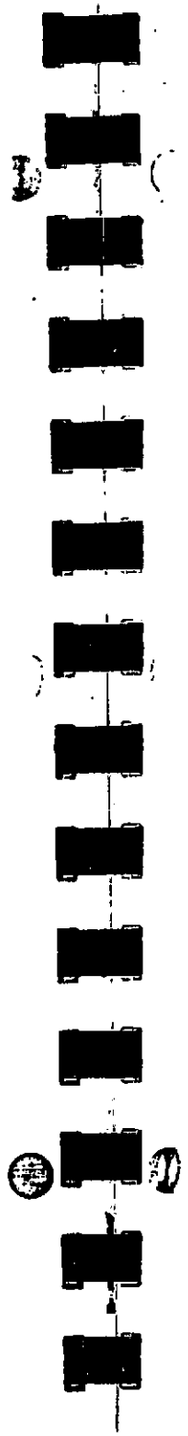
4.5 Controlling the Linker

This section explains how to use linker options to specify and control the tasks performed by the linker. All options begin with the linker option character, the forward slash (/). Options may be placed at the end of any Microsoft LINK response.

When more than one option is given, the options can be grouped at the end of a single response or distributed among several responses. Every option begins with the slash character, even if other options precede it on the same line.

When you use the command line method to invoke Microsoft LINK, options can appear at the end of the line or after individual responses on the line, but before the comma separating each response from the next item. In a response file, options can go after individual responses or by themselves on one of the prompt lines.

The options are named according to their function, with the result that some names are quite long. You can abbreviate the options to save on space and typing. Be sure that your abbreviation is unique so that the linker can determine which option you want. For example, several options begin with the letters "NO" so abbreviations for those options must be longer than "NO" to be unique.



Abbreviations must be sequential from the first letter of the option through the last letter typed. No gaps or transpositions are allowed. To illustrate this rule, the following list shows legal and illegal abbreviations for the /NOIGNORECASE option.

Legal	Illegal
/NOI	/NO
/NOIG	/NIG

Some linker options take numerical arguments. A numerical argument can be any of the following.

- A decimal number from 0 to 65,535.
- An octal number from 0 to 0177777. A number is interpreted as octal if it starts with a zero. For example, the number "10" is a decimal number, but the number "010" is an octal number, equivalent to 8 in decimal.
- A hexadecimal number from 0 to 0xFFFF. A number is interpreted as hexadecimal if it starts with "0x". For example, "0x10" is a hexadecimal number, equivalent to 16 in decimal.

4.5.1 Pausing During Linking

Option

/PAUSE

Unless you instruct it otherwise, Microsoft LINK performs the linking session from beginning to end without stopping. The /PAUSE option tells Microsoft LINK to pause in the link session before writing the executable file to disk. This option allows you to swap disks before Microsoft LINK outputs the executable (".EXE") file.

If the /PAUSE option is given, Microsoft LINK displays the following message before creating the run file.

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

The *letter* corresponds to the current drive. Microsoft LINK resumes processing when you press either the ENTER or the RETURN key.

Note

Do not remove the disk that will receive the list file or the disk used for the VM.TMP file, if one has been created.

4.5.2 Producing a Listing File

You can tell Microsoft LINK to produce a listing file by responding to the "List File" prompt. A listing file contains a list of segments in order of their appearance within the load module. The list might look like the following.

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

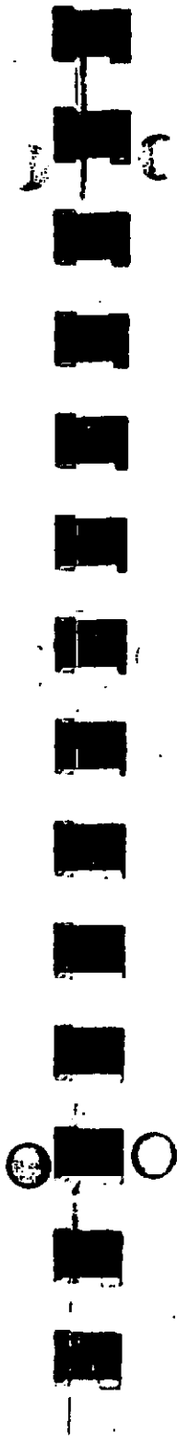
The information in the "Start" and "Stop" columns shows the 20-bit address (in hexadecimal) of each segment relative to the beginning of the load module. The load module begins at location zero. The "Length" column gives the length of the segment in bytes. The "Name" column gives the name of the segment, and the "Class" column gives information about the segment type. For example, a CODE segment contains program code and a DATA segment contains data.

The starting address and name of each group is listed at the end of the file. The group listing might look like the following.

```
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 bottom of the listing file, Microsoft LINK gives you the address of the program entry point.



4.5.2.1 Listing Public Symbols

Option

/MAP

You can list all public (global) symbols defined in an object file or files by using the /MAP option. The /MAP option forces a listing file to be created if it appears before the "List File" prompt. By default, the file is given the same basename as the executable file plus the extension ".MAP". You can override the default name by responding to the "List File" prompt. However, if the /MAP option appears *after* the "List File" prompt, the option takes effect only if you explicitly created a listing file (by giving a name at the "List File" prompt).

Without the /MAP option the listing file gives only the starting and ending addresses, length, and name of each segment. The /MAP option appends two lists of global symbols to the listing file. The first list is alphabetical by symbol name and the second is sorted by symbol address.

Example

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

Address      Publics by Name
0000:01dB    chkstk
0173:0035    fac
0000:1567    _brk
0000:1696    _chmod
0000_131C    _clearerr
   .
   .
   .

Address      Publics by Value
0000:0035     _chkln
0000:01D2    __fptrap
0000:01DB    __chkstk
0000:023F    _main
0000:025A    __exit
   .
   .
   .
```

The address of the external symbols are in "frame:offset" format, showing the location of the symbol relative to zero (the beginning of the load module).

When you examine a map file, you will notice that the names of globally visible functions and variables begin with an underscore. The Microsoft C compiler automatically prefixes an underscore to all global names to preserve compatibility with XENIX C compilers. If you write assembly language routines to interface with your C program, this naming convention is important; see Section 8.1.6 of Chapter 8, "Interfaces with Other Languages."

In the listing file, you may also see names that begin with more than one underscore. Identifiers with more than one leading underscore are reserved for internal use by the compiler. You should not attempt to use these identifiers in your program. Moreover, you should avoid creating global names that begin with an underscore. Since the compiler automatically

adds another leading underscore, these names end up with two leading underscores, possibly causing conflicts with the names reserved by the compiler.

4.5.2.2 Displaying Line Numbers

Option

```
/LINENUMBERS
```

You can include the line numbers and associated addresses of your source program in the linker listing by using the /LINENUMBERS option. Ordinarily the linker listing does not contain line numbers.

To produce a linker listing with line numbers, you must give Microsoft LINK an object file (or files) with line number information. With the C compiler you can use the /Zd switch to produce line numbers in the object file. If you give Microsoft LINK an object file without line number information, the /LINENUMBERS option has no effect.

The /LINENUMBERS option forces a listing file to be created if it appears before the "List File" prompt. By default, the file is given the same basename as the executable file plus the extension ".MAP". You can override the default name by responding to the "List File" prompt. However, if the /LINENUMBERS option appears *after* the "List File" prompt, the option takes effect only if you explicitly created a listing file (by giving a name at the "List File" prompt).

4.5.3 Ignoring Case

Option

```
/NOIGNORECASE
```

By default, Microsoft LINK treats uppercase and lowercase letters as equivalents. Thus, "ABC", "abc", and "Abc" are considered to be the same name. When you use the /NOIGNORECASE option (usually abbreviated /NOI), the linker distinguishes between uppercase and lowercase letters and considers "ABC", "abc", and "Abc" three separate names.

The C language is case-sensitive: two names are identical only if they have exactly the same letters in the same case. If your C programs rely on this behavior, you should use the /NOIGNORECASE option. Keep in mind, however, that some programs, such as assemblers and other language compilers, may not make case distinctions. If you want to link such programs with your C programs, it is best to give each identifier a unique spelling to avoid conflicts.

4.5.4 Ignoring Default Libraries

Option

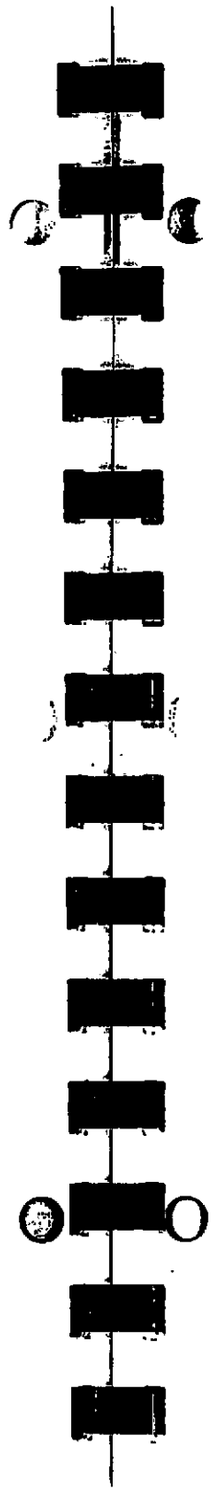
/NODEFAULTLIBRARYSEARCH

The /NODEFAULTLIBRARYSEARCH option (usually abbreviated to /NOD) tells Microsoft LINK *not* to search the default library, if there is one, to resolve external references. With C files this has the effect of telling Microsoft LINK to ignore the information in the object files that gives the names of the standard C library and selected floating-point library.

Most C programs will not work correctly without the standard C library, so if you use the /NOD option you should explicitly specify the name of the standard library, as well as any floating-point libraries needed by the program. If you do not use the standard library, you must provide your own start-up routine, or extract the start-up routine from the standard library and link it with your program. (See Section 2.4.3, "Library Files," in Chapter 2, "Getting Started," for a discussion of the start-up routine.)

When using the /NOD option with C programs, always use the following order to specify libraries.

1. Any libraries other than the standard C library or floating-point libraries
2. The floating-point library or libraries
3. The standard C library



4.5.5 Controlling Stack Size

Option

/STACK:*number*

The /STACK option allows you to specify the size of the stack for your program. The *number* is any positive value (decimal, octal, or hexadecimal) up to 65,536 (decimal). It represents the size, in bytes, of the stack.

All compilers and assemblers should provide information in the object modules that tells the linker how to set up the stack. For C programs, the default stack size is 2K bytes. The default stack size is set by the start-up routine (CRT0.OBJ) in the standard C library.

If your program has a large amount of local data or is heavily recursive, you may get a stack overflow message. In this case you need to increase the size of the stack. On the other hand, if your program uses very little local data, you may achieve some space savings by decreasing the stack size.

If Microsoft LINK cannot find the stack information it needs, it displays the following error message. Since the start-up file provides stack information, this message usually means that the start-up file is not being linked with your program.

```
WARNING: NO STACK SEGMENT
```

Note

The EXEMOD utility (described in Section 7.9.2 of Chapter 7, "Advanced Topics") can also be used to change the default stack size for C program files.

4.5.6 Allocating Paragraph Space

Option

/CPARMAXALLOC : *number*

This option allows you to change the default value of the "cparMaxALLOC" field, which controls the maximum number of paragraphs in memory allocated for your program. A paragraph is defined as the smallest memory unit addressed by a segment register, or 16 bytes.

The maximum number of paragraphs allocated for a program is determined by the value of the "cparMaxALLOC" field at offset 0x0c in the EXE header. See your *Microsoft MS-DOS Programmer's Reference Manual* for details.

By default, the "cparMaxALLOC" field is set to 65,535 (decimal), or 64K minus 1. The /CPARMAXALLOC option lets you set the value to any number between 1 and 65,535 by giving the desired *number* (decimal, octal, or hexadecimal) with the option. This is useful when you know that the efficiency of your program will not be increased by allocating all available memory, or when you want to execute another program from within your program and you need to reserve space for the executed program.

If the value you specify is less than the computed value of "cparMinAlloc" (at offset 0A hexadecimal), the linker uses the value of "cparMinAlloc" instead.

4.5.7 Controlling Segments

Option

/SEGMENTS : *number*

The /SEGMENTS option controls the number of segments the linker allows a program to have. The default is 128. The *number* can be set to any value (decimal, octal, or hexadecimal) in the range 1 to 1,024 (decimal).

For each segment, the linker must allocate some space to keep track of segment information. By using a relatively low segment limit as a default (128), the linker avoids having to allocate a large amount of storage space for all programs.

When you set the segment limit higher than 128, the linker correspondingly allocates more space for segment information. This option allows you to raise the segment limit for programs with a large number of segments. For programs with fewer than 128 segments, you can keep the storage requirements of the linker at the lowest level possible by setting the segment *number* to reflect the actual number of segments in the program.

4.5.8 Using Overlays

You can direct Microsoft LINK to create an overlaid version of your program. This means that parts of your program will only be loaded if and when they are needed, and will share the same space in memory. Your program will be smaller as a result, but will usually run more slowly because of the time needed to read and reread the code into memory.

Provided your modules obey the restrictions described below, all you have to do is specify the overlay structure to the linker. Loading of the overlays is automatic. You specify overlays in the list of modules that you submit to the linker by enclosing them in parentheses. Each parenthetical list represents one overlay. For example, in the following response to the "Object Modules" prompt,

```
Object Modules [.OBJ] a + (b+c) + (e+f) + g + (i)
```

the elements "(b+c)", "(e+f)", and "(i)" are overlays. The remaining modules, and any drawn from the run-time libraries, make up the resident part of your program, or root. Overlays are loaded into the same region of memory, so only one can be resident at a time. Duplicate names in different overlays are not supported, so each module can occur only once in a program.

The linker will replace calls from the root to an overlay and calls from an overlay to another overlay with an interrupt (followed by the module identifier and offset.) The interrupt number is 3F hexadecimal.

4.5.8.1 Restrictions

The name for the overlays is appended to the ".EXE" file, and the name of this file is encoded into the program so the Overlay Manager can access it. If, when the program is initiated, the overlay manager cannot find the ".EXE" file (perhaps you have renamed it or it is not in a directory specified by the PATH environment variable), then the overlay manager will prompt you for a new name.

You can only overlay modules to which control is transferred and returned by a standard 8086 long (32-bit) call/return instruction. With C programs, long calls are the default only in medium and large model programs. See Section 3.13 of Chapter 3, "Compiling," for details on the standard memory models.

You cannot use long jumps (via the *longjmp* library function) or indirect calls (through a function pointer) to pass control to an overlay. When a function is called through a pointer, the called function must be in the same overlay or in the root.

4.5.8.2 Overlay Manager Prompts

Suppose that B: is the default drive. In the following example,

```
Cannot find PAYROLL.EXE
Please enter new program spec:
```

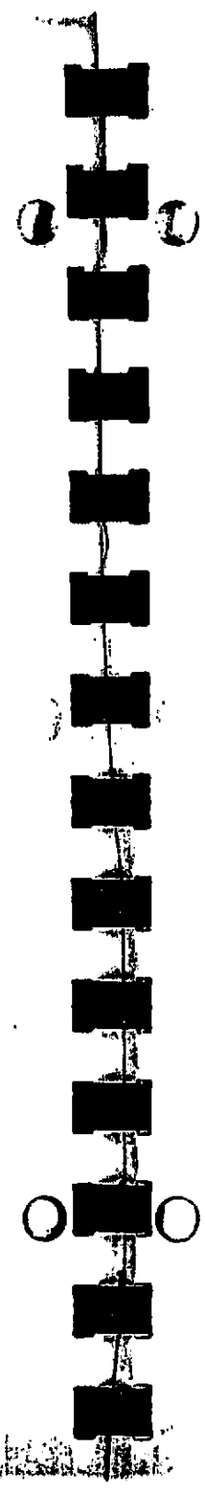
the response EMPLOYEE\DATA\ causes the overlay manager to look for EMPLOYEE\DATA\PAYROLL.EXE on B.

Now, suppose that it becomes necessary to change the disk in Drive B. If the overlay manager needs to swap overlays, it will find that PAYROLL.EXE is no longer on B, and will give the following message.

```
Please put diskette containing B:\EMPLOYEE\DATA\PAYROLL.EXE
in drive B: and strike any key when ready.
```

After the overlay has been read from the disk, the overlay manager will give the following message.

```
Please restore the original diskette.
Strike any key when ready.
```

4.5.9 Setting the Overlay Interrupt

Option

/OVERLAYINTERRUPT:*number*

By default, the interrupt number used for passing control to overlays is 3F hexadecimal. The overlay interrupt option allows the user to select a different interrupt number. The *number* can be a decimal number from 0 to 255, an octal number from 0 to 0377, or a hexadecimal number from 0 to 0xFF. Numbers that conflict with MS-DOS interrupts are not prohibited, but their use is not advised.

Warning

Do not use the /OVERLAYINTERRUPT option with C programs.

4.5.10 Ordering Segments

Option

/DOSSEG

The /DOSSEG switch forces segments to be ordered according to the following rules.

1. All segments with a class name ending in CODE come first.
2. All other segments outside of DGROUP come next.
3. DGROUP segments come last, in the following order.
   a. Any segments of class BEGDATA (this class name is reserved for Microsoft use)
   b. Any segments not of class BEGDATA, BSS, or STACK
   c. Segments of class BSS
   d. Segments of class STACK

See Section 7.13 of Chapter 7, "Advanced Topics," for a discussion of the segment names used by the C compiler.

4.5.11 Controlling Data Loading

Option

/DSALLOCATE

By default, Microsoft LINK loads all data starting at the low end of the data segment. At run time, the DS (data segment) pointer is set to the lowest possible address to allow the entire data segment to be used.

Use the /DSALLOCATE option to tell Microsoft LINK to load all data starting at the high end of the data segment instead. In this case the DS pointer is set at run time to the lowest data segment address that contains program data.

The /DSALLOCATE option is typically used with the /HIGH option, discussed in the next section, to take advantage of unused memory within the data segment. The user can allocate any available memory below the area specifically allocated for DGROUP, using the same DS pointer.

Warning

Do not use the /DSALLOCATE option with C programs.

4.5.12 Controlling Run File Loading

Option

/HIGH

The run file can be placed either as low or as high in memory as possible. Use of the /HIGH option causes Microsoft LINK to place the run file as high as possible in memory. Without the /HIGH option, Microsoft LINK places the run file as low as possible.

Warning

Do not use the /HIGH option with C programs.

4.5.13 Preserving Compatibility

Option

/NOGROUPASSOCIATION

The /NOGROUPASSOCIATION option causes the linker to process a certain class of fix-ups in a manner that is compatible with previous versions of the linker (Versions 2.02 and earlier). It is provided primarily for compatibility with previous versions of other Microsoft language compilers.

Warning

Do not use the /NOGROUPASSOCIATION option with C programs.

Chapter 5
Running Your C Program

5.1 Running a Program

Once you have created an executable file, you can run your program by giving the name of the file without the extension. However, the file must have a ".EXE" or ".COM" extension; otherwise MS-DOS will not execute it.

MS-DOS uses the PATH environment variable to find executable files. You can execute a program from any directory, as long as the executable program file is either in your current working directory or in one of the directories on the PATH.

The *spawn, exec,* and *system* routines provided in the Microsoft C Run-Time Library let you execute other programs and MS-DOS commands from within a program. See the *Microsoft C Run-Time Library Reference* for a description of these routines.

Example

```
MYPROG
```

This example runs the executable file named MYPROG.EXE (or MYPROG.COM).

5.2 Passing Data to a Program

You can access data on the command line or in the environment table at execution time by declaring arguments to the "main" function. Command line data are any data that appear on the same line as the program name when you execute the program. The environment table contains all environment settings in effect at execution time (see Section 2.5, "Setting Up the Environment," in Chapter 2, "Getting Started," for a discussion of environment variables). Since the "main" function is where program execution begins, "main" is the function that takes charge of data pased at execution time.

To pass data to your program on the command line, give one or more arguments after the program name when you execute it. Each argument must be separated from the arguments around it by one or more spaces or tab characters, and may be enclosed in quotation marks (""). If you want to give a single argument that includes spaces or tab characters, you must enclose the argument in quotation marks. For example, the command line

```
TEST 42 "de f" 16
```

executes the program named TEST.EXE (or TEST.COM) and passes three arguments: "42", "de f", and "16".

Under MS-DOS, each command line argument, regardless of its data type, is stored as a null-terminated string in an array of strings. The combined length of all arguments on the command line (including the program name) may not exceed 128 bytes.

To set up your program to receive the command-line data, declare arguments to the "main" function, as shown below. By declaring these variables as arguments to "main", you make them available as local variables in the "main" function.

```
main (argc, argv, envp)

int argc;
char *argv[];
char *envp[];
```

You do not have to declare all three arguments. However, you must declare the arguments in the order shown above. Thus, if you want to use the "envp" arguments, you must declare "argc" and "argv", even if you do not use them.

The command line is passed to the program as the "argv" array of strings. The number of arguments appearing on the command line is passed as the integer variable "argc".

The first argument of any command line is the name of the program to be executed. Thus, the program name is the first string stored in "argv", at "argv[0]". Since a program name must be given in all cases, the integer value of "argc" is always at least one. The first argument given after the program name is stored at "argv[1]", the second is stored at "argv[2]", and so on through the end of the arguments. The total number of arguments, including the program name, is stored in "argc".

Note

Under versions of MS-DOS earlier than 3.0, the program name normally stored in "argv[0]" is not available. References to "argv[0]" yield the empty string. Under MS-DOS Version 3.0 and later, references to "argv[0]" give the program name.

The third argument passed to the "main" function, "envp", is a pointer to the environment table. You can access the value of environment settings through this pointer. However, the *putenv* and *getenv* routines from the C run-time library accomplish the same task, and are easier and safer to use. Use of the *putenv* routine may change the location of the environment table in memory, depending on memory requirements. Thus, the value given to "envp" at the beginning of the program execution may not be valid throughout the program's execution. The *putenv* and *getenv* routines, on the other hand, access the environment table properly, even when its location changes. These routines use the global variable *environ* (described in the *Microsoft C Run-Time Library Reference*), which always points to the correct table location.

Example

```
MYPROG ABC "abc e" 3 8
```

This command line executes the program named MYPROG and passes the four command line arguments to the "main" function. The arguments are stored as null-terminated strings, and the number of arguments is stored in "argc". To access the last argument, for example, you would use an expression like the following.

```
argv[argc - 1]
```

Since the value of "argc" is 5 (counting the program name as an argument), this expression is equivalent to "argv[4]", or the fifth string of the array.

5.3 Expanding Wild Card Arguments

You can use the MS-DOS wild card characters, the question mark (?) and the asterisk (*), to specify filename and pathname arguments on the command line. To prepare for using wild cards, you must link with the special routine SSETARGV.OBJ (MSETARGV.OBJ or LSETARGV.OBJ for medium or large model programs). These object files are included with your compiler software. If you don't link with the appropriate object file, wild characters are treated literally.

The SSETARGV.OBJ expands the wild card characters in the same manner as MS-DOS. (See your MS-DOS documentation if you are unfamiliar with these characters.) Enclosing an argument in quotation marks ("") suppresses the wild card expansion. Within quoted arguments, you can represent quotation marks literally within an argument by preceding the double quote character with a backslash (\).

If no matches are found for the wild card argument, the argument is passed literally. For example, if the argument B:*.C is given, but no files with the extension ".C" are found in the root directory of Drive B, the argument is passed as the string "B:*.C".

If you frequently use wild card expansion, you may want to place the wild card routines (SSETARGV.OBJ, MSETARGV.OBJ, and LSETARGV.OBJ) in the appropriate standard C libraries (SLIBC.LIB, MLIBC.LIB, and LLIBC.LIB). That way the routine will be linked with your program automatically. To do this, use the LIB utility (described in Chapter 6, "Managing Libraries") to extract the module named *stdargv* from the library (the module name is the same in all three libraries) and insert SSETARGV, MSETARGV, or LSETARGV, as appropriate. When you replace *stdargv* with the appropriate SETARGV routine, wild card expansions will always be performed on command line arguments.

Example

```
LINK BETA SSETARGV;
BETA *.INC "WHY?" \"HELLO\"
```

In this example, SSETARGV.OBJ is linked with BETA.OBJ, producing the executable file BETA.EXE. When BETA.EXE is executed, the wild card character "*" is expanded, causing all filenames with the extension ".INC" in the current working directory to be passed as arguments to the

BETA program. The second command line argument, WHY?, is enclosed in quotation marks (""), so expansion of the wild card character "?" is suppressed and the argument WHY? is passed literally. In the third argument, the backslashes cause the quotations to be represented literally so the argument "HELLO" is passed.

5.4 Suppressing Command Line Processing

If your program does not take command line arguments, you can achieve a small space savings by suppressing use of the library routine that performs command line processing. This routine is called *_setargv*. To suppress its use, define a routine that does nothing in the same file that contains the "main" function, and name it *_setargv*. The call to *_setargv* will be satisfied by your definition of *_setargv*, and the library version will not be loaded.

Similarly, if you never access the environment table through the "envp" argument, you can provide your own empty routine to be used in place of *_setenvp*, the environment processing routine. In the same file that contains the "main" function, define a routine that does nothing and name it *_setenvp*.

If your program makes calls to the *spawn* or *exec* routines in the C runtime library, you may not want to suppress the environment processing routine. This routine is used to pass an environment from the parent process to the child process.

Example

```
_setargv()
{
}

_setenvp()
{
}
```

The above example shows how to define the *_setargv* and *_setenvp* functions to suppress command line and environment processing. It is recommended that you place these definitions in the file containing the "main" function.

5.5 Suppressing Null Pointer Checks

When you execute your C program, a special error-checking routine is automatically invoked to determine whether the contents of the NULL segment have changed and to display the following error message if they have.

```
Null pointer assignment
```

The NULL segment is a special location in low memory that is not normally used. If the contents of the NULL segment change during a program's execution, it means that the program has written to this area, usually by an inadvertent assignment through a null pointer. Notice that your program can contain null pointers without generating this message; the message appears only when you write to a memory location through the null pointer.

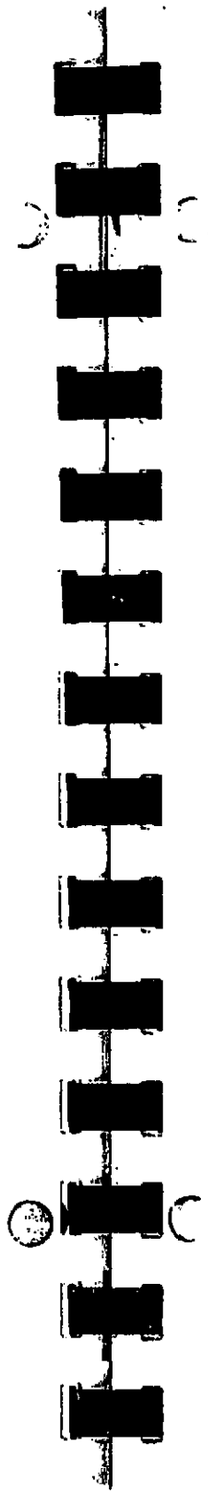
This error does not cause your program to terminate; the error is detected and the error message is printed following the normal termination of the program.

Important

The "null pointer assignment" message reflects a potentially serious error in your program. Although a program that produces this error may appear to operate correctly, it is likely to cause problems in the future and may fail to run in a different operating environment.

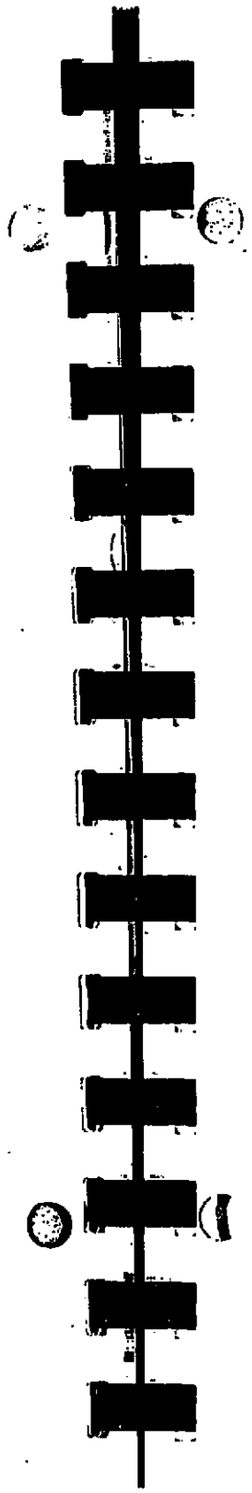
The library routine that performs the null pointer check is named *_nullcheck*. You can suppress the null pointer check for a particular program by defining your own routine named *_nullcheck* that does nothing. The call to *_nullcheck* will be satisfied by your definition of *_nullcheck*, and the library version will not be loaded. It is recommended that you place the *_nullcheck* definition in the file containing the "main" function.

To suppress the null pointer check for all programs, you can replace the corresponding error-checking routine in the standard C library. The routine is stored in a module called *chksum* in all three libraries (SLIBC.LIB, MLIBC.LIB, and LLIBC.LIB). Do not remove the routine entirely or there will be an unresolved reference in your program. Instead, use the LIB



utility (described in Chapter 6, "Managing Libraries") to replace the *chksum* module with a module containing the empty definition of *_nullcheck*. This replacement will satisfy the call to *_nullcheck* and null pointer checking will not be performed.

Chapter 6
Managing Libraries



6.1 Introduction

The Microsoft LIB Library Manager is a utility designed to help you create, organize, and maintain run-time libraries. Run-time libraries are collections of compiled or assembled functions that provide a common set of useful routines. Any program can call a run-time routine, exactly as if the function were included in the program. The program is linked with the run-time library file and the call to the run-time routine is resolved by finding the routine in the library file.

Run-time libraries are created by combining separately compiled object files into one library file. Library files are usually identified by their ".LIB" extension, although other extensions are allowed.

In addition to accepting MS-DOS object files and library files, Microsoft LIB accepts 286 XENIX archives and Intel-style libraries. You can add the contents of a 286 XENIX archive or an Intel-style library to an MS-DOS library by using the append operator (+).

Once an object file is incorporated into a library, it becomes an object "module." LIB makes a distinction between object files and object modules: an object "file" exists as an independent file while an object "module" is part of a larger library file. An object file can have a full pathname, including a drive designation, directory pathname, and filename extension (usually ".OBJ"). Object modules have just a name. For example, "B:\RUN\SORT.OBJ" is an object filename, while "SORT" is the corresponding object module name.

Using LIB, you can create a new library file, add object files to an existing library, delete library modules, replace library modules, and create object files from library modules. LIB also lets you combine the contents of two libraries into one library file.

The command syntax is straightforward, and LIB prompts you for responses. Once you have learned how LIB works and what its prompts mean, you can use one of the alternate methods of invoking LIB, described in the sections entitled "Using the Command Line" and "Using a Response File" later in this chapter. The alternative methods let you give LIB commands without waiting for the LIB prompts.

6.2 Overview of LIB Operation

You can perform a number of library management functions with Microsoft LIB. LIB can

- Create a library file
- Delete modules
- Extract a module and place it in a separate object file
- Extract a module and delete it
- Append an object file as a module of a library, or append the contents of a library
- Replace a module in the library file with a new module
- Produce a listing of all public symbols in the library modules

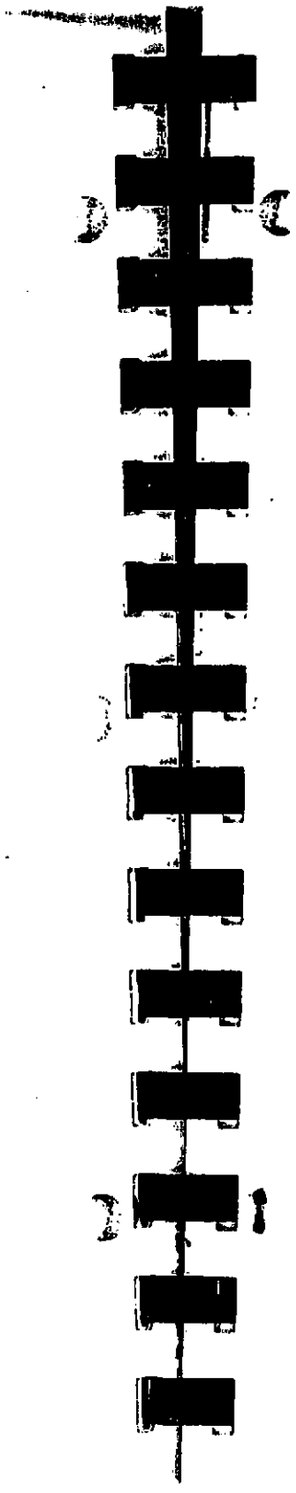
For each library session, LIB first reads and interprets the user's commands. It determines whether a new library is being created or an existing library is being examined or modified.

Deletion and extraction commands (if any) are the first commands processed. LIB does not actually delete modules from the existing file. Instead, it marks the selected modules for deletion, creates a new library file, and copies only the modules *not* marked for deletion into the new library file.

Next, LIB processes any addition commands. Like deletions, additions are not performed on the original library file. Instead, the additional modules are appended to the new library file. (If there were no deletion or extraction commands, a new library file is created in the addition stage by copying the original library file.)

As LIB carries out these commands, it reads the object modules in the library, checks them for validity, and gathers the information necessary to build a library index and a listing file. The library index is used by the linker to search the library.

The listing file contains a list of all public symbols in the index and the names of the modules in which they are defined. LIB produces the listing file only if you ask for it during the library session.



LIB never makes changes to the original library; it copies the library and makes changes to the copy. Thus, when you terminate LIB for any reason, you do not lose your original file. It also means that when you run LIB, enough space must be available on your disk for both the original library file and the copy.

When you modify a library file, LIB gives you the option of specifying a different name for the file containing the modifications. If you use this option, the modified library is stored under the name you give, and the original, unmodified version is preserved under its own name. If you choose not to give a new name, LIB gives the modified file the original library name, but keeps a backup copy of the original library file. This copy has the extension ".BAK" instead of ".LIB".

Example

```
LIB LANG-HEAP+HEAP;
```

This command first deletes the library module HEAP from the library file LANG.LIB, then adds the file HEAP.OBJ as the last module in the library. The same command could be given as

```
LIB LANG+HEAP-HEAP;
```

The effect is the same because delete operations are always carried out before append operations, regardless of the order of the operations in the command line. This order of execution prevents confusion in LIB when a new version of a module replaces an old version in the library file.

After the library is modified, the modified file is written back to LANG.LIB. A copy of the original LANG.LIB is stored in LANG.BAK.

6.3 Running LIB

LIB requires two types of input: a command to start LIB and responses to command prompts. Start LIB at the MS-DOS command level by typing

```
LIB
```

LIB prompts you for the input it needs by displaying the following four

messages, one at a time. LIB waits for you to respond to each prompt, then prints the next prompt.

```
Library name:
Operations:
List file:
Output library:
```

The responses you can make to each prompt are explained in the following four sections.

Once you understand the LIB prompts and operations, you may want to use one of the two alternate methods of running LIB. The command line method lets you type all commands, options, and filenames on the line used to start LIB. With the response file method, you create a file that contains all the necessary commands, then tell LIB where to find that file.

Both of the alternate methods require that you understand how LIB works and what your responses to its prompts mean. For this reason it is recommended that you allow LIB to prompt you for responses until you are comfortable with its commands and operations.

6.3.1 Library Name Prompt

At the "Library name" prompt, give the name of the library file you want. Usually library files are named with the ".LIB" extension. You can omit the ".LIB" extension when you give the library filename since LIB assumes that the filename extension is ".LIB".

If your library file does not have the ".LIB" extension, be sure to include the extension when you give the library filename. Otherwise, LIB cannot find the file.

Pathnames are allowed with the library filename, so you can give LIB the pathname of a library file in another directory or on another disk.

Because LIB manages only one library file at a time, only one filename is allowed in response to this prompt. There is no default response, so LIB produces an error message if you do not give a filename.

If you give the name of a library file that does not exist, LIB displays the prompt:

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


Type "y" to create the library file. If you answer "n", LIB returns control to the MS-DOS level.

If you type a library filename and follow it immediately with a semicolon (;), LIB performs only a consistency check on the given library. A consistency check tells you whether all the modules in the library are in usable form. LIB prints a message only if it finds an invalid object module; no message appears if all modules are intact.

You can also set the library page size following this prompt. For details, see Section 6.4.10 "Setting the Library Page Size."

6.3.2 Operations Prompt

Following the "Operations" prompt, you can type one of the command symbols for manipulating modules (+, -, -+, *, -*) followed immediately by the module name or the object filename. You can specify more than one operation following this prompt, in any order. The default for the "Operations" prompt is no changes.

When you have a large number of modules or files to manipulate (more than can be typed on one line), type an ampersand (&) as the last symbol on the line, immediately before the carriage return. The ampersand must follow a filename; you cannot give an operator as the last character on a line to be continued. The ampersand causes LIB to repeat the "Operations" prompt, allowing you to specify more operations and names.

The following list describes command symbols and their meanings and uses.

Command Symbol	Meaning and Use
+	The plus sign makes an object file the last module in the library file. Give the name of the object file immediately after the plus sign. You can use pathnames for the object file. LIB automatically supplies the ".OBJ" extension, so you can omit the extension from the object filename. You can also use the plus sign to combine two libraries. When you give a library name after the plus sign, a copy of the contents of the given library is added to the library file being modified. You must include the ".LIB"

extension when you give a library filename. Otherwise, LIB uses the default ".OBJ" extension when it looks for the file.

`-` The minus sign deletes a module from the library file. Give the name of the module to be deleted immediately after the minus sign. A module name has no pathname and no extension.

`-+` Type a minus sign followed by a plus sign to replace a module in the library. Give the name of the module to be replaced after the replacement symbol. Module names have no pathnames and no extensions.

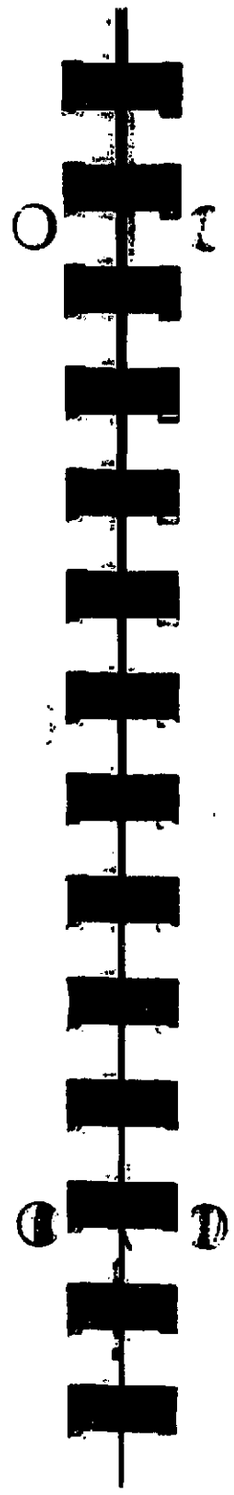
To replace a module, LIB first deletes the given module, then appends the object file having the same name as the module. The object file is assumed to have an ".OBJ" extension and to reside in the current working directory.

`*` Type an asterisk followed by a module name to copy a module from the library file into an object file of the same name. The module remains in the library file. When LIB copies the module to an object file, it adds the ".OBJ" extension and the drive designation and pathname of the current working directory to the module name to form a complete object filename. You cannot override the ".OBJ" extension, drive designation, or pathname given to the object file, but you can later rename the file or copy it to whatever location you like.

`-*` Use the minus sign followed by an asterisk to move an object module from the library file to an object file. This operation is equivalent to copying the module to an object file, as described above, then deleting the module from the library.

6.3.3 List File Prompt

After the "List file" prompt, you can give a filename for a cross-reference listing file. You can specify a full pathname for the listing file to cause it to be created outside your current working directory. You can give the listing file any name and any extension. LIB does not supply a default extension if you omit the extension.



A cross-reference listing file contains two lists. The first is an alphabetical listing of all external (public) symbols in the library. Each symbol name is followed by the name of the module in which it is referenced.

The second list is a list of the modules in the library. Under each module name is an alphabetical listing of the public symbols defined in that module. The default when you omit the response to this prompt is the special filename NUL., which tells LIB *not* to create a listing file.

6.3.4 Output Library Prompt

After the "Output library" prompt you can give the name of a new library file to create with the specified modifications. The default is the current library filename; the original, unmodified library is saved in a library file with the same name but with a ".BAK" extension replacing the ".LIB" extension. This prompt appears only if you specify modifications to the library following the "Operations" prompt.

6.3.5 Using the Command Line

The command line method of starting LIB has the form

`LIB` *library* `[/`*pagesize*`]` `[;]` *operations*`...` `[,` *listing*`]` `[,` *newlibrary*`]`

The entries following LIB are responses to the LIB command prompts.

The *library* entry, with the optional */pagesize* specification, corresponds to the "Library name" prompt. If you want LIB to perform a consistency check on the library, follow the *library* entry with a semicolon (;).

The *operations* entries are any of the operations allowed following the "Operations" prompt. The *listing* entry, if you include it, tells LIB to create a listing file with the given name. The *newlibrary* entry, if it appears, is the name of the revised library.

If you want to create a cross-reference listing, the name of the listing file must be separated from the last *operations* entry by a comma. If you give a filename in the new library field, the library name must be separated from the listing filename or the last *operations* entry by a comma.

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

Examples

1. `LIB LANG-+HEAP;`

2. `LIB C;`

3. `LIB LANG,LCROSS.PUB`

The first command instructs LIB to replace the HEAP module in the library LANG.LIB. LIB first deletes the HEAP module in the library, then appends the object file HEAP.OBJ as a new module in the library. The semicolon command symbol at the end of the command line tells LIB to use the default responses for the remaining prompts. This means that no listing file is created and that the changes are written back to the original library file instead of creating a new library file.

The second command causes LIB to perform a consistency check of the library file C.LIB. No other action is performed. LIB displays any consistency errors it finds and returns to the operating system level. The third command tells LIB to perform a consistency check of the library file LANG.LIB, then output a cross-reference listing file named LCROSS.PUB.

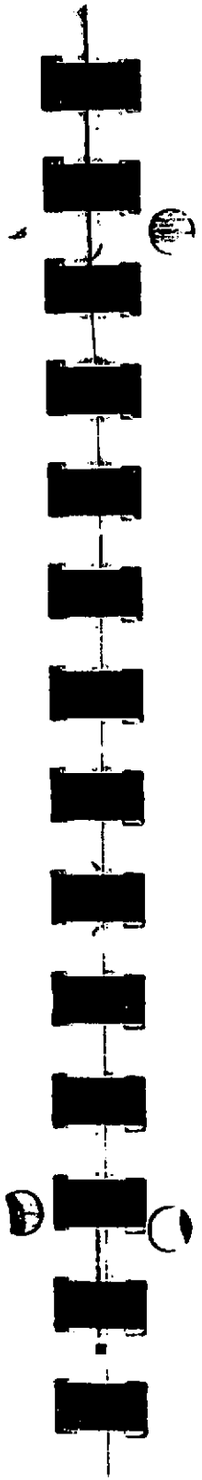
6.3.6 Using a Response File

The command to start LIB with a response file has the form

`LIB @`*response-file*

The *response-file* field is the name of a response file. The *response-file* name may be qualified with a drive and directory specification to name a response file from a directory other than the current working directory.

Before you use this method you must set up a response file containing answers to the LIB prompts. This method lets you conduct the library session without typing responses at the keyboard.



A response file has one text line for each prompt. Responses must appear in the same order as the command prompts appear. Use command symbols in the response file the same way you would for responses typed on the keyboard.

When you run LIB with a response file, the prompts are displayed along with the responses from the response file. If the response file does not contain answers for all the prompts, LIB uses the default responses.

Example

```
SLIBC
+CURSOR+HEAP-HEAP*FOIBLES
CROSSLST
```

This response file causes LIB to delete the module HEAP from the SLIBC.LIB library file; extract the module FOIBLES and place it in an object file named FOIBLES.OBJ; and then append the object files CURSOR.OBJ and HEAP.OBJ as the last two modules in the library. Finally, LIB creates a cross-reference file named CROSSLST.

6.3.7 Extending Lines

If you have many operations to perform during a library session, use the ampersand (&) command symbol to extend the operations line. Give the ampersand symbol after an object module or object filename; do not put the ampersand between an operations symbol and a name.

If you use the ampersand with the prompt method of invoking LIB, the ampersand will cause the "Operations" prompt to be repeated, allowing you to type in more operations. With the response file method, you can use the ampersand at the end of a line and then continue typing operations on the next line.

6.3.8 Terminating the Library Session

At any time, you can use CONTROL-C to terminate a library session. If you type an incorrect response, such as a wrong or incorrectly spelled filename or module name, you must enter CONTROL-C to exit LIB. You can then restart the program.

6.3.9 Selecting Default Responses to Prompts

After any entry but the first, use a single semicolon (;) followed immediately by a carriage return to select default responses to the remaining prompts. You can use the semicolon command symbol with the command line and response file methods of invoking LIB, but it is not really necessary, since LIB supplies the default responses wherever you omit responses.

The default response for the "Operations" prompt is no operation. The library file is unchanged.

The default response for the "List file" prompt is the special filename NUL., which tells LIB not to create a listing file.

The default response for the "Output library" file is the current library name. This prompt appears only if you specify at least one operation following the "Operations" prompt.

6.4 Library Tasks

This section summarizes the library management tasks you can perform with LIB.

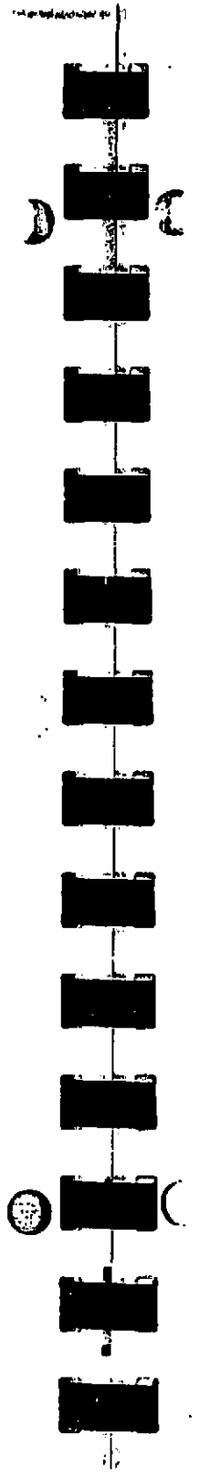
6.4.1 Creating a Library File

To create a new library file, simply give the name of the library file you want to create following the "Library name" prompt. LIB supplies the ".LIB" extension.

The name of the new library must not be the name of an existing file, or else LIB will assume you want to modify the existing file. When you give the name of a library file that does not currently exist, LIB displays the following prompt.

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

Type "yes" to create the file, "no" to terminate the library session.



You can specify a page size for the library when you create it. The default page size is 16 bytes. See the section on setting the page size below for a discussion of this option.

Once you have given the name of the new library file, you can insert object modules into the library by using the add operation (+) following the "Operations" prompt. You can also add the contents of another library if you wish. These options are discussed below in the sections entitled "Adding Library Modules" and "Combining Libraries," respectively.

6.4.2 Modifying a Library File

You can modify an existing library file by giving the name of the library file following the "Library name" prompt. All operations you specify following the "Operations" prompt are performed on that library.

However, LIB lets you keep both the unmodified library file and the newly modified version, if you like. You can do this by giving the name of a new library file following the "Output library" prompt. The modified library file is stored under the new library filename, while the original library file remains unchanged.

If you don't give a filename following the "Output library" prompt, the modified version of the library file replaces the original library file. Even in this case, LIB saves the original, unmodified library file. The unmodified library file has the extension ".BAK" instead of ".LIB". Thus, at the end of the session you have two library files: the modified version with the ".LIB" extension and the original, unmodified version with the ".BAK" extension.

6.4.3 Adding Library Modules

Use the plus sign (+) following the "Operations" prompt to add an object module to a library. Give the name of the object file to be added, without the ".OBJ" extension, immediately after the plus sign.

LIB strips the drive designation and the extension from the object file specification, leaving only the basename. This becomes the name of the object module in the library. For example, if the object file B:\CURSOR.OBJ is added to a library file, the name of the corresponding object module is "CURSOR".

Object modules are always added to the end of a library file.

6.4.4 Deleting Library Modules

Use the minus sign (-) following the "Operations" prompt to delete an object module from a library. Give the name of the module to be deleted immediately after the minus sign. A module name has no pathname and no extension; it is simply a name, like "CURSOR".

6.4.5 Replacing Library Modules

Use a minus sign followed by a plus sign (-+) to replace a module in the library. Give the name of the module to be replaced after the replacement symbol (-+). Remember that module names have no pathnames and no extensions.

To replace a module, LIB first deletes the given module, then appends the object file having the same name as the module. The object file is assumed to have an ".OBJ" extension and to reside in the current working directory.

6.4.6 Extracting Library Modules

Use an asterisk (*) followed by a module name to copy a module from the library file into an object file of the same name. The module remains in the library file. When LIB copies the module to an object file, it adds the ".OBJ" extension and the drive designation and pathname of the current working directory to the module name to form a complete object filename. You cannot override the ".OBJ" extension, drive designation, or pathname given to the object file, but you can later rename the file or copy it to whatever location you like.

Use the minus sign followed by an asterisk (-*) to move an object module from the library file to an object file. This operation is equivalent to copying the module to an object file, as described above, then deleting the module from the library.

6.4.7 Combining Libraries

You can add the contents of a library to another library by using the plus sign (+) with a library filename instead of an object filename. Following the "Operations" prompt, give the plus sign (+) followed by the name of the library whose contents you wish to add to the library being modified. When you use this option you must include the ".LIB" extension of the library filename. Otherwise, LIB assumes that the file is an object file and looks for the file with an ".OBJ" extension.

In addition to allowing MS-DOS libraries as input, LIB also accepts 286 XENIX archives and Intel-format libraries. Thus, you can use LIB to convert libraries from either of these formats to the Microsoft format.

LIB adds the modules of the library to the end of the library being modified. Notice that the added library still exists as an independent library. LIB copies the modules without deleting them.

Once you have added the contents of a library or libraries, you can save the new, combined library under a new name by giving a new name following the "Output library" prompt. If you omit the "Output library" response, LIB saves the combined library under the name of the original library being modified.

6.4.8 Creating a Cross-Reference Listing

Create a cross-reference listing by giving a name for the listing file following the "List file" prompt. If you omit the response to this prompt, LIB uses the special filename NUL., which means that no listing file is created.

You can give the listing file any name and any extension. You can specify a full pathname, including drive designation, for the listing file to cause it to be created outside your current working directory. LIB does not supply a default extension if you omit the extension.

A cross-reference listing file contains two lists. The first is an alphabetical listing of all public symbols in the library. Each symbol name is followed by the name of the module in which it is referenced.

The second list is an alphabetical list of the modules in the library. Under each module name is an alphabetical listing of the public symbols referenced in that module.

6.4.9 Performing Consistency Checks

When you give just a library name followed by a semicolon following the "Library name" prompt, LIB performs a consistency check, displaying messages about any errors it finds. No changes are made to the library. This option is not usually necessary, since LIB automatically checks object files for consistency before adding them to the library.

To produce a cross-reference listing along with a consistency check, use the command line method of invoking LIB. Give the library name followed by a semicolon, then give the name of the listing file. LIB performs the consistency check and then creates the cross-reference listing.

6.4.10 Setting the Library Page Size

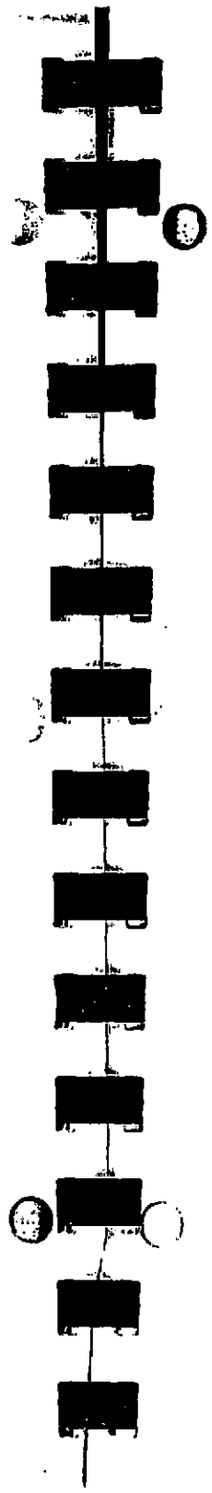
The page size of a library affects the alignment of modules stored in the library. Modules in the library are aligned so that they always start at a position that is a multiple of *n* bytes from the beginning of the file. The value of *n* is the page size. The default page size is 16 for a new library or the current page size for an existing library.

Because of the indexing technique used by LIB, a library with a large page size can hold more modules than a library with a smaller page size. However, for each module in the library, an average of $n/2$ bytes of storage space is wasted (where *n* is the page size). In most cases a small page size is advantageous, and you are advised to use the smallest page size possible.

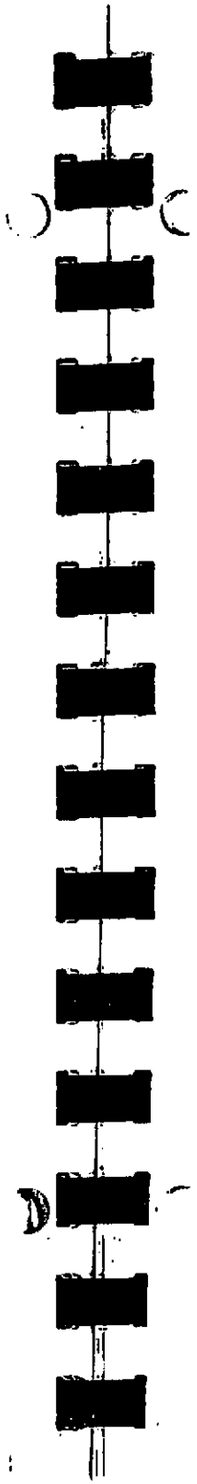
To set the library page size, add a page size option after the library filename following the "Library name" prompt:

library `/pagesize:`*n*

The value of *n* is the new page size. It must be a power of 2 and must fall between 16 and 32,768.



Chapter 7
Advanced Topics



7.1 Introduction

The Microsoft C Compiler offers a number of advanced programming options that give you control over the compilation process and the final form of the executable program. This chapter describes the advanced options.

7.2 Enabling Special Keywords

Option

/Ze

The /Ze option tells the compiler to consider the following identifiers keywords when processing a given file.

far
fortran
huge
near
pascal

You must give this option when compiling any program that uses these identifiers as keywords. The **huge** keyword is not yet implemented but is reserved for future use.

7.3 Packing Structure Members

Option

/Zp

When storage is allocated for structures, structure members larger than a **char** are ordinarily stored beginning at an **int** boundary. To conserve space you may want to store your structures more compactly. The /Zp option causes structure data to be "packed" tightly into memory. This option is also useful when you want to read existing packed structures from a data file.

When you give the /Zp option, each structure member (after the first) is stored beginning at the first available byte, without regard to int boundaries.

On most processors, using the /Zp option results in slower program execution because of the time required to unpack structure members when they are accessed. This option also reduces efficiency when a program accesses 16-bit members (with int type) that begin on odd boundaries.

Example

```
MSC /Zp PROG.C;
```

This command causes all structures in the program PROG.C to be stored without extra space for alignment of members on int boundaries.

7.4 Restricting Length of External Names

Option

/Hnumber

The MSC command allows you to restrict the length of external (public) names by using the /H option. The *number* is an integer specifying the maximum number of significant characters in external names.

When you use the /H option, the compiler considers only the first *number* characters of external names used in the program. The program may contain external names longer than *number* characters; the extra characters are simply ignored.

The /H option is typically used to conserve space or to aid in creating portable programs. The Microsoft C Compiler imposes no restrictions on the length of external names (although it uses only the first 31 characters), but other compilers or linkers may produce errors when they encounter names longer than a predetermined limit.

7.5 Labeling the Object File

Option

/V"string"

Use the /V (for "version") option to imbed a given text *string* into an object file. The quotation marks surrounding the string may be omitted if the string does not contain whitespace characters.

Object files are machine readable but are not easily read and understood by humans. A typical use of the /V option is labeling an object file with a version number or copyright notice.

Example

```
MSC MAIN.C, /V"Microsoft C Compiler Version 3.0";
```

The above command places the string "Microsoft C Compiler Version 3.0" in the object file MAIN.OBJ.

7.6 Suppressing Default Library Selection

Option

```
/Zl
```

Ordinarily the compiler places the names of the default libraries (the standard C library plus the selected floating-point library or libraries) in the object file for the linker to read. This allows the default libraries to be linked with a program automatically.

The /Zl option suppresses the selection of default libraries. No library names are placed in the object file; as a result, the object file is slightly smaller.

The /Zl option is useful when you are building a library of routines. It is not necessary for every routine in the library to contain the default library information. Although the /Zl option saves only a small amount of space for a single object file, the total space savings is significant in a library containing many object modules. When you link a library of object modules created *with* the /Zl option with a C program file compiled *without* the /Zl option, the default library information is supplied by the program file.

Example

```
MSC ONE.C;
MSC /Zl TWO.C;
LINK ONE+TWO;
```

The first two commands create an object file named ONE.OBJ that contains the names of the standard C library (SLIBC.LIB) plus the emulator library and floating-point math library (EM.LIB and SLIBFP.LIB) and an object file named TWO.OBJ that contains no default library information. When ONE.OBJ and TWO.OBJ are linked, the default library information in ONE.OBJ causes the given libraries to be searched for any unresolved references in either ONE.OBJ or TWO.OBJ.

7.7 Controlling Floating-Point Operations

By default, the compiler handles floating-point operations by using calls to an emulator library, which emulates the operation of an 8087 or 80287 coprocessor. If an 8087 or 80287 coprocessor is present at run time, it will be used. The floating-point (/FP) options give you a choice of five different methods of handling floating-point operations.

The advantages and disadvantages of each of the five /FP options are described in Section 3.7 of Chapter 3, "Compiling." You should read the discussion of floating-point options before reading this section. This section discusses two additional ways to control floating-point operations: by changing libraries at link time and by using the NO87 environment variable.

7.7.1 Changing Libraries at Link Time

When you compile using one of the floating-point options, the name of the corresponding library or libraries is placed in the object file for the linker to use. You can cause the linker to use a different floating-point library instead by using the /NOD (for no default library search) option at link time and specifying the name of a different library or libraries. The floating-point library names you can give on the link command line are

1. EM.LIB (the emulator) plus SLIBFP.LIB, MLIBFP.LIB, or LLIBFP.LIB, depending on the memory model
2. 87.LIB (the 8087/80287 library), plus SLIBFP.LIB, MLIBFP.LIB, or LLIBFP.LIB, depending on the memory model
3. SLIBFA.LIB, MLIBFA.LIB, or LLIBFA.LIB (the alternate math library), depending on the memory model

The 8087/80287 library (87.LIB) provides only minimal floating-point support. When you specify this library, an 8087 or 80287 coprocessor must be present at run time or the program will fail.

When you compile using the /FPa, /FPc, or /FPc87 option, you can specify any of the above libraries at link time. However, when you compile using the /FPi or /FPi87 option, you are not allowed to specify the alternate math library at link time; if you want to override the default library at link time, you must use either the emulator library or the 8087/80287 library, as appropriate.

When you use the /NOD option, the linker ignores all default library information in the object file. This means that you must give the name of the standard C library as well as the names of floating-point libraries at link time. Always give the name of the floating-point library or libraries *before* the name of the standard C library on the command line.

Examples

```
1.  MSC /AM CALC;
    LINK CALC+ANOTHER+SUM /NOD,,,MLIBFA, MLIBC;

2.  MSC /FPa CALC;
    LINK CALC+ANOTHER+SUM /NOD,,,EM.LIB+SLIBFP.LIB+SLIBC.LIB;

3.  MSC /FPi87 CALC;
    LINK CALC+ANOTHER+SUM /NOD,,,EM.LIB+SLIBFP.LIB+SLIBC.LIB;
```

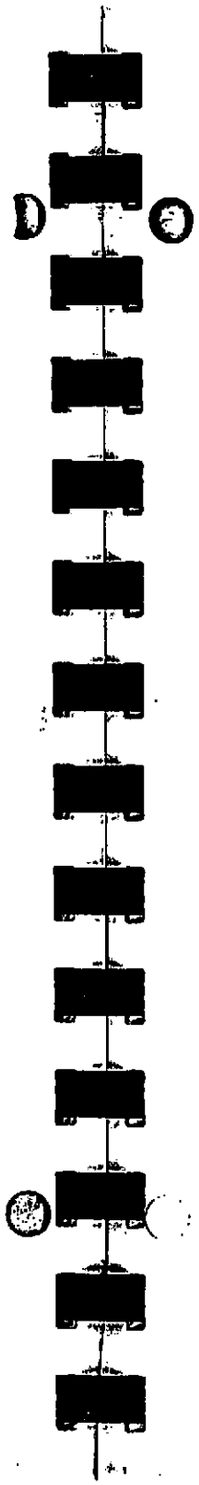
In the first example, the program CALC.C is compiled with the medium model option (/AM). No floating-point option is specified, so the default, /FPc, is used. /FPc generates floating-point calls and specifies the emulator (EM.LIB) plus MLIBFP.LIB in the object file. In the LINK command, the /NOD option is specified and the names of the alternate math library and the standard C library are given in the "Libraries" field. This causes any floating-point calls in CALC.OBJ, ANOTHER.OBJ, and SUM.OBJ to refer to the alternate math library instead of to the emulator. Notice that the medium model libraries (MLIBFA.LIB and MLIBC.LIB) must be used.

In the second example, CALC is compiled as small model (by default) and with the alternate math option (/FPa). The LINK command specifies the /NOD option and gives the names EM.LIB, SLIBFP.LIB, and SLIBC.LIB in the "Libraries" field, causing all floating-point calls to refer to the emulator library instead of the alternate math library.

In the third example, CALC.C is compiled with the /FPi87 option. The library names 87.LIB and SLIBFP.LIB are placed in the object file. The LINK command overrides the default library specification by giving the /NOD option and the names of the emulator library (EM.LIB), SLIBFP.LIB, and the standard library (SLIBC.LIB).

7.7.2 Using the NO87 Environment Variable

Programs compiled using the /FPc or /FPi option will automatically use an 8087/80287 coprocessor at run time if one is installed. You can override this and force the use of the emulator instead by setting an environment variable named NO87. (See Section 2.5 of Chapter 2, "Getting Started," or your MS-DOS documentation for a discussion of environment variables.)



If NO87 is currently set to any value when the program is executed, use of the 8087/80287 coprocessor is suppressed. The value of the NO87 setting is printed on the standard output as a message. The message is only printed if an 8087/80287 is present and suppressed; if no coprocessor is present, no message appears. If you don't want a message to be printed, set NO87 equal to one or more spaces; nothing will be printed.

Note that only the presence or absence of the NO87 definition is important in suppressing use of the coprocessor. The actual value of the NO87 setting is only used for printing the message.

The NO87 variable takes effect with any program linked with the emulator library (EM.LIB). It has no effect on programs linked with 87.LIB or SLIBFA.LIB (MLIBFA.LIB or LLIBFA.LIB for medium and large model programs).

Examples

```
1.  SET NO87=Use of coprocessor suppressed
2.  SET NO87=space
```

The first example causes the message "Use of coprocessor suppressed" to appear on the screen when a program that can use an 8087 or 80287 is executed.

The second example sets the NO87 variable to the space character. Use of the coprocessor is still suppressed, but no message is displayed.

7.8 Advanced Optimizing

This section describes additional optimizing procedures that can be used with the optimizing options described in Section 3.11 of Chapter 3, "Compiling," to create more efficient programs from your code.

7.8.1 Removing Stack Probes

Option

`/Gs`

You can reduce the size of a program and speed up execution slightly by using the /Gs option to remove all stack probes. A stack probe is a short routine called by a function to check the program stack for available space. The stack probe routine is called at every entry point. Ordinarily, the stack probe routine generates a message when it detects a stack overflow. When the /Gs option is used, no message is printed.

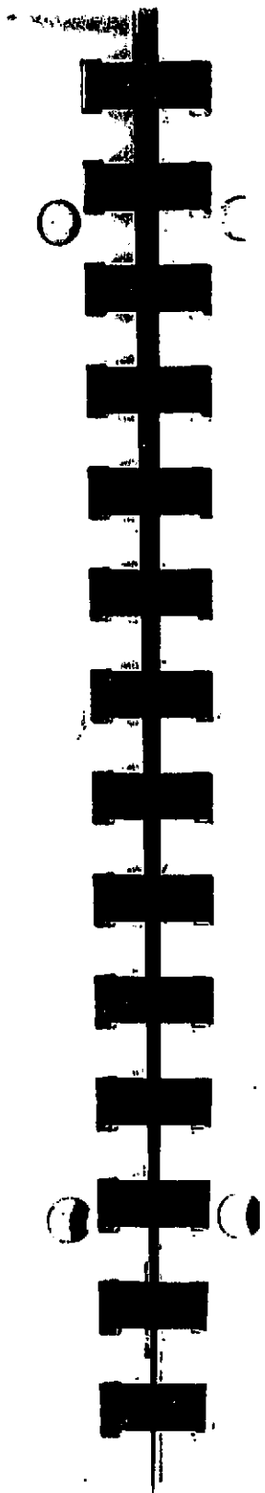
The /Gs option is useful when a program is known not to exceed the available stack space. For example, stack probes may not be needed for programs that make very few function calls.

Although the /Gs option, combined with the /Osa option, described with the O string options in Section 3.11 "Optimizing," makes the smallest possible program, it should be used with great care. Removing stack probes from a program means that some execution errors may not be detected.

Example

```
MSC FILE.C /Ota /Gs;
```

This example optimizes the file FILE.C by removing stack probes with the /Gs option and relaxing alias checking with the /Ota option. The letter "t" in the /Ota option tells the compiler to favor execution time over code size in the optimization.



7.8.2 Maximum Optimization

Option

`/Ox`

The /Ox option is a shorthand way to combine optimizing options to produce the smallest possible program. Its effect is equivalent to

`/Oas /Gs`

Thus, the /Ox option removes stack probes, relaxes alias checking, and favors code size over execution time.

7.9 Modifying the Executable File

The EXEPACK and EXEMOD utilities (supplied with your compiler software) allow you to modify an executable program file. The EXEPACK utility "compresses" the executable file by removing sequences of characters from the file and by optimizing the relocation table. Programs that have large data structures (arrays and `struct` types) contain sequences of null characters to initialize the data structures. By using EXEPACK on such program files, you can significantly reduce the size of the executable file and the time required for loading.

The EXEMOD utility allows you to examine and modify file header information. The program assumes that you are familiar with the fields and format of the file header. See your *Microsoft MS-DOS Programmer's Reference Manual* for details.

The following sections explain how to invoke and pass arguments to the EXEPACK and EXEMOD programs.

7.9.1 The EXEPACK Utility

Command

EXEPACK *executable-file output-file*

The EXEPACK utility compresses sequences of identical characters from the given *executable-file* and optimizes the relocation table. The compressed file is written to the output file, and the original file is unmodified.

The EXEPACK utility produces self-explanatory error messages if it is unable to compress the given file. See Section E.6 of Appendix E, "Error Messages," for a listing of these error messages.

7.9.2 The EXEMOD Utility

Command

EXEMOD *executable-file* [/stack *n*] [/min *n*] [/max *n*]

The EXEMOD utility modifies fields in the header according to instructions given on the command line. To display the header fields without modifying them, give the *executable-file* without any options.

The options are shown with the forward slash (/) option character, but a hyphen (-) may be used instead if you prefer. They can be given in either uppercase or lowercase. The options have the effects listed below.

Option	Effect
/stack *n*	Sets the initial SP (stack pointer) value to *n*, where *n* is a hexadecimal value in bytes. The minimum allocation value is adjusted upward if necessary.
/min *n*	Sets the minimum allocation value to *n*, where *n* is a hexadecimal value in paragraphs. The actual value set may be different from the requested value if adjustments are necessary to accommodate the stack.
/max *n*	Sets the maximum allocation to *n*, where *n* is a hexadecimal value in paragraphs. The maximum allocation value must be greater than or equal to the minimum allocation value.

Notice that the modifications you can make with the /stack and /max options can also be made by relinking the program with the corresponding linker options (/STACK and /CPARMAXALLOC). The advantage of the EXEMOD utility is that it modifies the executable file directly, without requiring the program to be relinked.

The EXEMOD program produces self-explanatory error messages when it is unable to carry out the given instructions. For a listing of these messages, see Section E.7 of Appendix E, "Error Messages."

Warning

The /stack option can only be used on 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 with other programs may cause the program to fail.

7.10 Controlling Binary and Text Modes

Most C programs use one or more data files for input and output. Under MS-DOS, data files are ordinarily processed in "text" mode. In text mode, carriage return-linefeed combinations (CR-LF) are translated into a single linefeed (LF) character on input. Linefeed characters are translated to carriage return-linefeed combinations on output.

In some cases you may want to process files without making these translations. In binary mode, carriage return-linefeed translations are suppressed.

Standard library routines such as *fopen* or *open* give you the option to override the default mode when you open a particular file. You can also change the default mode for an entire program from text to binary mode. Do this by linking your program with the file BINMODE.OBJ, which is supplied as part of your C compiler software. Simply add the filename or pathname of BINMODE.OBJ to the list of filenames when you link your program.

When you link with BINMODE.OBJ, all files opened in your program default to binary mode, with the exception of *stdin*, *stdout*, and *stderr*. However, linking with BINMODE.OBJ does not force you to process all data files in binary mode. You still have the option to override the default mode when you open the file.

Use the *setmode* library function when you want to change the default mode of *stdin*, *stdout*, or *stderr* from text to binary, or the default mode of *stdaux* or *stdprn* from binary to text. The *setmode* function can change the current mode for any file and is primarily used for changing the mode of *stdin, stdout, stderr, stdaux,* and *stdprn*.

7.11 Mixed Model Programming

Option

/Astring

The Microsoft C Compiler defines three standard memory models (small, medium, and large) to accommodate programs with differing memory requirements. For an introduction to memory models, along with a discussion of the standard memory models and an alternate form of the /A option, see Section 3.13 of Chapter 3, "Compiling."

One limitation of the predefined memory model structure is that all pointers for code or data change size at once when you change memory models. To overcome this limitation, the Microsoft C Compiler lets you override the default addressing convention for a given memory model and access an item with either a "near" or a "far" pointer. This is particularly useful when you have a very large or infrequently used data item that you want to access from a small or medium model program. You can access that item in another segment, saving space in your default data segment.

There are two ways to create mixed model programs. You can use the special keywords **near** and **far** in your program declarations to override the default addressing conventions for individual items. In a small model program, the **far** keyword lets you access data and functions in segments outside the program. In medium and large model programs, **near** lets you access data with just an offset. The **near** and **far** keywords can be used with a standard memory model to overcome addressing limitations for particular items without changing the addressing conventions for the program as a whole.

In the second method you combine features of the standard memory models to create your own customized memory model. To do this, you use the /A option followed by a string of three letters that tells the compiler the attributes of the memory model you want to use. The three fields of the string correspond to the code pointer size, the data pointer size, and the stack and data segment setup. The letters allowed in each field are unique, so you can give them in any order after /A. All three letters must be present.

Using the **near** and **far** keywords is the recommended procedure for creating mixed model programs. These keywords are particularly useful when you have a very large or infrequently used data item that you want to access from a small or medium model program. You can use the **far** keyword to cause a new segment to be allocated for the data item, and then access that item with a far pointer, while still using near pointers (the default) for your other data. You must take care when calling library routines in programs that use the **near** and **far** keywords (for example, you cannot pass a **far** data item to a small model library routine), but in most cases you can easily use the standard libraries (small, medium, or large model) with these programs.

When you use a customized memory model (the /A*string* option), you are responsible for making sure that your program reflects the customized model and handles pointers and the stack and data segments consistently and correctly. Moreover, you are responsible for selecting and modifying a library for your program. Library support is not guaranteed for mixed model programs, and adapting one of the standard libraries (small, medium, or large model) to produce the appropriate combination of **near** and **far** routines and data may require considerable time and effort.

The sections below discuss the **near** and **far** keywords, pointer size, the stack and data segment setup, and library support for mixed model programs.

7.11.1 Using the Near and Far Keywords

The **near** and **far** keywords let you override the default addressing conventions of a particular memory model for an individual item (either code or data) without changing the default addressing conventions. The **near** and **far** keywords are special type modifiers you may use in your program declarations to override the default length and meaning of the address of a given variable. The **near** keyword defines an object with a 16-bit address. The **far** keyword defines an object with a full 32-bit segmented address. Any data item or function can be accessed with a **far** pointer.

When you use the **near** and **far** keywords in a program, you must specify the /Ze option at compile time to enable the keywords. (See Section 7.2, "Enabling Special Keywords.") Without the /Ze option, the compiler will treat **near** and **far** as ordinary identifiers, causing program errors.

Since there is no type-checking between items in separate source files, the **near** and **far** keywords should be used with great care.

The examples in the Table 7.1 illustrate the **far** and **near** keywords as used in declarations in a small model program.

Table 7.1

Uses of near and far Keywords with Small Model

Declaration	Address Size	Item Size
`char c;`	near (16 bits)	8 bits (data)
`char far d;`	far (32 bits)	8 bits (data)[a]
`char *p;`	near (16 bits)	16 bits (near pointer)
`char far *q;`	near (16 bits)	32 bits (far pointer)
`char * far r;`	far (32 bits)	16 bits (near pointer)[b]
`char far * far s;`	far (32 bits)	32 bits (far pointer)[c]
`int foo();`	near (16 bits)	function returning 16 bits
`int far foo();`	far (32 bits)	function returning 16 bits[d]

[a] This causes the variable to be allocated to a new segment.

[b] This example has no meaning; it is shown for syntactic completeness only.

[c] This is similar to accessing data in a long model program.

[d] This example leads to trouble in small model programs. In small model programs, calls to library functions are near. The far call shown in the example changes the CS (code segment) register, making the offset values used for library calls invalid.

The following example is from a medium model compilation.

```
int near foo();
```

This declaration prepares for a near function call in an otherwise far (calling) program.

7.11.2 Creating Customized Memory Models

The /A *string* option lets you change the attributes of the standard memory models to create your own memory models. With the /A *string* option you can control the default code pointer size, data pointer size, and the stack and data segment setup, as described below.

The standard memory model options (/AS, /AM, and /AL) can also be specified in the /A *string* form. The standard memory model options are listed with their /A *string* equivalents below.

/AS	/Asnd
/AM	/Alnd
/AL	/Alfd

7.11.2.1 Code Pointers

Options

/Aszz
/Alzz

The letter "s" (for "short") tells the compiler to generate near (16-bit) pointers for all code items. This is the default for small model programs.

The letter "l" (for "long") means that far (32-bit) pointers are used to address all code items. Far pointers are the default for medium and large model programs.

The letters "l" and "s" are used for code pointers to distinguish them in the memory model string from the letters for data pointers, discussed below. The terms "short" and "long" are equivalent to "near" and "far", respectively.

7.11.2.2 Data Pointers

Options

/An*xx*
/Af*xx*

Two sizes are available for data pointers: near and far. The letter "n" tells the compiler to use near (16-bit) pointers for all data. This is the default for small and medium model programs.

The letter "f" specifies that all data pointers are far (32-bit). This is the default for large model programs. When far data pointers are used, no single data item may be larger than a segment (64K bytes) because address arithmetic is done only on 16 bits (the offset portion) of the address. All elements or members of the data item must be within the same segment to be referenced correctly.

7.11.2.3 Setting Up Segments

Options

/Ad*xx*
/Au*xx*
/Aw*xx*

The letter "d" tells the compiler that SS equals DS; that is, the stack and data segment pointers contain the same address. This is the default for all programs. In small and medium model programs, the stack and all data segments combined must occupy less than 64K bytes; thus, any data item is accessed with just 16-bit offset from the address in SS and DS.

In large model programs, global and static data are placed in a single segment called the "default" data segment. The address of this segment is stored in the DS and SS registers. All pointers to data, including pointers to local data (the stack), are full 32-bit addresses. This is important to keep in mind when passing pointers as arguments in large model programs.

The letter "u" allocates different segments for the stack and the data segments. Each object file (module) is allocated its own segment for global and static data items. When the "u" option is specified, the address in the DS register is saved at the entry point for each program module, and the new DS value for the module is loaded into the register. The previous DS value is restored at the module exit point. Thus, only one data segment is accessible at any given time.

A single segment is allocated for the stack, and its address is stored in the stack register. The stack cannot be placed in a data segment since it must be available throughout the entire program.

The "w" option sets up a separate stack segment, like the "u" option, but does not automatically load the DS register at each module entry point. This option is typically used when writing application programs that interface with an operating system or with a program running at the operating system level. The operating system or the program running beneath the operating system actually receives the data intended for the application program and places it in a segment; then it must load the DS register with the segment address for the application program.

7.11.3 Library Support

Most C programs make function calls to the routines in the C run-time library. Library support is provided for the three standard memory models (small, medium, and large) through three separate run-time libraries. When you write mixed model programs, you are responsible for determining which library (if any) is suitable for your program and for ensuring that the appropriate library is used.

When using the **near** and **far** keywords to modify addressing conventions for particular items, you can usually use one of the standard libraries (small, medium, or large) with your program. However, you must take care when calling library routines; for example, you cannot pass **far** data items to a small model library routine.

Library support for programs using a customized memory model (the */Astring* option) is not guaranteed, and you will probably need to spend some time creating a customized library to be used with your customized memory model. It is recommended that you use the /NOD (for no default library search) option when linking, and then explicitly specify the library files and object files you want to use. Be sure to use the correct startup routine for your memory model; for example, if the source file containing the "main" function is compiled with far code pointers and near data

pointers as the default, you should use the startup file from the medium model library.

Moreover, you must make sure that any library routines called in those files are taken from the appropriate library. For example, suppose you compile the "main" function with the small memory model option (/AS) and compile another source file from the same program with the option /Alnd (which makes all code pointers far pointers). If the second program file contains calls to library routines, you can create a customized library file (using the LIB utility) that replaces each routine called in the second file with the corresponding large model version.

Notice that you must declare these large model routines appropriately as well. This procedure requires great caution, however, because the replaced routines may be called in other source files of the program as well. Moreover, some routines in the run-time library call other run-time routines to accomplish their tasks.

7.12 Setting the Data Threshold

Options

/Gt[*n*]

By default, the compiler allocates all static and global data items to the default data segment. The /Gt option causes all data items greater than *n* bytes to be allocated to a new data segment. When *n* is specified, it must follow the /Gt option immediately, with no intervening spaces. When *n* is omitted, the default threshold value is 256.

You can only use the /Gt option with large model programs since small and medium model programs have only one data segment. It is particularly useful with programs that have more than 64K bytes of static and global data in small data items.

7.13 Naming Modules and Segments

Options

/NMmodule-name
/NTtext-segment-name
/NDdata-segment-name

"Module" is another name for an object file created by the C compiler. Every module has a name. The compiler uses this name in error messages if problems are encountered during processing. The module name is usually the same as the source filename. You can change this name using the /NM (for "name module") option. The new *module-name* can be any combination of letters and digits.

A "segment" is a contiguous block of binary information (code or data) produced by the C compiler. Every module has at least two segments: a text segment containing the program instructions and a data segment containing the program data. Each segment in every module has a name. This name is used by the linker to define the order in which the segments of the program appear in memory when loaded for execution. (Note that the segments in the group named DGROUP are an exception; see Section 8.1.1.1, "Segments," in Chapter 8, "Interfaces with Other Languages," for details.)

Text and data segment names are normally created by the C compiler. These default names depend on the memory model chosen for the program. For example, in small model programs the text segment is named "_TEXT" and the data segment is named "_DATA". These names are the same for all small model modules, so all text segments from all modules are loaded as one contiguous block, and all data segments from all modules form another contiguous block.

In medium model programs, the text from each module is placed in a separate segment with a distinct name, formed by using the module basename along with the suffix "_TEXT". The data segment is named "_DATA", as in the small model.

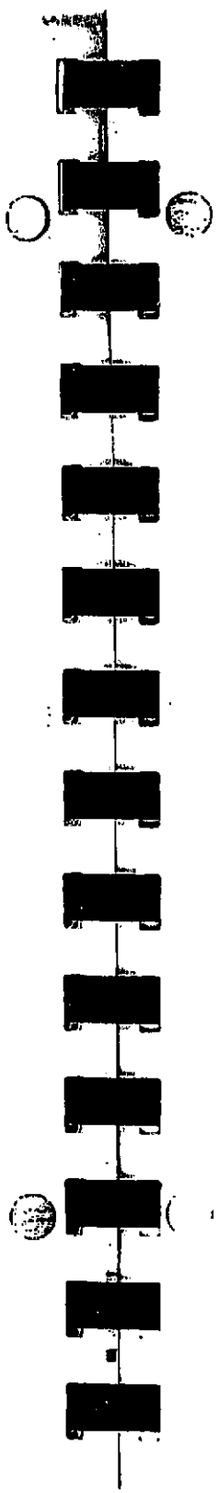
In large model programs, the text and data from each module are loaded into separate segments with distinct names. Each text segment is given the name of the module plus the suffix "_TEXT". The data from each segment are placed in a private segment with a unique name (except for global and static data placed in the default data segment). The naming conventions for text and data segments are summarized in Table 7.2.

Table 7.2
Segment Naming Conventions

Model	Text	Data	Module
Small	_TEXT	_DATA	*filename*
Medium	*module*_TEXT	_DATA	*filename*
Large	*module*_TEXT	_DATA[a]	*filename*

[a] Name of default data segment.

You can override the default names used by the C compiler (thus overriding the default loading order) by using the /NT (for "name text") and /ND (for "name data") options. These options set the names of the text and data segments, in each module being compiled, to a given name. The *text-segment-name* and *data-segment-name* can be any combination of letters and digits.



Chapter 8
Interfaces with Other Languages

8.1 Assembly Language Interface

This section explains how to use 8086/8088 assembly language routines with C language programs and functions. In particular, it outlines the segment model used by the Microsoft C compiler and explains how to call assembly language routines from C language programs and vice versa. This assembly language interface is especially useful for those assembly language programmers who want to use the functions of the standard C library and other C libraries.

If you have assembly language programs that were written to work with the Microsoft C Compiler Version 2.03 or earlier, turn to Section D.4 of Appendix D, "Converting from Previous Versions of the Compiler," for a discussion of differences between the assembly language interface for this compiler and earlier versions.

8.1.1 Segment Model

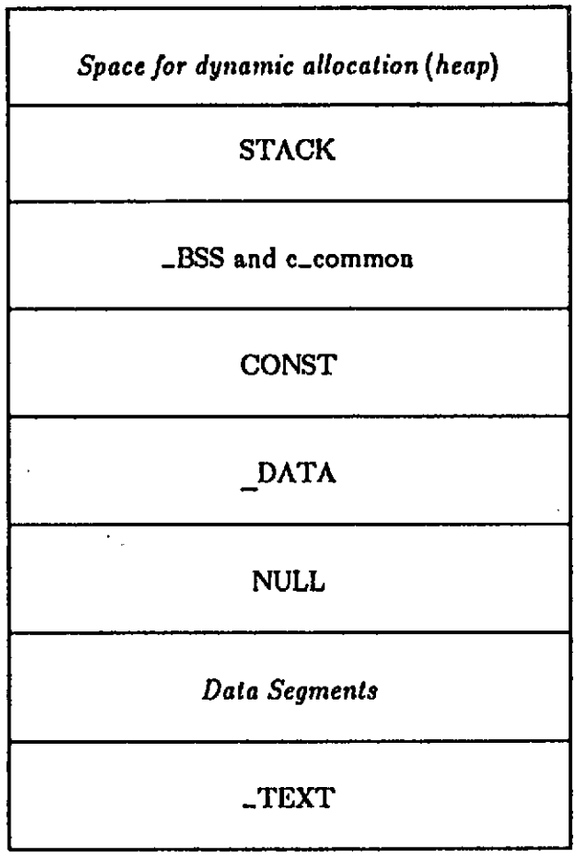
This section describes the run-time structure of Microsoft C programs. Memory on the 8086/8088 processor is divided into segments, each containing up to 64K bytes. When a program is linked, the segments are organized into groups and classes. The segments, groups, and classes of Microsoft C programs are described below.

8.1.1.1 Segments

Figure 8.1 shows the order of primary segments of a C program in memory, from the highest memory location to the lowest. When you look at a map file produced by linking a C program, you may notice other segments in addition to the names listed below. These additional segments have specialized uses for Microsoft languages and should not be used by other programs.

The /DOSSEG option available with Microsoft LINK produces the ordering shown here. Since this is the default ordering for C programs, you do not need to use /DOSSEG with C programs, but you may find it useful when linking assembly language routines.

HIGH MEMORY



LOW MEMORY

Figure 8.1 Segment Setup in C Programs

The "heap" is the area of unallocated memory that is available for dynamic allocation by the program. Its size varies, depending on the program's other storage requirements.

The segment contents are listed below.

STACK	The STACK segment contains the user's stack, which is used for all local data items.

_BSS	The _BSS segment contains all uninitialized static data items except for those that are explicitly declared as far items in the source file.
c_common	The c_common segment contains all uninitialized global data items for small and medium model programs. In large model programs, this type of data item is placed in a data segment with class FAR_BSS.
CONST	The CONST segment contains all constants that can only be read. These include floating-point constants, as well as segment values for data items declared far in the source file or data items that are forced into their own segment by use of the /Gt option. Writing to string literals is allowed in C. Thus, strings are stored in the _DATA segment rather than the CONST segment.
_DATA	The _DATA segment is the default data segment. All initialized global and static data reside in this segment for all memory models, except for data explicitly declared far or data forced into different segments by use of the /Gt option.
NULL	The NULL segment is a special purpose segment that occurs at the beginning of DGROUP. The NULL segment contains the compiler copyright notice. This segment is checked before and after the program executes. If the contents of the NULL segment change in the course of program execution, it means that the program has written to this area, usually by an inadvertent assignment through a null pointer. The error message "Null pointer assignment" is displayed to notify the user.
Data Segments	Initialized static and global far data items are always placed in their own segments with class name FAR_DATA. This allows the linker to combine these items so that they all come before DGROUP. Uninitialized static and global far data items are placed in segments that have class FAR_BSS. Again, this allows the linker to place these items between the TEXT segment or segments and DGROUP. In large model programs global uninitialized data are treated as though

declared as far (unless specifically declared near) and given class FAR_BSS.

_TEXT

The _TEXT segment is the code segment. In small model programs the code for all modules is combined in this segment. In medium and large model programs each module is allocated its own text segment. The segments are not combined, so there are multiple text segments in medium and large model programs. Each segment in a medium or large model program is given the name of the module plus the suffix _TEXT.

When implementing an assembly language routine to call or be called from a C program, you will probably refer to the _TEXT and _DATA segments most frequently. The code for the assembly language routine should be placed in the _TEXT segment (or *module-name*_TEXT for medium and large model programs). Data should be placed in whichever segment is appropriate to their use, as described above. Usually this is the default data segment, _DATA.

8.1.1.2 Groups

All segments with the same group name must fit into a single physical segment, which is up to 64K bytes long. This allows all segments in a group to be accessed through the same segment register. The Microsoft C compiler defines one group named DGROUP.

The NULL, _DATA, CONST, BSS, c_common, and STACK segments are grouped together in the data group, called DGROUP. This allows the compiler to generate code for accessing data in each of these segments without constantly loading the segment values or using many segment overrides on instructions. DGROUP is addressed using the DS or SS segment register. DS and SS always contain the same value except when the "u" or "w" option of the /A option is used.

In large model programs, or small and medium model programs using far data declarations, DS may be changed temporarily to a different value to allow the program to access data outside the default data segment. The ES register may also be used in these cases.

SS is never changed; its segment registers always contain abstract "segment values" and the contents are never examined or operated on. This provides compatibility with the Intel 80286 processor.

In small model programs, there is only one text segment, named _TEXT. In medium and large model programs, the names of all text segments must end with the suffix "_TEXT". The text segments are not grouped.

8.1.1.3 Classes

Table 8.1 gives the align type, combine class, class name, and group for each segment discussed above. All segments with the same class name are loaded next to each other.

Table 8.1

Segments, Groups, and Classes for Standard Memory Models

Memory Model	Segment Name	Align Type	Combine Class	Class Name	Group
Small	_TEXT	byte	public	CODE	
	Data Segments[a]	para	private	FAR_DATA	
	Data Segments[b]	para	public	FAR_BSS	
	NULL	para	public	BEGDATA	DGROUP
	_DATA	word	public	DATA	DGROUP
	CONST	word	public	CONST	DGROUP
	_BSS	word	public	BSS	DGROUP
	STACK	para	stack	STACK	DGROUP
Medium	*module*_TEXT	byte	public	CODE	
	.				
	.				
	.				
	Data Segments[a]	para	private	FAR_DATA	
	Data Segments[b]	para	public	FAR_BSS	
	NULL	para	public	BEGDATA	DGROUP
	_DATA	word	public	DATA	DGROUP
	CONST	word	public	CONST	DGROUP
	_BSS	word	public	BSS	DGROUP
	STACK	para	stack	STACK	DGROUP
Large	*module*_TEXT	byte	public	CODE	
	.				
	.				
	.				
	Data Segments[c]	para	private	FAR_DATA	
	Data Segments[d]	para	public	FAR_BSS	
	NULL	para	public	BEGDATA	DGROUP
	_DATA	word	public	DATA	DGROUP
	CONST	word	public	CONST	DGROUP
	_BSS	word	public	BSS	DGROUP
	STACK	para	stack	STACK	DGROUP

[a] Segment(s) for initialized far data.

[b] Segment(s) for uninitialized far data.

[c] Segment(s) for initialized global and static data.

[d] Segment(s) for uninitialized global and static data.

8.1.2 The C Calling Sequence

To receive values from C language function calls or to pass values to C functions, assembly language routines must follow the C argument passing conventions. C language function calls pass their arguments to the given functions by pushing the value of each argument onto the stack. The call pushes the value of the last argument first and the first argument last. If an argument is an expression, the call computes the expression's value before pushing it onto the stack.

Arguments with **char**, **short**, **int**, **unsigned char**, **unsigned short**, or **unsigned int** type occupy a single word (16 bits) on the stack. Arguments with **long** or **unsigned long** type occupy a double word (32 bits); the value's high order word is pushed first. Arguments with **float** type are converted to **double** type (64 bits). Note that **char** type arguments are sign-extended to **int** type before being pushed on the stack; **unsigned char** type arguments are zero-extended to **unsigned int** type.

Pointers occupy either 16 or 32 bits, depending on the memory model, the type of item addressed (code or data), and whether the pointer is modified with a **near** or **far** declaration. The segment value of far pointers is pushed first, then the offset.

If an argument is a structure, the function call pushes the last word of the structure first and each successive word in turn until the first word is pushed. Arrays are passed by reference; the array identifier evaluates to the array address, which is used to access the array.

After a function returns control to a routine, the calling routine is responsible for removing arguments from the stack.

8.1.3 Entering an Assembly Routine

Assembly language routines that receive control from C function calls should preserve the contents of the BP, SI, and DI registers and set the BP register to the current SP register value before proceeding with their tasks. (It is not necessary to preserve the contents of SI and DI if the assembly language routine does not modify them.)

If the assembly routine modifies the contents of the SS (stack segment), DS (data segment), and CS (code segment) registers, their values should be saved on entry and restored at exit. The values of SS and DS are always equal in C programs unless the "u" or "w" letter of the /A option is

specified to set up separate stack and data segments.

The following example illustrates the recommended instruction sequence for entry to an assembly language routine.

```
entry:
        push    bp
        mov     bp,sp
        push    di
        push    si
```

This is the same sequence used by the C compiler.

If this sequence is used, the last argument pushed by the function call (which is also the first argument given in the call's argument list) is at address [bp+4] for a near function call, [bp+6] for a far function call. Subsequent arguments begin at [bp+6], [bp+8], or [bp+10], depending upon the size of the first argument and whether the function call is near or far. If the first argument is a single word and the function call is near, the next argument starts at [bp+6]. If the first argument is a single word and the function call is far, or the first argument is a double word and the function call is near, the next argument starts at [bp+8]. If the first argument is a double word and the function call is far, the next argument starts at [bp+10].

Notice that the push instructions in the above sequence are not necessary if the assembly language routine does not modify the contents of the SI and DI registers, which are used by the compiler to store register variables.

It is recommended that macros be written and used to distinguish between near and far function calls and returns. Such macros make the code more readable and can help to insulate a program from changes in the calling sequence.

8.1.4 Return Values

Assembly language routines that wish to return values to a C language program or receive return values from C functions must follow the C return value conventions. The conventions are shown in Table 8.2.

Table 8.2
C Return Value Conventions

Return Value Type	Register
char	AX
short	AX
int	AX
unsigned char	AX
unsigned short	AX
unsigned int	AX
long	high order word in DX; low order word in AX
unsigned long	high order word in DX; low order word in AX
struct or union	address of value in AX; value must be in a static area in memory
float or double	address of value in AX; value must be in a static area in memory
near pointer	AX
far pointer	segment selector in DX; offset in AX

8.1.5 Exiting a Routine

Assembly language routines that return control to C programs should restore the values of the BP, SI, and DI registers before returning control. (The contents of the SI and DI registers do not have to be restored if the entry sequence did not push them.) The following example illustrates the recommended instruction sequence for exiting a routine called by a small model program.

```
pop     si
pop     di
mov     sp,bp
pop     bp
ret
```

This sequence does not change the AX, BX, CX, or DX registers or any of the segment registers. The sequence does not remove arguments from the stack; this is the responsibility of the calling routine.

Notice that the "pop" instructions for SI and DI in the above sequence are not necessary if the contents of the SI and DI registers are not modified by the assembly language routine and were not saved on entry.

8.1.6 Naming Conventions

An assembly language routine can access globally visible items (data or functions) in a C program by prefixing the item name with an underscore (_). (C items declared as **static** cannot be accessed.) For example, a C function named *add* can be accessed in an assembly language program by declaring the name *_add* as external.

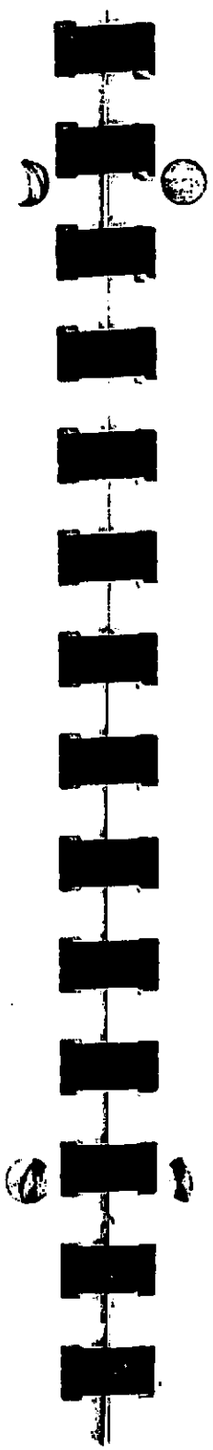
For a C program to access an assembly language routine or data item, the name of the assembly language item must begin with an underscore (_). The C program refers to the assembly language item without the underscore. For example, a C program could call a publicly defined assembly language routine named *_mix* by declaring

```
extern mix();
```

If the assembly language name does not begin with an underscore, it cannot be accessed in a C program.

The C compiler reserves some identifiers beginning with two underscores for internal use. You should avoid using identifiers with two leading underscores in your assembly routines and identifiers with one leading underscore in your C source files, as these identifiers may conflict with internal names.

Some assemblers translate all lowercase letters to uppercase, or vice versa. Since the C language is case-sensitive, this can pose problems. Check your assembler documentation for information on this topic. The Microsoft Macro Assembler, Version 3.0 or later, offers an option to control case sensitivity.



8.1.7 Register Considerations

The SI and DI registers are used to store the values of variables given **register** storage in a C program. An assembly language routine that changes the SI and DI registers is responsible for saving their contents on entry and restoring them before exiting.

The C compiler assumes that the direction flag is always cleared. If your assembly routine sets the direction flag, be sure to clear it (using the "cld" instruction) before returning.

If the assembly routine modifies the contents of the SS (stack segment), DS (data segment), and CS (code segment) registers, their values should be saved on entry and restored at exit. The values of SS and DS are always equal in C programs unless the "u" or "w" letter of the /A option is specified to set up separate stack and data segments.

8.1.8 Program Example

To illustrate the assembly language interface, consider the following example. The example assumes that the C program is small model.

```
int a=1, b=1, c;

main()
{
        c = add(a,b);
}

add(i,j)
int i,j;
{
        return(i+j);
}
```

If the *add* function is written as an assembly language routine instead of a C function, it must save the proper registers; retrieve the arguments from the stack; add the arguments; place the return value in the AX register; and then restore registers and return control. Here is an example of how the routine can be written. Notice that preserving and restoring SI and DI is shown for illustration, although the procedure is not strictly necessary in this case. (If the assembly routine were written to work with a medium or large model C program, the *_add* procedure would be declared FAR instead of NEAR.)

```
; i = [bp + 4]
; j = [bp + 6]

_add    PROC NEAR
        push    bp
        mov     bp,sp
        push    di
        push    si

        mov     ax,[bp+4]
        add     ax,[bp+6]

        pop     si
        pop     di
        mov     sp,bp
        pop     bp
        ret

_add endP
```

If, on the other hand, the C function is to be called by an assembly language routine, the routine must contain instructions that push the arguments on the stack in the proper order, call the function, and clear the stack. It may then use the return value in the AX register. These instructions are shown in the following example.

```
push    [_b]
push    [_a]
call    _add
add     sp,4
mov     [_c],ax
```

8.2 Calling FORTRAN and Pascal Routines

This section describes how to use the **fortran** and **pascal** keywords in C programs to declare routines written in Microsoft FORTRAN or Microsoft Pascal. To call such routines, you must have Version 3.3 or later of the Microsoft FORTRAN or Microsoft Pascal compilers.

Writing mixed-language programs requires that you have a thorough understanding of the languages involved. For example, you must consider

- the correspondence between data types in different languages
- the rules for converting, passing, and returning values in different languages
- naming conventions
- library support and compatibility between libraries
- the memory models available in each language

This section discusses only the syntax of **pascal** and **fortran** declarations and does not attempt to cover the many issues that arise in mixed language programming. For a complete discussion of mixed language programming, see your Pascal or FORTRAN manual, Version 3.3 or later.

The special keywords **fortran** and **pascal** let you declare external FORTRAN and Pascal routines in C programs. To enable the keywords, you must use the /Ze option when compiling.

You can also use the **fortran** and **pascal** keywords when declaring a C function. In this case the keyword instructs the C compiler to use FORTRAN or Pascal conventions for entry and exit sequences when compiling the C function. This option is useful when you want to call a C function from within a FORTRAN or Pascal routine.

You declare FORTRAN and Pascal routines in the same manner that you declare C functions: you specify the function identifier, the return type, and the type and number of arguments to the function. (See the *Microsoft C Language Reference* for a complete discussion of the syntax of function declarations.)

The following additional rules apply to **fortran** and **pascal** declarations.

1. Whenever a **fortran** or **pascal** keyword is used in a declaration, the types of the arguments must be declared with an argument type list. (Ordinarily the argument type list is optional.)
2. The **fortran** and **pascal** keywords modify the item immediately to the right in a declaration.
3. The special **near** and **far** keywords can be used with the **fortran** and **pascal** keywords in declarations. The sequences **far fortran** and **fortran far** are equivalent.

Complex declarators are allowed in **pascal** and **fortran** declarations, just as in C function declarations. With the addition of the **near** and **far** modifiers, these declarations can become quite long. The following

examples illustrate the syntax of **pascal** and **fortran** declarations. For more on complex declarators, see the *Microsoft C Language Reference*.

Examples

```
1.  short pascal thing(short, short);

2.  long (pascal *thing)(void);

3.  short (pascal *(*thing)(void))(long);

4.  short near pascal thing(short);

5.  short pascal near thing(short);
```

Example 1 declares *thing* to be a Pascal routine taking two **short** arguments and returning a **short** value.

In Example 2, *thing* is declared as a pointer to a Pascal routine that takes no arguments and returns a **long** value.

Example 3 declares *thing* to be a pointer to a C function taking no arguments that returns a pointer to a Pascal routine. The Pascal routine is declared to take one **long** argument and return a **short** value.

Examples 4 and 5 are equivalent. Both declare *thing* to be a **near** Pascal routine. The routine takes one **short** argument and returns a **short** value.

User's Guide Appendices

Appendix A
ASCII Character Codes

Dec	Oct	Hex	Chr	Dec	Oct	Hex	Chr
000	000	00H	NUL	032	040	20H	SP
001	001	01H	SOH	033	041	21H	!
002	002	02H	STX	034	042	22H	"
003	003	03H	ETX	035	043	23H	#
004	004	04H	EOT	036	044	24H	$
005	005	05H	ENQ	037	045	25H	%
006	006	06H	ACK	038	046	26H	&
007	007	07H	BEL	039	047	27H	'
008	010	08H	BS	040	050	28H	(
009	011	09H	HT	041	051	29H	)
010	012	0AH	LF	042	052	2AH	*
011	013	0BH	VT	043	053	2BH	+
012	014	0CH	FF	044	054	2CH	,
013	015	0DH	CR	045	055	2DH	-
014	016	0EH	SO	046	056	2EH	.
015	017	0FH	SI	047	057	2FH	/
016	020	10H	DLE	048	060	30H	0
017	021	11H	DC1	049	061	31H	1
018	022	12H	DC2	050	062	32H	2
019	023	13H	DC3	051	063	33H	3
020	024	14H	DC4	052	064	34H	4
021	025	15H	NAK	053	065	35H	5
022	026	16H	SYN	054	066	36H	6
023	027	17H	ETB	055	067	37H	7
024	030	18H	CAN	056	070	38H	8
025	031	19H	EM	057	071	39H	9
026	032	1AH	SUB	058	072	3AH	:
027	033	1BH	ESC	059	073	3BH	;
028	034	1CH	FS	060	074	3CH	<
029	035	1DH	GS	061	075	3DH	=
030	036	1EH	RS	062	076	3EH	>
031	037	1FH	US	063	077	3FH	?

Dec=Decimal, Oct=Octal, Hex=Hexadecimal(H), Chr=Character
LF=Line Feed, FF=Form Feed, CR=Carriage Return
DEL=Rubout

Dec	Oct	Hex	Chr	Dec	Oct	Hex	Chr
064	100	40H	@	096	140	60H	`
065	101	41H	A	097	141	61H	a
066	102	42H	B	098	142	62H	b
067	103	43H	C	099	143	63H	c
068	104	44H	D	100	144	64H	d
069	105	45H	E	101	145	65H	e
070	106	46H	F	102	146	66H	f
071	107	47H	G	103	147	67H	g
072	110	48H	H	104	150	68H	h
073	111	49H	I	105	151	69H	i
074	112	4AH	J	106	152	6AH	j
075	113	4BH	K	107	153	6BH	k
076	114	4CH	L	108	154	6CH	l
077	115	4DH	M	109	155	6DH	m
078	116	4EH	N	110	156	6EH	n
079	117	4FH	O	111	157	6FH	o
080	120	50H	P	112	160	70H	p
081	121	51H	Q	113	161	71H	q
082	122	52H	R	114	162	72H	r
083	123	53H	S	115	163	73H	s
084	124	54H	T	116	164	74H	t
085	125	55H	U	117	165	75H	u
086	126	56H	V	118	166	76H	v
087	127	57H	W	119	167	77H	w
088	130	58H	X	120	170	78H	x
089	131	59H	Y	121	171	79H	y
090	132	5AH	Z	122	172	7AH	z
091	133	5BH	[	123	173	7BH	{
092	134	5CH	\	124	174	7CH	|
093	135	5DH	]	125	175	7DH	}
094	136	5EH	^	126	176	7EH	~
095	137	5FH	_	127	177	7FH	DEL

Dec=Decimal, Oct=Octal, Hex=Hexadecimal(H), Chr=Character
LF=Line Feed, FF=Form Feed, CR=Carriage Return
DEL=Rubout

Appendix B
Command Summary

B.1 Introduction

This appendix summarizes the commands and options available with MSC and the following Microsoft utilities: LINK, LIB, EXEPACK, and EXEMOD.

B.2 Compiler Summary

The compiler is invoked with the MSC command. Type MSC to be prompted for responses or use the command line method to give information to MSC. If you don't give MSC all the information it needs on the command line, it will prompt you for the remaining responses.

Options can appear anywhere a space can appear. The options available with MSC are summarized below.

MSC uses three environment variables to locate the files it needs. Before invoking MSC, use the MS-DOS commands PATH and SET to assign a pathname or pathnames to the following variables.

PATH	Executable compiler files
INCLUDE	Include files
TMP	Temporary files

B.2.1 MSC Options

The following is a complete list of MSC options in alphabetical order. The hyphen character (-) can be used in place of the forward slash (/) to introduce the option if you prefer. Some additional options are available with the CL command; see Section C.4 of Appendix C, "The CL Command."

Option	Task
/A*letter*	Sets the program configuration. The *letter* may be S, M, or L, standing for "small," "medium," and "large" model, respectively.
/A*string*	Sets the program configuration. The *string* consists of three characters in any order, one from each of the following groups. Code Pointer Size: s small; l large Data Pointer Size: n near; f far Segment Setup: d SS equal to DS; u SS not equal to DS, DS loaded for each module; w SS not equal to DS, DS fixed
/C	Preserves comments when preprocessing a file (use only with /E, /P, or /EP).
/D*identifier* [= [*string*]]	Defines *identifier* to the preprocessor. The value is *string* or empty.
/E	Preprocesses the source file, copying the result to the standard output and inserting #line directives.
/EP	Preprocesses the source file, copying the result to the standard output without #line directives.
/Fa [*filename*]	Produces assembly listing.
/Fc [*filename*]	Produces combined source-assembly listing.

/Fl [*filename*]	Produces object listing.
/Fo*filename*	Names the object file.
/FPa	Generates floating-point calls and selects alternate math library.
/FPc	Generates floating-point calls and selects emulator (uses 8087/80287 if one is present).
/FPc87	Generates floating-point calls and selects 8087/80287 library (requires an 8087 or 80287 at run time).
/FPi	Generates in-line 8087/80287 instructions and selects emulator (uses 8087 or 80287 if one is present).
/FPi87	Generates in-line 8087/80287 instructions and selects 8087/80287 library (requires an 8087 or 80287 at run time).
/G0	Generates 8086/8088 instructions.
/G1	Generates 80186/80188 instructions.
/G2	Generates 80286 instructions.
/Gs	Removes calls to stack probe routine.
/Gt [*number*]	Places data items greater than *number* bytes in new segment (256 bytes is the default).
/H*number*	Restricts significant characters of external names to *number* characters.
/I*directory*	Adds *directory* to the top of the list of directories to be searched for include files.
/ND*data-segment-name*	Sets the data segment name.
/NM*module-name*	Sets the module name.
/NT*text-segment-name*	Sets the text segment name.

/O*string*	Controls optimization. The *string* consists of one or more of the following characters. d disable optimization a relax alias checking s favor code size t favor execution time x maximum optimization (equivalent to /Oas /Gs)
/P	Preprocesses the source file and sends output to file with the basename of the source file and the extension ".I".
/u	Removes definitions of all four predefined identifiers.
/U*identifier*	Removes definition of the given predefined identifier.
/V*string*	Copies *string* to the object file.
/w	Suppresses compiler warning messages.
/W*n*	Sets the output level ($n = 0$, 1, 2, or 3) for compiler warning messages.
/X	Ignores the list of "standard places" in the search for include files.
/Zd	Includes line number information in object file.
/Ze	Enables language extensions far, fortran, huge, near, and pascal.
/Zg	Generates function declarations from function definitions and writes declarations to standard output.
/Zl	Removes default library information from object file.
/Zp	Packs structure members.
/Zs	Performs syntax check only.

B.2.2 Standard Memory Models

Table B.1 defines the number of text and data segments for small, medium, and large memory models.

Table B.1

Text and Data Segments in Standard Memory Models

Model	Text Segments	Data Segments
Small	1	1
Medium	1 per module	1
Large	1 per module	1 default data segment[a]

[a] The number of additional data segments depends on the program requirements.

B.2.3 Pointer and Integer Sizes

Table B.2 defines the sizes (in bits) of data pointers, text pointers, and integers (int type) in the three standard memory models.

Table B.2

Pointer and Integer Sizes in Standard Memory Models

Model	Data Pointer	Text Pointer	Integer
Small	16	16	16
Medium	16	32	16
Large	32	32	16

B.2.4 Segment Names

The Table B.3 lists the default text and data segment names in the standard memory models. The default *modulename* is the filename.

Table B.3
Segment Names in Standard Memory Models

Model	Text	Data
Small	_TEXT	_DATA
Medium	*modulename*_TEXT	_DATA
Large	*modulename*_TEXT	_DATA[a]

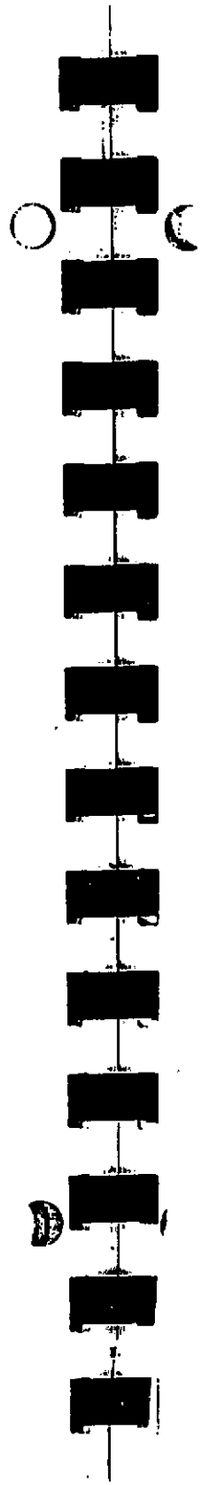
[a] Name of default data segment; other data segments have unique, private names.

B.3 Linker Summary

Microsoft LINK, the object code linker utility, recognizes the command characters and options listed in this section. LINK uses the environment variable LIB to locate library files. Before invoking LINK, use the MS-DOS command SET to assign a pathname or pathnames to the LIB variable.

B.3.1 Linker Command Characters

Character	Task
+	Use the plus sign (+) to separate entries and to extend the current line in response to the "Object Modules" and "Libraries" prompts.
;	To select default responses to the remaining prompts, use a single semicolon (;) followed immediately by a RETURN any time after the first prompt.
CONTROL-C	Type CONTROL-C to terminate the link session at any time.



B.3.2 Linker Options

Options control various linker functions. Options must be typed at the end of a prompt response, regardless of which method is used to start Microsoft LINK. Options may be grouped at the end of any response, or may be scattered at the end of several responses. If more than one option is typed at the end of a response, each option must be preceded by a forward slash (/).

All options may be abbreviated. The only restrictions are that an abbreviation must be sequential from the first letter through the last typed and must uniquely identify the option.

Some linker options take numerical arguments. A numerical argument can be any of the following.

- A decimal number from 0 to 65,535.
- An octal number from 0 to 0177777. A number is interpreted as octal if it starts with a zero. For example, the number "10" is a decimal number, but the number "010" is an octal number, equivalent to 8 in decimal.
- A hexadecimal number from 0 to 0xFFFF. A number is interpreted as hexadecimal if it starts with "0x". For example, "0x10" is a hexadecimal number, equivalent to 16 in decimal.

The linker options, summarized below, are listed in alphabetical order.

Option **Task**

/CPARMAXALLOC:*number*
Sets the cparmaxALLOC field at offset 0x0c in the EXE header to *number*.

/DOSSEG Enforces the following loading order.

1. All segments with a class name ending with CODE.
2. All other segments outside of DGROUP.
3. DGROUP segments, in this order: (a) any segments of class BEGDATA (this class name is reserved for Microsoft use); (b) any segments not of class BEGDATA, BSS or STACK; (c) segments of class BSS; (d) segments of class STACK.

/DSALLOCATE
Tells Microsoft LINK to load all data at the high end of the data segment. Do not use the /DSALLOCATE option with C programs.

/HIGH
Causes the run file to be placed as high as possible in memory. Do not use the /HIGH option with C programs.

/LINENUMBERS
Includes in the list file the line numbers and addresses of the source statements in the input modules.

/MAP
Creates a file listing all public (global) symbols defined in the input modules.

/NODEFAULTLIBRARYSEARCH
Causes default libraries to be ignored.

/NOGROUPASSOCIATION
Provides compatibility with previous versions of LINK. Do not use the /NOGROUPASSOCIATION option with C programs.

/NOIGNORECASE
Causes the linker to distinguish between uppercase and lowercase letters.

/OVERLAYINTERRUPT:*number*
Sets the overlay interrupt number to *number*.

/PAUSE
Causes Microsoft LINK to pause in the link session when the option is encountered.

/SEGMENTS:*number*
Sets the number of segments the linker allows a program to have. The default is 128.

/STACK:*number*
Sets the stack size to *number*, which may be any positive value up to 65,536 bytes. The default for C programs is 2K (2,048 bytes).

B.4 The LIB Utility

The following command characters are recognized by Microsoft LIB, the library manager utility.

Character	Task
+	Appends an object file or library file to the given library.
-	Deletes a module from the library.
-+	Replaces a module by deleting a module and appending an object file with the same name.
*	Extracts a module from the library and saves it in an object file.
-*	Extracts a module from the library and deletes it from the library after saving it in an object file.
;	Uses default responses to remaining prompts.
&	Extends current physical line; repeats command prompt.
CONTROL-C	Terminates library session.

B.5 The EXEPACK Utility

Command

EXEPACK *executable-file output-file*

The EXEPACK utility compresses sequences of identical characters from the given *executable-file* and optimizes the relocation table. The compressed file is written to the output file, and the original file is unmodified.

B.6 The EXEMOD Utility

Command

EXEMOD *executable-file* [/stack *n*] [/min *n*] [/max *n*]

The EXEMOD utility modifies fields in the header according to instructions given on the command line. To display the header fields without modifying them, give the *executable-file* without any options.

Option	Task
/stack *n*	Sets the initial SP (stack pointer) value to *n*, where *n* is a hexadecimal value in bytes. The minimum allocation value is adjusted upward if necessary.
/min *n*	Sets the minimum allocation value to *n*, where *n* is a hexadecimal value in paragraphs. The actual value set may be different from the requested value if adjustments are necessary to accommodate the stack.
/max *n*	Sets the maximum allocation to *n*, where *n* is a hexadecimal value in paragraphs. The maximum allocation value must be greater than or equal to the minimum allocation value.

Appendix C
The CL Command

C.1 Introduction

This appendix summarizes the CL command. The CL command can be used instead of the MSC and LINK commands to invoke the compiler and linker. It is similar to the "cc" interface used on XENIX and UNIX systems; therefore, it may be familiar to some users.

The CL command uses the same four environment variables used by the MSC and LINK commands. They are:

PATH
INCLUDE
TMP
LIB

C.2 Command Syntax and Options

The CL command has the form:

`CL [`*`options...`*`] `*`filename...`* `[/link `*`lib-field`*`]`

where each *option* is a command option, and each *filename* specifies a file to be processed. You can give more than one option or filename, but you must set off each item with one or more spaces. The /link option allows you to pass information to the linker; see Section C.3, "Linking with the CL Command," for a description of the /link option.

Each *filename* must be the name of a C language source file or an object file. If a source file, the filename must include the extension ".c" or ".C". When CL processes the file, it looks at the filename extension to determine whether it should start processing at the compiling or linking stage. Any files ending with ".c" or ".C" are compiled; files with any other extension or no extension are assumed to be object files.

You can use the MS-DOS "wild card" characters (? and *) in filenames on the CL command line. The CL command expands these characters in the same manner that MS-DOS does. See your MS-DOS documentation for more details.

An option consists of a forward slash (/) followed by a combination of one or more letters that have special meaning to CL. You can use a hyphen (-) instead of the slash as an option character if you prefer. You can use any of the options available with the MSC command with CL. Additional options that apply to CL only are given below.

Since you can process more than one file at a time with the CL command, the order in which you give listing options (the /F group of options) is important. The /Fa, /Fc, /Fl, and /Fo options available with the MSC command are also available with CL. In addition, you can use the /Fe option to name the executable file produced in the linking stage and the /Fm option to create a map file. The /F options that can be used with the CL command are summarized in Table C.1. Some additional rules that apply to arguments of the /F options when used with the CL command are given in Table C.2.

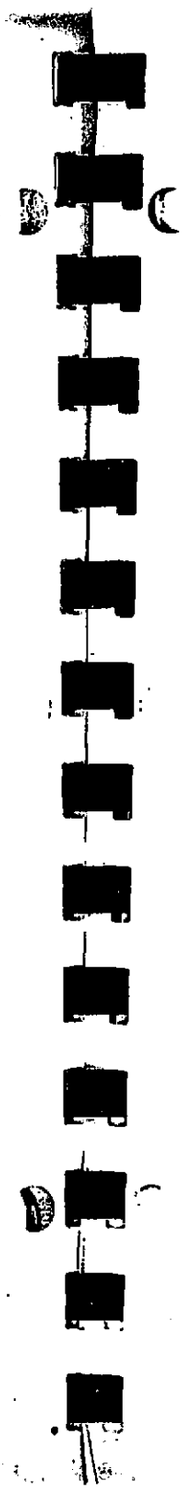


Table C.1
Summary of /F Options

Option	Task	Default Filename[a]	Default Extension
/Fa	Produces assembly listing	Basename of source file plus ".ASM"	.ASM
/Fc	Produces combined source-assembly listing	Basename of source file plus ".COD"	.COD
/Fe	Names the executable file	Basename of first source or object file on command line plus ".EXE"	.EXE
/Fl	Produces object listing	Basename of source file plus ".COD"	.COD
/Fm	Creates map file	Basename of first source or object file on the command line plus ".MAP"	.MAP
/Fo	Names object file	Basename of source file plus ".OBJ"	.OBJ

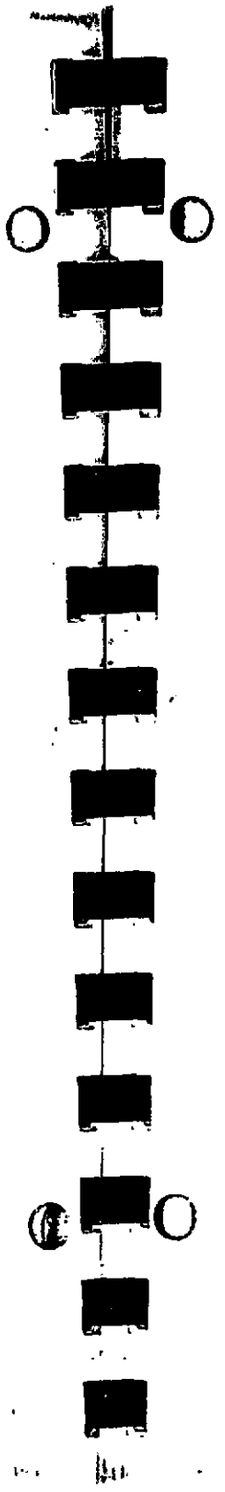
[a] The default filename for the /Fa, /Fc, /Fl, and /Fm options is used when the option is given with no argument or with a directory name as argument. The default filename for the /Fe and /Fo options is used when the option is not given at all, or when a directory name is given as the argument to the option.

Table C.2
Arguments to /F Options

Options	Filename Argument	Pathname Argument	No Argument
/Fa, /Fc, /Fl	Creates a listing for next source file on command line; uses default extension if no extension is supplied.	Creates listings in the given directory for every source file listed after the option on the command line; uses default names.	Creates listings in the default directory for every source file listed after the option on the command line; uses default names.
/Fe	Uses given filename for the executable file; uses default extension if no extension is supplied.	Creates executable file in the given directory; uses default name.	Not applicable; argument is required.
/Fm	Uses given filename for the map file; uses default extension if no extension is supplied.	Creates map file in the given directory; uses default name.	Uses default name.
/Fo	Uses given filename as the object filename for the next source file on command line; uses default extension if no extension is supplied.	Creates object files in the given directory for every source file listed after the option on the command line; uses default names.	Not applicable; argument is required.

Note: No spaces are allowed between the option and the argument (if any) for any of the /F options.

Unlike the MSC command, the CL command invokes the linker as well as the compiler. By default, CL automatically performs linking; you can override this with the /c option, described in Section C.4, "Additional Options." You can also pass your own arguments to the linker, in addition to the default arguments given by CL. This is described in Section C.3, "Linking with the CL Command."



C.3 Linking with the CL Command

By default, the CL command invokes the linker after compiling. You can override the default and cause CL to stop after compiling by giving the /c ("compile only") option.

The CL command uses the response file method of invoking the linker. By default, it builds the following response file.

```
LINK object-list /NOI
basename
NUL;
```

Notice that, by default, the "Libraries" field is not given. The names of the default libraries (the standard C library of the appropriate memory model, plus the appropriate floating-point library as determined by the floating-point option used) are encoded in the object file. The linker searches for the default libraries in the current working directory, then in the directories specified in the LIB environment variable, if any.

The *object-list* is a list of all object files produced in the compiling stage of the CL command, plus any object files specified on the CL command line. The /NOI option tells the linker *not* to ignore case; uppercase and lowercase letters are considered distinct. The *basename* is the name supplied for the executable file; it corresponds to the basename of the first source or object file on the CL command line. (However, you can provide a different name by using the /Fe option.) By default, no map file is produced, since the name NUL is provided in the third field. Notice, however, that the /Fm option can be used in the CL command to override the default and produce a map file.

You can supply your own responses for the "Libraries" field by using the /link option. This option, if included, must be the last item on the CL command line. Any libraries specified in the *library-field* are searched before the default libraries.

The *library-field* can contain one or more of the following.

1. *A pathname.* The linker searches the given pathname for the default libraries *before* searching directories given by the LIB variable.
2. *Additional or alternate library names.* If a pathname is included with the library name, only that pathname is searched. Otherwise,

the linker uses the standard library search path.

3. *The name of a floating-point library or libraries.* Any floating-point calls in your program refer to the given floating-point library instead of the default floating-point library.
4. *Options.* You can give any of the linker options described in Chapter 4, "Linking."

See Chapter 4, "Linking," for more details on default libraries (Sections 4.3.2 and 4.3.3); the library search path (Section 4.3.2); and linker options (Sections 4.3.4 and 4.5).

C.4 Additional Options

In addition to the MSC options summarized in Section B.2.1 of Appendix B, "Command Summary," the CL command recognizes the options listed below. The options are shown with the slash character (/) but can be given with the hyphen (-) character if you prefer.

Option	Task
/c	Creates an object file for each source file on the command line; suppresses linking.
/Fe*progname*	Names the executable program file with *progname*.
/Fm [*mapname*]	Creates a map file.
/link *library-field*	Passes the given *library-field* to the linker as the "Libraries" field.

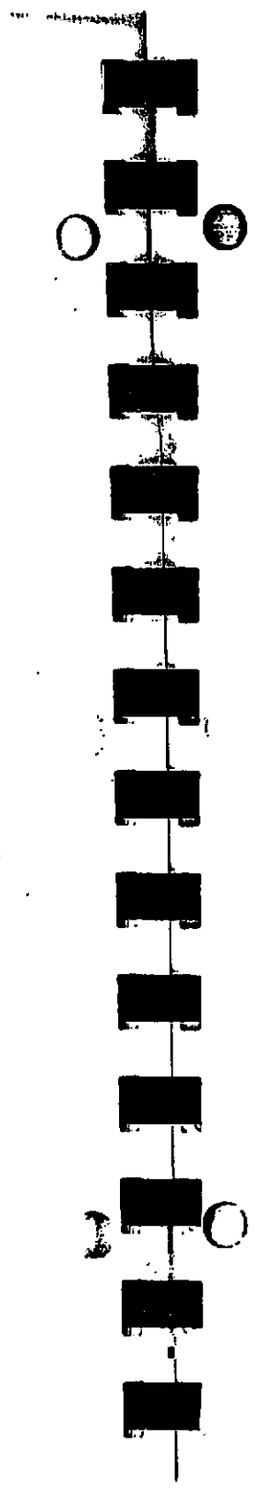
C.5 XENIX-Compatible Options

To provide as much compatibility as possible with XENIX C compilers, the CL command also accepts the options recognized by the "cc" command on XENIX systems. Many of these options are identical to the MSC and CL options given in this manual; others have identical functions but different names. The complete list of XENIX options accepted by the CL command is given in Table C.3.

Table C.3
XENIX Options Accepted by the CL Command

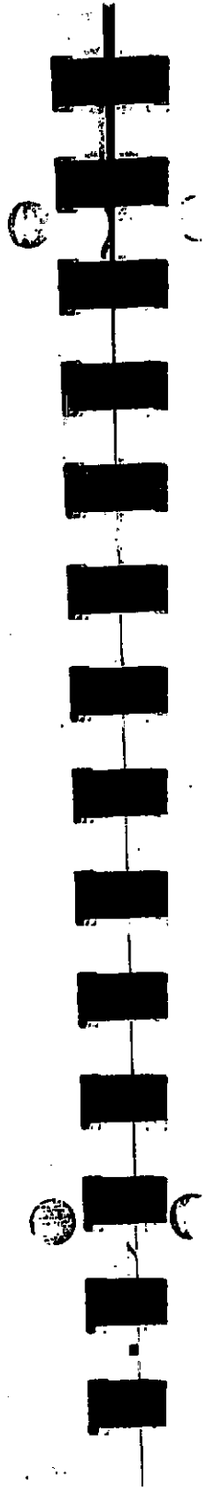
XENIX Option	Task	MSC/CL Option
-c	Creates a linkable object file for each source file.	Same; CL only.
-C	Preserves comments when preprocessing a file (only when -P or -E).	Same.
-D name [=string]	Defines *name* to the preprocessor. The value is *string* or 1.	Same.
-E	Preprocesses each source file, copying the result to the standard output.	Same.
-F num	Sets the size of the program stack. The given size must be a hexadecimal number.	No equivalent in MSC, but /STACK option can be used with LINK or with the /link option of CL.
-I pathname	Adds *pathname* to the list of directories to be searched for #include files.	Same.
-K	Removes stack probes from a program.	-Gs
-L	Creates an object listing file containing assembled object code.	-Fl
-Mstring	Sets the program configuration. The *string* may be any combination of "s" (small model), "m" (middle model), "l" (large model), "e" (enable far and near keywords), "2" (enable 286 code generation), "b" (reverse word order), and "t" (set data threshold for largest item in a segment). The "s", "m", and "l" options are mutually exclusive.	-Me is equivalent to Ze. -M2 is equivalent to G2. -Mt is equivalent to Gt. -Mb has no equivalent. -Ms is equivalent to AS. -Mm is equivalent to AM. -Ml is equivalent to AL.
-nl	Sets the maximum length of external symbols.	-H

-ND name	Sets the data segment name.	Same.
-NM name	Sets the module name.	Same.
-NT name	Sets the text segment name.	Same.
-o filename	Makes *filename* the name of the final executable program.	-Fo
-O	Invokes the object code optimizer.	-Os (default)
-P	Preprocesses source files and sends output to files with the extension ".i".	Same.
-S	Creates an assembly source listing.	-Fa
-V string	Copies *string* to the object file.	Same.
-w	Suppresses compiler warning messages.	Same.
-W num	Sets the output level for compiler warning messages.	Same.
-X	Removes the standard directories from the list of directories to be searched for #include files.	Same.



Appendix D

Converting from Previous Versions of the Compiler



D.1 Introduction

This appendix describes differences between Version 3.0 and earlier versions of the Microsoft C Compiler. The differences fall into three categories: language definition differences, run-time library differences, and assembly language interface differences.

The changes in Version 3.0 are designed to conform to the ANSI standard for the C language (still under development) and to the original language definition. Some features of Versions 2.03 and earlier were not compatible with these standards; such features have been eliminated or changed in Version 3.0. The changes in Version 3.0 also provide greater portability of source code, particularly in the run-time library.

An include file, V2TOV3.H, accompanies your compiler software to help you run your existing Microsoft C programs under Version 3.0.

The following sections describe language, run-time library, and assembly language differences in detail, and outline strategies for converting existing programs.

D.2 Language Definition Differences

This section lists differences in the definition of the C language between Versions 3.0 and earlier versions. The differences are listed by section number from Appendix A of *The C Programming Language*, by Brian Kernighan and Dennis Ritchie, published in 1978 by Prentice-Hall.

Section Number Kernighan and Ritchie	Differences in Versions of Microsoft C
2.1	Comments do not nest in Version 3.0. Versions 2.03 and earlier permitted nesting of comments, unless nesting was deliberately turned off with a command line option. Code containing nested comments will not compile correctly under Version 3.0. In Version 3.0 you can suppress compilation of program sections that contain comments by using a

preprocessor directive (#if).

2.2 Under Version 3.0, identifiers must begin with a letter of the alphabet (uppercase or lowercase) or the underscore character (_). The same characters plus the digits 0-9 are allowed for the rest of the identifier. Versions 2.03 and earlier also allow the dollar sign character ($) in identifiers. This is no longer permitted; code containing dollar signs in identifiers will not compile correctly under Version 3.0.

2.4.3 Multicharacter char constants are allowed under Versions 2.03 and earlier, but are not permitted under Version 3.0. Code containing multicharacter char constants will not compile correctly under Version 3.0.

2.5 Every C string is unique. A string can initialize an array and can be modified at run time. Version 3.0 gives every string separate storage, whether or not a string is identical to another string in the program. Versions 2.03 and earlier detect whether two strings are the same and store only one instance of the string. Existing programs that depend on common storage for identical string literals will not run properly under Version 3.0.

4 Version 3.0 implements the char type as a signed quantity, and provides the unsigned char type to represent unsigned quantities of the same size. Versions 2.03 and earlier treat the char type as unsigned. Programs that depend on the char type being unsigned will not run properly under Version 3.0.

Version 3.0 implements the unsigned long type, a feature not provided in previous versions.

The enumeration type is also provided in Version 3.0. Previous versions did not offer this feature.

7.1 Under Version 3.0, an array or function identifier is considered an address: a constant pointer to the named array or procedure. You can express the address of the array or function simply by giving the identifier. Under Versions 2.03 and earlier, the address-of operator (&) must be applied to an array or function name to express the address of the array or function. Expressions that use this convention will produce unexpected results under Version 3.0.

In Version 3.0, the name of a structure or union variable represents the *value* of that structure or union. In versions 2.03 and earlier, the name of a structure or union represents the *address* of the structure or union. Expressions that depend on this convention will produce unexpected results under Version 3.0.

7.2 In Version 3.0, casting a value to a pointer type produces an lvalue. This was not true in previous versions.

7.6 – 7.7 The relational and equality operators perform the usual arithmetic conversions in Version 3.0. In Versions 2.03 and earlier, the right operand is converted to the type of the left operand.

8.5 Version 3.0 allocates bitfields low order to high order, whereas versions 2.03 and earlier allocate bitfields high order to low order.

11.2 Version 3.0 differs from earlier versions in its treatment of uninitialized variables declared outside of functions (at the external level). A variable declaration at the external level that lacks both a storage class specifier and initializer is treated either as a reference to a definition of the variable elsewhere in the program or, if no definition appears, as a

"communal" variable that is allocated storage by the linker and initialized to zero when the program is loaded. In previous versions, the variable was allocated storage and initialized at compile time.

12

Version 3.0 adds several new features to the C preprocessor. The special constant-expression "`defined(`*identifier*`)`" can follow any `#if` or `#elif` directive. The line "#if defined(ANYTHING)" has the same effect as "#ifdef ANYTHING".

The new directive `#elif` directive allows for "else-if" clauses in `#if` blocks.

In Version 3.0, the pound sign (#) introducing the preprocessor directive can be preceded on the same line by any combination of tabs and spaces. In previous versions, the pound sign had to be the first character of the line.

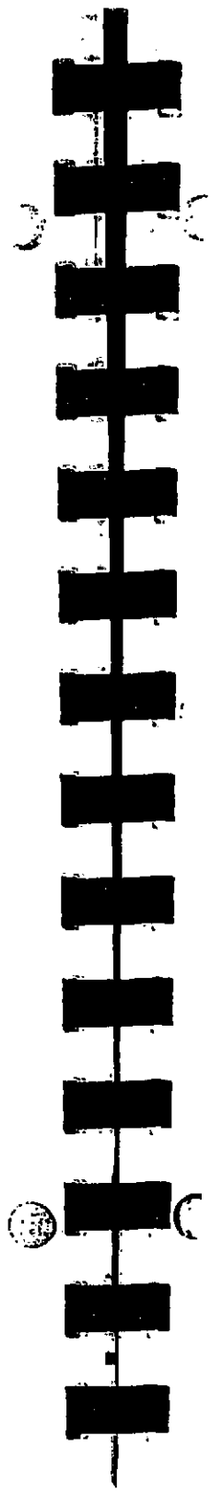
Macro definitions can occupy more than one line in Version 3.0. The newline character is preceded by a backslash (\) to indicate continuation. Earlier versions do not allow continuation.

14.1

Under Version 3.0, structures and unions can be assigned values, passed as arguments to functions, and returned from functions. Earlier versions do not support these features.

14.3

Version 3.0 allows you to check array limits by comparing pointer values against the address just beyond the end of the array. Earlier versions also allow this, but they warn that you have exceeded the array bounds. For example, the following code fragment checks for the bounds of an array.




```
int a[MAX];
proc()
{
        int *p;
        .
        .
        .
        if (p < &a[MAX]) {
        .
        .
        .
        }
}
```

Version 3.0 accepts this construction, while earlier versions generate a warning message.

15

The logical AND and OR operators (&& and | |, respectively) can be used in constant-expressions. These operators were inadvertently omitted from the language reference in previous versions.

Miscellaneous

Version 3.0 makes conservative assumptions about aliases through pointer variables. This means that, in the optimizing stage, the compiler assumes that a memory location referenced indirectly through a pointer variable may also be referenced directly, by another name. The possibility that a program uses aliases (references to the same location by different names) restricts some of the compiler's optimizing procedures. You can use a command line option with Version 3.0 to override the conservative assumptions, allowing the compiler greater freedom in optimization.

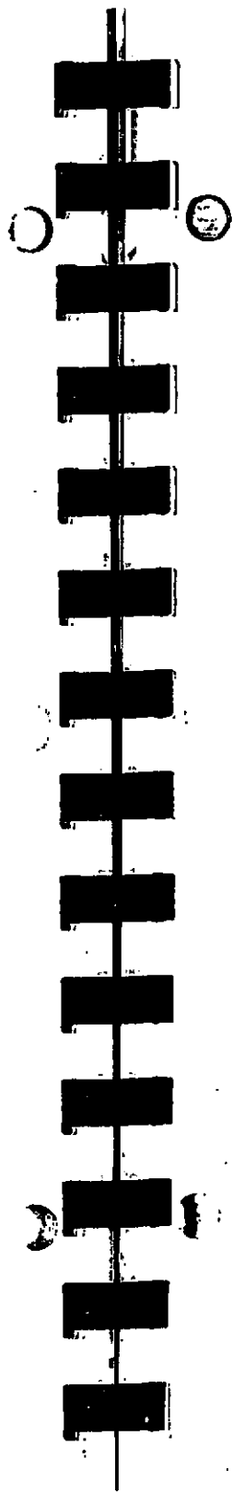
Earlier versions do not make any assumptions about aliases and do not restrict optimization.

D.3 Run-Time Library Differences

Many of the library routines documented in Version 2.03 and earlier will run without change under Microsoft Version 3.0. However, some routines are supported in Version 3.0 under a different function name or syntax. These routines are described in detail below. The changes to the routines are designed to provide greater compatibility with UNIX and XENIX standard libraries.

The include file V2TOV3.H provided with your compiler software allows you to use the modified routines in their original form under Version 3.0. In many cases, you can convert your programs by including V2TOV3.H, without having to change your source code.

Some routines supported under Version 2.03 are not supported at all under Version 3.0. The following routines from Version 2.03 are *not* supported in any form under Version 3.0.



Version 2.03 Routines	Definition
allmem	Level 2 memory allocation
getmem	Level 2 memory allocation
peek	Utility
poke	Utility
rlsmem	Level 2 memory allocation
rbrk	Level 1 memory allocation
repmem	Utility
rstmem	Level 2 memory allocation
sizmem	Level 2 memory allocation
stcarg	String manipulation
stccpy	String manipulation
stcd_i	String manipulation
stch_i	String manipulation
stci_d	String manipulation
stcpam	String manipulation
stcpm	String manipulation
stcu_d	String manipulation
stpblk	String manipulation
stpsym	String manipulation
stptok	String manipulation
stspfp	String manipulation

If your program uses any of the above routines, you must provide your own definition of the routine or alter the code to remove the call to the routine.

The following functions and macros are supported in Version 3.0 in a slightly different manner than in previous versions. The routines are listed under their Version 2.03 names; the sections that follow describe the differences between the versions.

abs	iscsym	movmem	stcis	stpchr
creat	iscsymf	open	stcisn	stscmp
fopen	max	setmem	stclen	
freopen	min	setnbuf	stpbrk	

Use the include file V2TOV3.H, or the appropriate definitions from V2TOV3.H, if your program calls any of the above routines. The remainder of this section summarizes the changes to each of the above routines, and lists the corresponding definition from V2TOV3.H that provides compatibility.

D.3.1 abs

The macro *abs* is defined in the include file V2TOV3.H as follows.

```
#define abs(a,b) (((a) < 0)) ? -(a) : (a))
```

If *abs* is not defined as a macro, it will be interpreted as a call to the standard math library function *abs*.

In previous versions, the *abs*, *min*, and *max* macros were defined in *stdio.h*.

D.3.2 creat

The previous version of this function differs from the Version 3.0 in two ways. In Version 3.0, the permission bits specified in the *mode* argument control access to the created file. For example, if a file is opened for reading, an attempt to write to the file causes an error. Versions 2.03 and earlier do not guarantee this interpretation of the permission bits.

Versions 2.03 and earlier allow the user to create a file in binary mode by giving the O_RAW flag in the *creat* call. The flag can be joined with the permission setting arguments with the bitwise OR operator (|).

Version 3.0 maintains a distinction between the permission setting of a file and the file translation mode. You can specify the permission setting of a file when you create it using the *creat* routine, but you cannot join the translation flag with the file permission setting. The *creat* routine creates a file in the current default mode, whether that is text mode or binary mode. You can change the default mode for a single file with the *setmode* function, or change the default mode for all opened files from text mode to binary mode by linking with BINMODE.OBJ. BINMODE.OBJ is discussed in Section 7.10 of Chapter 7, "Advanced Topics."

You can also control the translation mode of a particular file by giving an appropriate flag when you open the file. See the discussion of *open* later in this section.

The O_RAW flag is renamed to O_BINARY in Version 3.0. V2TOV3.H can be included to define O_RAW as O_BINARY. However, the user is responsible for removing the O_RAW flag from calls to *creat* and for providing appropriate calls to *open* instead.

Example

```
int infile;
        /* VERSIONS 2.03 AND EARLIER */
infile = creat("test.dat", O_RAW);

        /* EQUIVALENT CALL IN VERSION 3.0 */
infile = open ("test.dat",
         (O_CREAT | O_TRUNC | O_BINARY | O_RDWR),
         (S_IREAD | S_IWRITE));
```

This example shows a call to *creat* in Version 2.03 and an equivalent call in Version 3.0. The call to *open* specifies the O_CREAT and O_TRUNC flags, thus accomplishing the same task as the *creat* call. Using *open* rather than *creat* is recommended for new code.

D.3.3 fopen, freopen

In Version 3.0, when a file is opened for appending ("a" or "at" type), all write operations take place at the end of the file. Although the file pointer can be repositioned using *fseek* or *rewind*, the file pointer is always moved back to the end of the file before any write operation is carried out.

In Version 2.03 and earlier, when a file is opened for appending, the file pointer is initially positioned at the end of the file. All write operations take place at the current position of the file pointer; if the file pointer is repositioned (using *fseek* or *rewind*), any write operations will be carried out at the new position.

D.3.4 iscsym, iscsymf

These macros are extensions to the character classification (*ctype*) macros. Version 3.0 does not include the *iscsym* and *iscsymf* macros in the *ctype* set, but you can continue to use them by including the file V2TOV3.H along with the CTYPE.H file.

The V2TOV3.H file defines these macros as follows.

```
#define iscsym(c)       (isalnum(c) || ((c) == '_'))
#define iscsymf(c)      (isalpha(c) || ((c) == '_'))
```

The Version 3.0 definition of *iscsymf* produces a slightly different result than produced by previous versions since Version 3.0 does not allow the dollar sign ($) as a character in identifiers. Note that if the argument *c* has side effects the results of these macros are unpredictable.

D.3.5 max

The macro *max* is defined in the include file V2TOV3.H as follows.

```
#define max(a,b) (((a) > (b)) ? (a) : (b))
```

In previous versions, the *abs*, *min*, and *max* macros were defined in *stdio.h*.

D.3.6 min

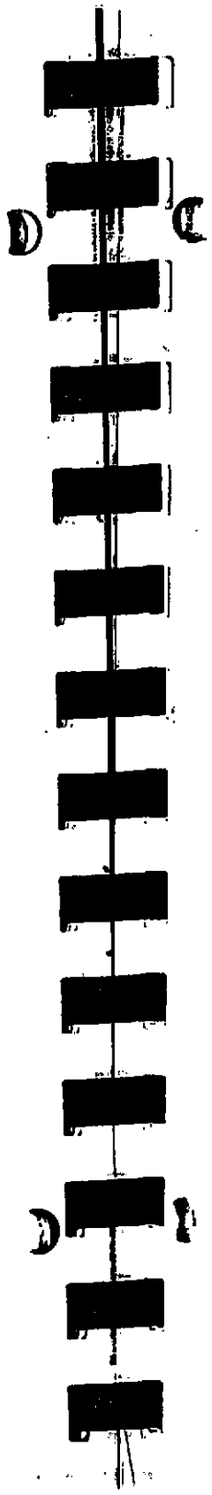
The macro *min* is defined in the include file V2TOV3.H as follows.

```
#define min(a,b) (((a) < (b)) ? (a) : (b))
```

In previous versions, the *abs*, *min*, and *max* macros were defined in *stdio.h*.

D.3.7 movmem

This routine copies a specified number of characters from a given source string to a given destination string. The *movmem* routine handles the transfer correctly in cases where the source and destination strings overlap.



The Version 3.0 routine *memcpy* performs the same task as the *movmem* routine, but the arguments are given in a different order. The include file V2TOV3.H defines *movmem* as follows.

```
#define movmem(s, d, n)          memcpy(d, s, n)
```

D.3.8 open

The *open* routine has the same basic form and function in Version 3.0 as it does in earlier versions, with two exceptions.

1. The flag for binary mode is named O_BINARY instead of O_RAW.
2. The *pmode* argument is required when O_CREAT is specified.

To process a program that uses the O_RAW flag in the call to *open*, include V2TOV3.H or the following definition in your program.

```
#define O_RAW    O_BINARY
```

The Version 3.0 *open* routine takes a third argument. The third argument gives the permission setting of the file; it is optional except when using the O_CREAT flag to create a new file.

D.3.9 setmem

This routine sets a specified number of bytes in a buffer to a given character. The Version 3.0 routine *memset* performs the same task as the *setmem* routine, but the arguments are given in a different order. The include file V2TOV3.H defines *setmem* as follows.

```
#define setmem(p, n, c)          memset(p, c, n)
```

D.3.10 setnbuf

The *setnbuf* routine sets up an empty buffer, and is equivalent to the call

```
setbuf(stream, NULL);
```

Version 3.0 supports the *setnbuf* routine through the following definition in V2TOV3.H.

```
#define setnbuf(stream)  setbuf(stream, NULL)
```

D.3.11 stcis, stcisn, stclen, stpbrk, stpchr, stscmp

These routines are renamed in Version 3.0, but otherwise function exactly the same as in Versions 2.03 and earlier. The names in Version 3.0 are as follows.

Version 2.03 Name	Version 3.0 Name
stcis	strspn
stcisn	strcspn
stclen	strlen
stpbrk	strpbrk
stpchr	strchr
stscmp	strcmp

You can continue using these routines under their Version 2.03 names by including V2TOV3.H or the following definitions in your program.

```
#define stcis(s1, s2)          strspn(s1, s2)
#define stcisn(s1, s2)         strcspn(s1, s2)
#define stclen(s)              strlen(s)
#define stpbrk(s, b)           strpbrk(s, b)
#define stpchr(s, c)           strchr(s, c)
#define stscmp(s1, s2)         strcmp(s1, s2)
```

D.4 Differences in Assembly Language Interface

This section covers the basics of converting assembly language routines written for Versions 2.03 and earlier to run with Version 3.0. Much of the information in this section is also presented in Section 8.1 of Chapter 8, "Interfaces with Other Languages"; this discussion attempts to consolidate the information to make the task of conversion easier. For additional assembly language information not found below, see Section 8.1 of Chapter 8, "Interfaces with Other Languages."

Assembly language routines that are compatible with Versions 2.03 and earlier differ from Version 3.0-compatible routines in five basic areas:

1. Register usage conventions
2. Local variable access (stack setup)
3. Subroutine entry/exit code
4. Global variable naming conventions
5. Segment usage and naming

Each of these areas is discussed in detail below.

D.4.1 Register Usage Conventions

The S and P model programs of Version 2.03 correspond to the small and medium model programs of Version 3.0. In S and P model programs under Version 2.03, ES is always assumed to point to the same segment as SS and DS. However, the "mixed model" programming supported by Version 3.0 allows data in segments outside DS to be accessed. In mixed model programs, the compiler uses ES to reference data outside of the data segment (DS). Thus, ES may not always contain the same value as SS and DS. (Note that SS and DS always contain the same value in Version 3.0 small and medium model programs, unless specifically overridden with the "u" or "w" letter in the /A option.)

Version 3.0 also expects the direction flag of the 8086/8088 processor to be cleared at all times. Therefore, if the assembly routine sets the direction flag, it must clear it (using the CLD instruction) before calling or returning to a C function. This is not required in Version 2.03.

Version 3.0 implements register variables, which were not available in previous versions. The Version 3.0 compiler allows up to two register variables per function. (More than two may be declared, but the extra register requests will be ignored.)

The compiler uses the SI and DI registers to store any register variables. This means that any routine that uses either the SI or DI register must save the register contents upon entry to the subroutine and must restore the original contents before exiting. The compiler takes care of this automatically for C routines, but the user is responsible for providing the necessary instructions in assembly routines. Any assembly routine called from a C function that uses either or both of the SI and DI registers should push the values of the registers onto the local stack (after the stack

is set up) and pop them off the stack before returning to the calling routine.

In the reverse case, where an assembly routine calls a C function, these instructions are not necessary, since the C function automatically saves and restores SI and DI. Since the assembly routine can rely on the values in these registers being preserved across calls to C routines, the registers may not need to be reloaded as often. This assumption may allow more efficient register usage in the assembly routine. See Section D.4.2 for an example.

D.4.2 Stack Setup and Subroutine Entry/Exit Code

All versions of the C compiler use BP as a "frame pointer." Local variables and parameters (also called the "stack frame") are always accessed using offsets from the BP register. However, Version 3.0 differs from earlier versions of the compiler in the entry/exit sequences for subroutines and in the setup of the local stack for subroutine calls.

Figures D.1 and D.2 show the stack frame setups under Version 2.03 and Version 3.0, respectively.

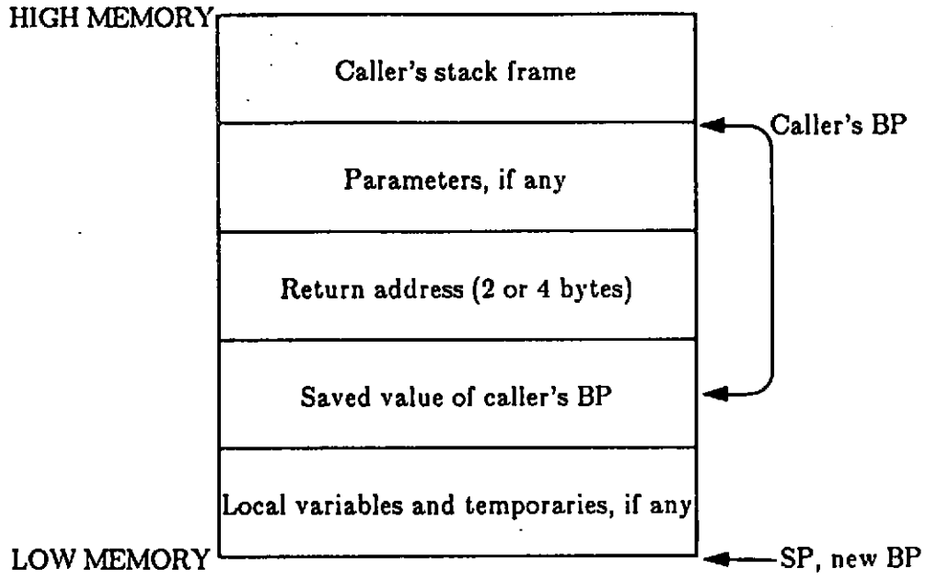


Figure D.1 Version 2.03 Stack Frame Setup

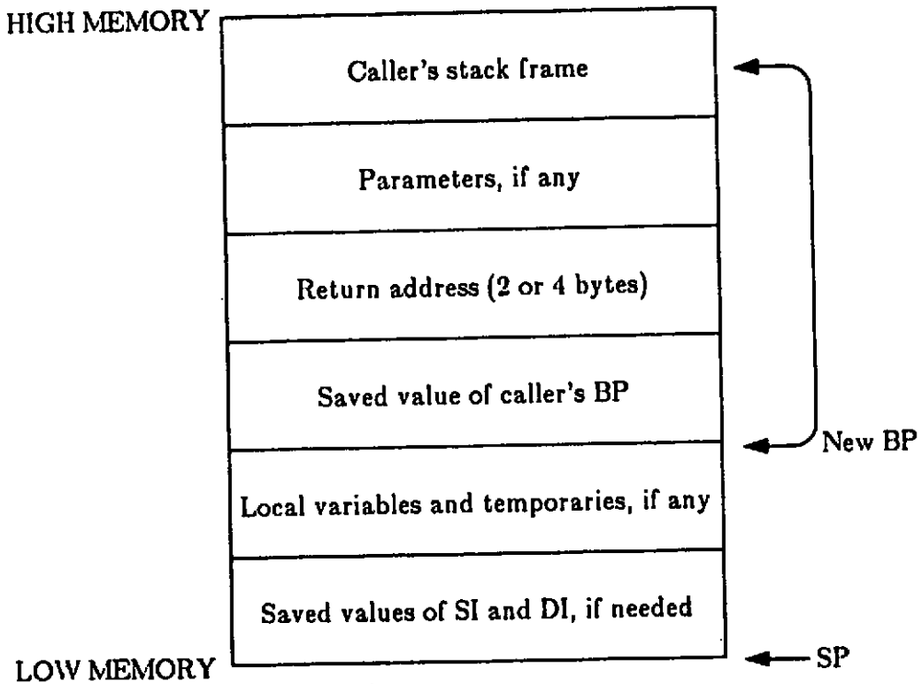


Figure D.2 Version 3.0 Stack Frame Setup

The differences in these two setups are reflected

1. in the entry and exit code sequences for subroutine calls; and
2. in the locations for local variables and parameters.

In Version 3.0, parameter references are positive offsets from BP. Local variable references are negative offsets from BP. The first parameter occurs at either [BP+4] or [BP+6], depending on whether the routine was called using a near call (2-byte address) or a far call (4-byte address).

In Version 2.03, all parameters and local variables are referenced via positive offsets from the BP register. The offset to the first parameter can be calculated as $(n + 4)$ or $(n + 6)$, depending on whether the routine is accessed by a near or far call. The value n is the number of bytes of local storage allocated following the saved caller's frame pointer. Frequently n is zero, in which case parameter offsets in Versions 2.03 and 3.0 are identical.

The versions also differ in the handling of stack checking and stack allocation for local variables other than parameters. In Version 2.03, when stack overflow checking is enabled, the number of bytes of local storage desired (which should be a positive even number in all cases) is subtracted from the stack pointer (SP). The resulting value is compared to a predefined limit value. If the value is less than the limit, a routine named XCOVF is called to report the stack overflow error and terminate the program. If stack checking is disabled, the number of bytes is subtracted from the stack pointer and no overflow checking is performed.

Version 3.0 uses the *__chkstk* routine for stack checking. (The *__chkstk* routine was chosen to help ensure compatibility with XENIX C compilers.) The *__chkstk* routine performs stack checking and produces an error message when appropriate. If stack overflow checking is enabled (the default), the number of bytes of stack space desired is stored in the AX register and the *__chkstk* routine is called. The *__chkstk* routine determines if the request will cause the stack to overflow. If so, *__chkstk* produces an error message to this effect and terminates the program. Otherwise the routine subtracts the given value from the stack pointer and returns. If stack checking is disabled (using the /Gs or /Ox option), the compiler simply subtracts the requested number of bytes from the stack pointer and continues.

Because of the differences in stack setups, exit sequences for Version 3.0 also differ from previous versions. In Version 3.0, the called routine sets SP to the same value as BP. This has the effect of removing local variables from the stack and causing SP to point to the location where the caller's BP was stored. The called routine then pops the caller's saved frame pointer back into BP and returns. The calling routine is responsible for readjusting SP by adding the number of bytes of arguments that were pushed.

In Version 2.03 the called routine first adds the number of bytes of local variables and temporaries to SP, thus causing SP to point to the location of the saved caller's frame pointer. Then the called routine pops the saved frame pointer into BP and returns. After the return, the calling routine must restore the stack pointer by copying the value of BP into SP.

The examples below show typical entry/exit sequences for Versions 2.03 and 3.0. Both examples assume that stack checking is disabled and that 8 bytes is the amount of local variable space required.

Version 2.03

```
ENTRY   push bp         ;save caller's frame pointer (BP)
        sub sp,8        ;allocate local variable space on stack
        mov  bp,sp      ;new frame pointer points to bottom
                        ;  of stack
          .
          .
          .
EXIT    add sp,8        ;deallocate local variable space
        pop  bp         ;restore caller's frame pointer
        ret             ;appropriate to type of call
```

Version 3.0

```
ENTRY   push bp         ;save caller's frame pointer (BP)
        mov  bp,sp      ;frame pointer points to old BP
        sub  sp,8       ;allocate local variable space on stack
        push di         ;required only if routine changes di
        push si         ;required only if routine changes si
          .
          .
          .
EXIT    pop  si         ;required only if si saved on entry
        pop  di         ;required only if di saved on entry
          .
        mov  sp,bp      ;remove local variable space
        pop  bp         ;restore caller's frame pointer
        ret             ;appropriate to type of call
```

Despite the differences listed above, it is not strictly necessary to change the entry/exit sequence of your assembly routines from Version 2.03 to Version 3.0 *unless* your routines attempt to check for stack overflow or use the SI and DI registers. For all other contexts, the setup of the local stack is irrelevant. The parameters are pushed onto the stack in the same way in both versions; the local variable access method is always defined by the routine itself, so any method can be used. The exit sequences of both versions work in essentially the same manner and return to the calling routine with the stack pointer in the same position.

However, changing your code to conform to the Version 3.0 format is still recommended. Debugging will be much easier if your programs consistently use one stack format instead of two.

D.4.3 Global Variable Naming Conventions

In Version 2.03 and earlier, a global name such as XYZ causes a public definition of the name XYZ to be put in the object module. In Version 3.0, for reasons of compatibility with XENIX compilers, an underscore is added to the beginning of the global name when the public definition is put in the object module. For example, the global name XYZ in the source file produces a public definition for the name _XYZ in the object module.

The underscore convention in Version 3.0 means that the name of any assembly routine called from a Version 3.0 program must be defined with a leading underscore in the assembly routine. The C program calls the assembly routine *without* the leading underscore, since the underscore is automatically added by the compiler. For example, the name of an assembly routine might be defined as *_strdo*; the corresponding call in the C program would be "strdo (...)".

Another difference between the compilers arises in the area of case sensitivity. In Version 2.03, external names are not case-sensitive; in Version 3.0, they are. However, when invoking the linker directly (through the LINK command) with a Version 3.0 program, case is ignored by default. You can take advantage of this behavior when linking programs from Version 2.03 and earlier. By contrast, the Version 3.0 compiler control program CL.EXE, which can be used to invoke the linker, automatically tells the linker *not* to ignore case.

Some assemblers are not sensitive to the case of external names, so care must be taken when defining the name of an assembly routine in a C source program.

D.4.4 Segment Usage and Naming

The structure of a Version 3.0 program in memory differs slightly from the Version 2.03 structure. Version 3.0 has the same structure in all memory models. In Version 2.03 there are two different memory layouts, depending on the memory model used. The S and P models in Version 2.03, which correspond to the small and medium models in Version 3.0, use a different layout from Version 3.0. The Version 2.03 D and L models use essentially the same layout as do Version 3.0 programs, with two exceptions:

1. There is no equivalent to the Version 3.0 segment for far data.
2. In Version 2.03 and earlier, SS always points to the stack segment base instead of DS. This is similar to specifying the letter "w" with the memory model (/A) option in Version 3.0.

The Version 2.03 S and P model layout and the Version 3.0 layout are shown in Figures D.3 and D.4, respectively.

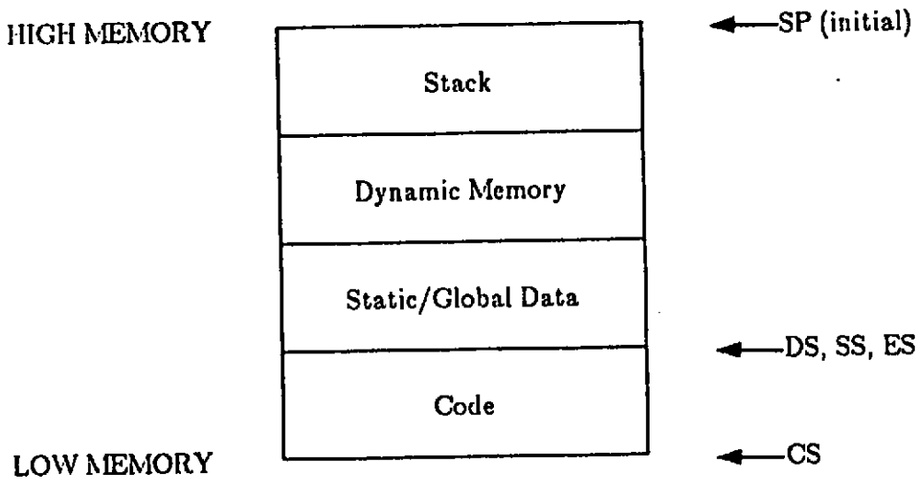


Figure D.3 Version 2.03 S and P Model Layout

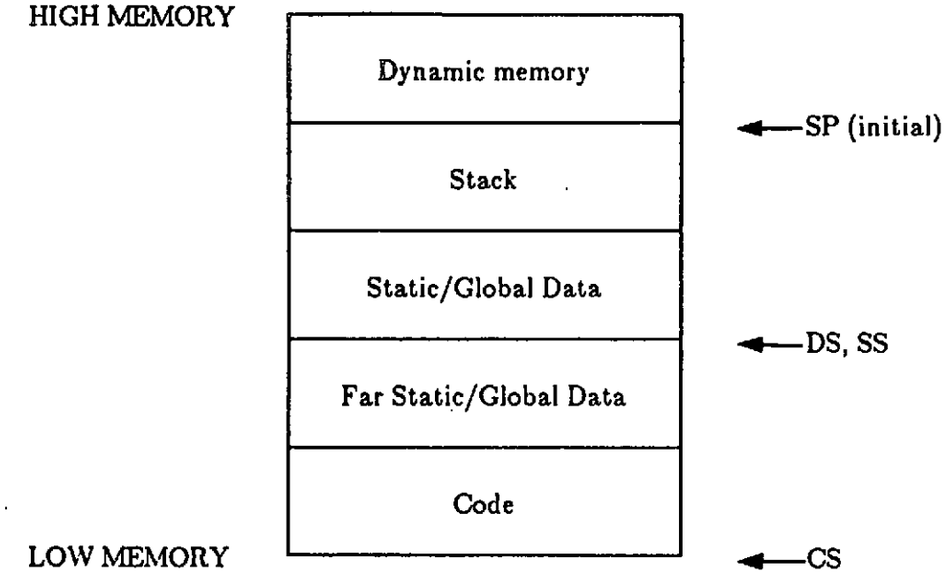


Figure D.4 Version 3.0 Layout

There are two main differences between the above layouts. First, in Version 2.03, the stack resides above dynamic memory. In Version 3.0 it resides below dynamic memory. The Version 3.0 layout means that a program that uses little or no dynamic allocation requires much less space for execution.

The second difference is more important for assembly programmers. In Version 3.0, ES does not necessarily contain the same value as DS. Version 3.0 supports the concept of "far" data in small and medium model programs, while Version 2.03 does not. When far data items are referenced, ES is used to hold the segment value for the far item. Since the compiler has no way of knowing in advance that no far data will occur in the program, it does not rely on ES being the same as DS. Instead, the compiler loads ES whenever it is needed.

Assembly routines written to run with Version 2.03 S and P model programs may have relied on ES being the same as DS. Under Version 3.0, they must load DS into ES to be safe.

Some additional differences between the compilers in the naming of segments and classes are as follows.

- In Version 2.03 the code segments in an S model program are all given class PROG. In Version 3.0 all code segments have class CODE. In Version 3.0 small model, the code segments are all named _TEXT by default; in medium and large model, each compiland forms a segment named *module-name*_TEXT.
- In Version 2.03 S model programs, the code segment is grouped into a group named PGROUP. In Version 3.0 small model programs, the code segment is not grouped.

Both versions use DGROUP to group the DATA and STACK segments in all models.

The general rules and methods for accessing segments in both versions are the same. Usually, the programmer should only be accessing the CODE, _DATA, BSS, c_common, and STACK segments. (Other data segments with class FAR_DATA or FAR_BSS can be useful in some cases.) See Section 8.1.1, "Segment Model," in Chapter 8, "Interfaces with Other Languages," for more information on what kinds of data items are stored in each of the segments.

Appendix E
Error Messages

E.1 Introduction

This appendix lists error messages you may encounter as you develop a program and gives a brief description of the action to take to correct the error. The first section lists run-time errors. Run-time errors are errors you encounter when you execute your program.

The remaining sections describe errors generated by the following programs:

- The Microsoft C Compiler
- The Microsoft LINK utility
- The Microsoft LIB utility
- The EXEPACK utility
- The EXEMOD utility

E.2 Run-Time Error Messages

Run-time error messages fall into four categories:

1. Error messages generated by the run-time library to notify you of serious errors. These messages are listed and described below.
2. Floating-point exceptions generated by the 8087/80287 hardware or the emulator. These exceptions are listed and described in Section E.2.2.
3. Error messages generated by calls in the program to error-handling routines in the C run-time library (the *abort*, *assert*, and *perror* routines.) These routines print an error message to standard error whenever the program calls the given routine. For a description of these routines and the corresponding error messages, see the *Microsoft C Run-Time Library Reference*.
4. Error messages generated by calls to math routines in the C run-time library. On error, the math routines return an error value and some print a message to the standard error. See the *Microsoft C Run-Time Library Reference* for a description of the math routines and corresponding error messages.

E.2.1 Run-Time Library Error Messages

The following messages may be generated at run time when your program has serious errors.

Floating point not loaded
: Your program needs the floating-point library, but the library was not loaded. The error causes the program to be terminated with an exit status of 255. This occurs in two situations:

1. A format string for one of the routines in the *printf* or *scanf* family contains a floating-point format specification and there are no floating-point values or variables in the program. The C compiler attempts to minimize the size of a program by loading floating-point support only when necessary. Floating-point format specifications within format strings are not detected, so the necessary floating-point routines are not loaded. To correct this error use a floating-point argument to correspond to the floating-point format specification. This causes floating-point support to be loaded.
2. *X*LIBFP.LIB or *X*LIBFA.LIB (where *X* is S, M, or L, depending on the memory model) was specified after *X*LIBC.LIB in the linking stage. You must relink the program with the correct library specification.

Null pointer assignment in program
: The contents of the NULL segment have changed in the course of program execution. The NULL segment is a special location in low memory that is not normally used. If the contents of the NULL segment change during a program's execution, it means that the program has written to this area, usually by an inadvertent assignment through a null pointer. Notice that your program can contain null pointers without generating this message; the message appears only when you access a memory location through the null pointer.

  This error does not cause your program to terminate; the error message is printed following the normal termination of the program.

  This message reflects a potentially serious error in your program. Although a program that produces this error may appear to operate correctly, it is likely to cause problems in the future and may fail to run in a different operating environment.

Stack overflow
: Your program has run out of stack space. This can occur when a program uses a large amount of local data or is heavily recursive. The program is terminated with an exit status of 255. To correct the problem, relink using the linker /STACK option to allocate a large stack, or modify the stack information in the executable file header by using the EXEMOD program.

E.2.2 Floating-Point Exceptions

The error messages listed below correspond to exceptions generated by the 8087/80287 hardware. Refer to the Intel documentation for your processor for a detailed discussion of hardware exceptions.

Using C's default 8087/80287 control word settings, the following exceptions are masked and do not occur.

Exception	Default Masked Action
Denormal	Exception masked
Underflow	Result goes to 0.0
Precision	Exception masked

The following errors do not occur with code generated by the Microsoft C Compiler or provided in the Microsoft C Run-Time Library.

Square root

Stack underflow

Unemulated

The floating-point exceptions are listed and described below.

Floating point error: Denormal
: A very small floating-point number was generated, which may no longer be valid due to loss of significance. Denormals are normally masked, causing them to be trapped and operated on.

Floating point error: Divide by 0
: An attempt was made to divide by zero.

Floating point error: Integer overflow
Overflow on assigning a floating-point value to an integer.

Floating point error: Invalid
Invalid operation; usually involves operating on NaN's or infinities.

Floating point error: Overflow
Overflow in floating-point operation.

Floating point error: Precision
Loss of precision occurred in a floating-point operation. This exception is normally masked, since almost any floating-point operation can cause loss of precision.

Floating point error: Stack overflow
A floating-point expression has used too many stack levels on the 8087/80287 or emulator. (Stack overflow exceptions are trapped up to a limit of seven additional levels beyond the eight levels normally supported by the 8087/80287 processor.)

Floating point error: Stack underflow
A floating-point operation resulted in a stack underflow on the 8087/80287 or emulator.

Floating point error: Square root
The operand in a square root operation was negative. (Note: the *sqrt* function in the C run-time library checks the argument before performing the operation and returns an error value if the operand is negative; see the *Microsoft C Run-Time Library Reference* for details on *sqrt*.)

Floating point error: Underflow
Underflow in a floating-point operation. (An underflow is normally masked so that the operation yields the result 0.0.)

Floating point error: Unemulated
An attempt was made to execute an 8087/80287 instruction not supported by the emulator or an invalid 8087/80287 instruction.

E.2.3 Run-Time Limits

Table E.1 summarizes the limits that apply to programs at run time. If your program exceeds one of these limits, an error message will inform you of the problem.

Table E.1
Program Limits at Run Time

Program Item	Description	Limit
Files	Maximum file size	$2^{32}-1$ bytes (4 gigabytes)
	Maximum number of open files (streams)	20[a]
Command Line	Maximum number of characters (including program name)	128
Environment Table	Maximum size	32K

[a] Five streams are opened automatically (*stdin, stdout, stderr, stdaux,* and *stdprn*), leaving 15 available for the program to open.

E.3 Compiler Error Messages

The error messages produced by the C compiler fall into five categories:

1. Warning messages
2. Fatal error messages
3. Compilation error messages
4. Command line error messages
5. Compiler internal error messages

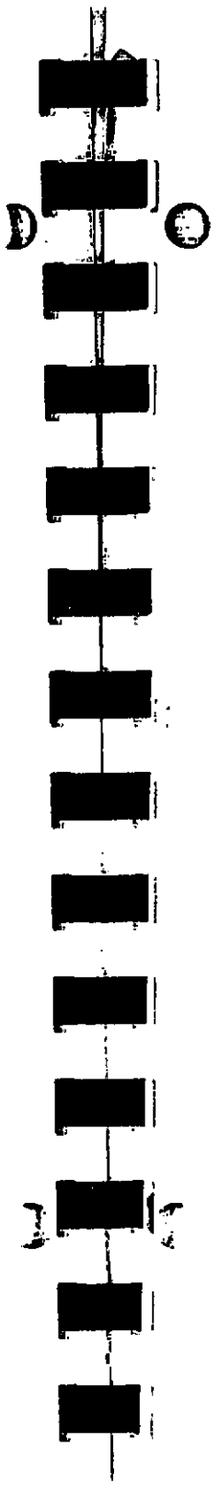
Warning messages are informational only; they do not prevent compilation and linking. You can control the level of warnings generated by the compiler by using the /W option, described in Section 3.9.2 of Chapter 3, "Compiling." The list of warning messages below include a number for each message indicating the minimum level that must be set for the message to appear.

Fatal error messages indicate a severe problem, one that prevents the compiler from processing your program. After printing out a message about the fatal error, the compiler terminates without producing an object file or checking for further errors.

Compilation error messages identify actual program errors. No object file is produced for a source file that has such errors. When the compiler encounters a nonfatal program error, it attempts to recover from the error. If possible, the compiler continues to process the source file and produce error messages. If errors are too numerous or too severe, the compiler terminates processing.

Command line messages give you information about invalid or inconsistent command line options. If possible, the compiler continues operation, printing a warning message to indicate which command line options are in effect and which are disregarded. In some cases, command line errors are fatal, and the compiler terminates processing.

Compiler internal error messages indicate an error on the part of the compiler rather than your program. The following messages are compiler internal error messages. No matter what your source program contains, these messages should not appear. If they do, please report the condition to Microsoft Corporation, using the Software Problem Report at the back of this manual. Although these errors are not the fault of your program,



you will probably want to rearrange your code so that the program can be compiled.

Compiler error (assertion): file *filename*, line *n* source=*filename*
: The compiler performs internal consistency checks during the course of compilation. This message indicates that the consistency check failed. In this message, the first *filename* identifies the compiler file producing the error; the line number *n* refers to that file. The second *filename* gives the name of the source file being compiled.

Compiler error (code generation)
: The compiler could not generate code for an expression. Usually this occurs with a complex expression; try rearranging the expression.

Compiler error (internal)
: The compiler's internal consistency check failed, and the compiler cannot continue.

Fatal(assertion count exceeds 5; stopping compilation)
: When this message appears, more than five assertion errors have accumulated, and the compiler cannot continue operation.

Error messages in the warning, fatal, and compilation error message categories have the same basic form

filename (*linenumber*) : *message*

where *filename* is the name of the source file being compiled, *linenumber* identifies the line of the file containing the error, and *message* is a self-explanatory description of the error or warning. For warning messages and fatal errors, the word "warning" or "fatal" appears at the beginning of the message, followed by a colon.

Command line error messages simply give a message about the command line, so they do not contain references to line numbers and filenames.

The messages for each category are listed below in alphabetical order, along with a brief explanation of each error. To look up an error message, first determine the message category. If the message pertains to the command line, look in the command line section. If the message begins with the word "warning" or "fatal," look in the corresponding section. Otherwise, the message is a compilation error message, and you can find it in that section.

Within each section the error messages are listed alphabetically. Disregard opening single quotes (') when looking up error messages alphabetically.

An error message may begin with an operator or with an item from your program, such as an identifier. In these cases look up the error message under the first complete word following the symbol or program item in the message. For example, to find the description of the message

```
warning : '=' : illegal pointer combination
```

look in the "warning" messages section under "illegal".

If the message begins with a keyword or preprocessor directive, such as `typedef` or `#define`, look up the message under the keyword or the first letter of the directive. For example, look up the message

```
#include expected a file name
```

under "include" in the error message section.

Section E.3.5, "Compiler Limits," summarizes limits imposed by the Microsoft C Compiler (for example, the maximum size of a macro definition).

E.3.1 Warning Error Messages

The messages listed in this section indicate potential problems but do not hinder compilation and linking. The number in square brackets ([ ]) at the end of each message gives the minimum warning level that must be set for the message to appear.

warning : address of frame variable taken, DS != SS [1]
: You must use a `far` pointer when taking the address of a frame variable in a program with separate data and stack segments.

warning : array's declared subscripts differ [1]
: An array is declared twice with differing sizes. The larger size is used.

warning : at least one void operand [1]
: An expression with type `void` is used as an operand.

warning : bad storage class *specifier* on function *identifier* [1]
: Functions must have `static` or `extern` class; any other storage class specifier is ignored.

warning : bitfield type must be integral [1]
: Bitfields must be declared as `unsigned` integral types. A conversion has been supplied.

warning : bitfield type must be unsigned [1]
: Bitfields must be declared as `unsigned` integral types. A conversion has been supplied.

warning : cast of int expression to far pointer [1]
: A `far` pointer represents a full segmented address. On an 8086/8088 processor, casting an `int` value to a `far` pointer produces an address with a meaningless segment value.

warning : constant too big [1]
: Information is lost because a constant value is too large to be represented in the type to which it is assigned.

warning : conversion lost segment [1]
: The conversion of a `far` pointer (a full segmented address) to a `near` pointer (a segment offset) results in the loss of the segment address.

warning : converting a long address to a short address [1]
: The conversion of a long address (a 32-bit pointer) to a short address (a 16-bit pointer) results in the loss of the segment address.

warning : data conversion [3]
: Two data items in an expression had different types, causing the type of one item to be converted.

warning : declared parameter list differs from definition [1]
: The argument type list given in a function declaration does not agree with the types of the formal parameters given in the function definition.

warning : different enum types [1]
: Two different `enum` types are used in an expression.

warning :'*operator*': different levels of indirection [1]
: An expression involving the specified operator has inconsistent levels of indirection. For example,

```
char **p;
char *q;

p=q;  /* different levels of indirection */
```

warning : different types : parameter *n*
The type of the given parameter in a function call does not agree with the argument type list or the function definition.

warning : first parameter list is longer than the second [1]
A function is declared more than once and the argument type lists in the declarations differ.

warning : *identifier* : formal parameter has bad storage class [1]
Formal parameters must have **auto** or **register** storage class.

warning : formal parameter *identifier* is redefined [1]
The given formal parameter is redefined in the function body, making the corresponding actual argument unavailable in the function.

warning : '*identifier*' : formal parameters ignored [1]
Formal parameters appeared in a function declaration (for example, "extern int *f(a,b,c);"). The formal parameters are ignored.

warning : '*identifier*' : function used as an argument [1]
A formal parameter to a function is declared to be a function, which is illegal. The formal parameter is converted to a function pointer.

warning : function declaration specified variable args [1]
The argument type list in a function declaration ends with a comma, indicating that the function can take a variable number of arguments, but no formal parameters for the function are declared.

warning : function must return a value [2]
A function is expected to return a value unless it is declared as **void**.

warning : function *identifier* too large for post-optimizer [0]
The named function was not optimized because insufficient space was available. To correct this problem, reduce the size of the function by breaking it down into two or more smaller functions.

warning : function was declared with formal arguments [1]
The function was declared to take arguments, but the function definition does not declare formal parameters.

warning : function was declared without formal arguments [1]
The function was declared to take no argument (the argument type list consists of the word **void**) but formal parameters are declared in the function definition or arguments are given in a call to the function.

warning : '*identifier*' : has bad storage class [1]
The specified storage class cannot be used in this context (for example, function parameters cannot be given **extern** class). The default storage class for that context is used in place of the illegal class.

warning : identifier truncated to "*identifier*" [1]
Only the first 31 characters of an identifier are significant.

warning : illegal null char [1]
The single quotes delimiting a character constant must contain one character. For example, the declaration "char a = ' '" is illegal. To represent a null character constant, use an escape sequence (for example, '\0').

warning : '*operator*' : illegal pointer combination [1]
A pointer to a given type is forced to point to an object with a different type.

warning : '*operator*' : illegal with enums [1]
You may not use the given operator with an **enum** value. The **enum** value is converted to **int** type.

warning : *operator* : indirection to different types [1]
The indirection operator (*) is used in an expression to access values of different types.

warning : long/short mismatch in arguments : conversion supplied [1]
An integral type is assigned to an integer of a different size, causing a conversion to take place. For example, a **long** is given where a **short** was declared, etc.

warning : macro *identifier* requires parameters [1]
The given *identifier* was defined as a macro taking one or more arguments, but the *identifier* is used in the program without arguments.

warning : missing close parenthesis after 'defined' [1]
The closing parenthesis is missing from an #if defined phrase.

warning : near/far on *identifier* ignored [1]
The near or far keyword has no effect in the declaration of the given *identifier* and is ignored.

warning : near/far mismatch in arguments: conversion supplied [1]
A pointer is assigned to a pointer with a different size, resulting in the loss of a segment address from a far pointer or the addition of a segment address to a near pointer.

warning : newline in string constant [1]
A newline character is not preceded by an escape character (\) in a string constant.

warning : no function return type [2]
The return type is missing from a function declaration; the default return type will be int.

warning : no return value [2]
A function declared to return a value does not do so.

warning : not enough actual parameters for macro *identifier* [1]
The number of actual arguments specified with an identifier is less than the number of formal parameters given in the macro definition of the identifier.

warning : not enough arguments [1]
The number of arguments specified in a function call is less than the number of parameters specified in the argument type list or in the function definition.

warning : '&' on function/array, ignored [1]
You cannot apply the address-of operator to a function or array identifier.

warning : overflow in constant arithmetic [1]
The result of an operation exceeds 0x7FFFFFFF.

warning : overflow in constant multiplication [1]
The result of an operation exceeds 0x7FFFFFFF.

warning : '*identifier*' : overflows array bounds [1]
Too many initializers are present for the given array. The excess initializers are ignored.

warning : parameter *n* differs
The type of the given parameter does not agree with the corresponding type in the argument type list or with the corresponding formal parameter.

warning : parameter *n*'s type is not in union *identifier*
The declaration of the given parameter specifies a union type, but the parameter's type does not correspond to the type of any of the union members.

warning : pointer mismatch: parameter *n* [1]
The given parameter has a different pointer type than is specified in the argument type list or the function definition.

warning : procedure too large, skipping loop inversion optimization and continuing [0]
Some optimizations for a function are skipped because insufficient space is available for optimization. To correct this problem, reduce the size of the function by breaking it down into two or more smaller functions.

warning : procedure too large, skipping branch sequence optimization and continuing [0]
Some optimizations are skipped because insufficient space is available for optimization. To correct this problem, reduce the size of the function by breaking it down into two or more smaller functions.

warning : procedure too large, skipping cross jump optimization and continuing [0]
Some optimizations for a function are skipped because insufficient space is available for optimization. To correct this problem, reduce the size of the function by breaking it down into two or more smaller functions.

warning : recoverable heap overflow in post optimizer - some optimizations may be missed [0]
Some optimizations are skipped because insufficient space is available for optimization. To correct this problem, reduce the size of the function by breaking it down into two or more smaller functions.

warning : *identifier* : redefinition [1]
The given *identifier* is redefined.

warning : 'register' on '*identifier*' ignored [1]
Only integral and pointer type variables may be given register storage class.

warning : second parameter list is longer than the first [1]
A function is declared more than once, and the argument type lists in the declarations differ.

warning: sizeof returns 0 [2]
The **sizeof** operator is applied to an operand that yields a size of zero.

warning : string too big, leading chars truncated [1]
A string exceeds the compiler limit on string size. To correct this problem, you must break the string down into two or more strings.

warning : strong type mis-match [2]
Two different but compatible types are used: for example, a **typedef** type with a non-**typedef** type, or two different but equivalent **struct** or **union** types.

warning : too many actual parameters for macro *identifier* [1]
The number of actual arguments specified with an identifier is greater than the number of formal parameters given in the macro definition of the identifier.

warning : too many actual parameters [1]
The number of arguments specified in a function call is greater than the number of parameters specified in the argument type list or in the function definition.

warning : type following '*keyword*' is illegal, ignored [1]
An illegal combination occurs (for example, **unsigned float**).

warning : #undef expected an identifier
The name of the identifier whose definition is to be removed must be given with the **#undef** directive.

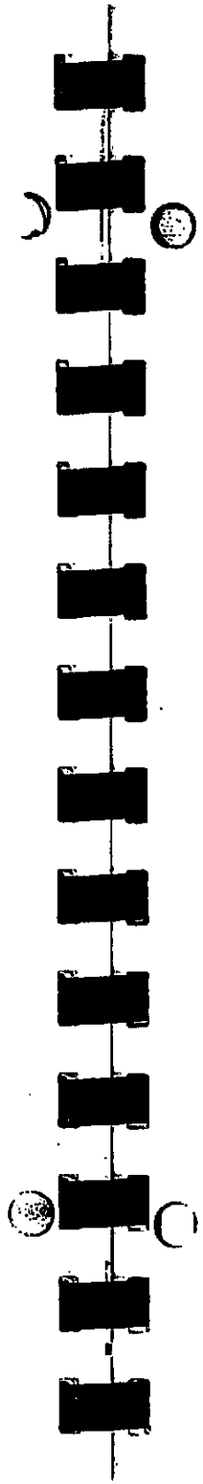
warning : unexpected formal parameter list [1]
A formal parameter list is given in a function declaration and is ignored.

warning : '*identifier*' : unknown size [1]
The size of the named variable is not specified.

warning : unmatched close comment '*/' [1]
A comment is started (with '/*') but is not closed (with '*/').

warning : unnamed struct/union as parameter [1]
The structure or union type being passed as an argument is not named, so the declaration of the formal parameter cannot use the name and must declare the type.

warning : *identifier* uses undefined struct/union *identifier* [2]
The name of a structure or union type is used before the type is defined.



warning : '*identifier*' : void type changed to int [1]
Only functions may be declared to have **void** type.

E.3.2 Fatal Error Messages

The following messages identify fatal errors. The compiler cannot recover from a fatal error; it terminates after printing the error message.

fatal : bad parenthesis nesting
The parentheses in a preprocessor directive are not matched.

fatal : bad preprocessor command '*string*'.
The characters following the number sign (#) do not form a preprocessor directive.

fatal : cannot find *filename*
The given file does not exist or cannot be found. Check to make sure your environment settings are valid and that you have given the correct pathname for the file.

fatal : cannot open *filename*
The given file cannot be opened.

fatal : cannot open listing file *filename*
The filename or pathname given for the listing file is not valid.

fatal : cannot open source file *filename*
The filename or pathname given for the source file is not valid.

fatal : compiler limit : macro expansion too big
The expansion of a macro exceeds the available space.

fatal : compiler limit : possibly a recursively defined macro
The expansion of a macro exceeds the available space. Check to see whether the macro is recursively defined, or if the expanded text is too large.

fatal : DGROUP data allocation exceeds 64K
Large model allocation of variables to the default segment exceeds 64K; use the /Gt option to move items into separate segments.

fatal : error count exceeds *n*; stopping compilation
Errors in the program are too numerous or too severe to allow recovery, and the compiler must terminate.

fatal : error on compiler intermediate file
: Not enough disk space is available for the compiler to create intermediate files used in the compilation process. The space required is approximately two times the size of the source file. To correct this problem, you must make more space available.

fatal : expected '#endif'
: An **#if**, **#ifdef**, or **#ifndef** directive was not terminated with an **#endif** directive.

fatal : #if [n] def expected an identifier
: You must specify an identifier with the **#ifdef** and **#ifndef** directives.

fatal : no input file specified
: You must give at least one source file as input to the compiler.

fatal : not able to execute compiler pass
: One of the compiler executable files cannot be found. Check your environment settings and directory setup.

fatal : parser stack overflow, please simplify your program
: Your program cannot be processed because the space required to parse the program causes a stack overflow in the compiler. To solve this problem, try to simplify your program.

fatal : recursively defined macro *identifier*
: The given identifier is defined recursively.

fatal : too many include files
: Nesting of **#include** directives exceeds the limit of ten levels.

fatal : too many open files, cannot redirect *filename*
: The specified redirection cannot be carried out because the system limit on the number of open files has been reached.

fatal : unexpected '#elif'
: The **#elif** directive is legal only when it appears within an **#if**, **#ifdef**, or **#ifndef** directive.

fatal : unexpected '#else'
: The **#else** directive is legal only when it appears within an **#if**, **#ifdef**, or **#ifndef** directive.

fatal : unexpected '#endif'
: An **#endif** directive appears without a matching **#if**, **#ifdef**, or **#ifndef** directive.

fatal : unexpected EOF
: This message appears when you have insufficient space on the default disk drive for the compiler to create the temporary files it needs. The space required is approximately two times the size of the source file.

E.3.3 Compilation Error Messages

The messages listed below indicate that your program has errors. When the compiler encounters any of the errors listed in this section, it continues parsing the program (if possible) and outputs additional error messages. However, no object file is produced.

'+' : 2 pointers
: Two pointers may not be added.

'*identifier*' : array inits require curly braces
: The braces ({}) around an array initializer are missing.

array of functions
: Arrays of functions are not allowed.

auto allocation exceeds 32K
: The space allocated for the local variables of a function exceeds the limit of 32 kilobytes.

'*class*' : bad storage class
: The given storage *class* cannot be used in this context.

operator : bad left operand
: The left-hand operand of the given *operator* is an illegal value.

bad octal number '*n*'.
: The character *n* is not a valid octal digit.

operator : bad right operand
: The right-hand operand of the given *operator* is an illegal value.

'*identifier*' : base type with near/far not allowed
: Declarations of structure and union members may not use the **near** and **far** keywords.

identifier : can't init arrays in functions
Arrays can only be initialized at the external level.

cannot init struct/union in functions
Structures and unions can only be initialized at the external level.

case expression not constant
Case expressions must be integral constants.

case expression not integral
Case expressions must be integral constants.

case value '*n*' already used
The case value *n* has already been used in this `switch` statement.

cast has illegal formal parameter list
A formal parameter list is given in a type cast expression.

cast of 'void' term to non-void
The `void` type may not be cast to any other type.

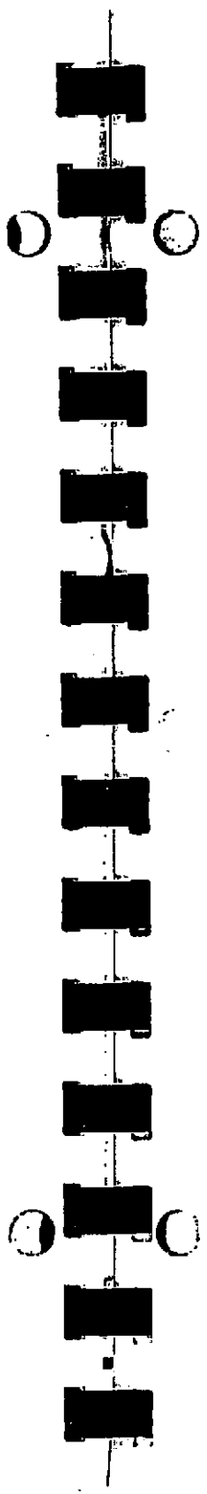
cast to array type is illegal
An object cannot be cast to an array type.

cast to function returning . . . is illegal
An object cannot be cast to a function type.

compiler error
Compiler errors indicate an error on the part of the compiler rather than your program. See Section E.3, "Compiler Error Messages," for instructions on notifying Microsoft about these errors.

Although this message does not indicate errors in your program, you may want to try rearranging your code so that the program will compile. In the error message, the first *filename* identifies the compiler file producing the error; the line number *n* refers to that file. The second *filename* gives the name of the source file being compiled.

compiler limit : initializers too deeply nested
The compiler limit on nesting of initializers has been exceeded. The limit ranges from 10 to 15 levels, depending on the combination of types being initialized. To correct this problem, simplify the data type being initialized to reduce the levels of nesting, or assign initial values in separate statements after the declaration.



compiler limit : struct/union nesting
Nesting of structure and union definitions may not exceed five levels.

constant expression is not integral
The context requires an integral constant expression.

#define syntax
A `#define` directive has a syntax error.

'*identifier*' : definition too big
Macro definitions may not exceed 256 bytes.

'*operator*' : different aggregate types
Pointers to different structure or union types are not allowed with the given *operator*.

divide by 0
The second operand in a division operation (/) evaluates to zero, giving undefined results.

'*identifier*' : enum/struct/union type redefinition
The given *identifier* has already been used for an enumeration, structure, or union tag.

expected '(' to follow '*identifier*'
The context requires parentheses after the function *identifier*.

expected constant expression
The context requires a constant expression.

expected 'defined(id)'
An `#if` directive has a syntax error.

expected exponent value, not '*n*'
The exponent of a floating-point constant is not a valid number.

expected formal parameter list, not a type list
An argument type list appears in a function definition instead of a formal parameter list.

expected preprocessor command, found '*c*'
The character following a number sign (#) is not the first letter of a preprocessor directive.

'*identifier*' : field has indirection
The bitfield is declared as a pointer (*), which is not allowed.

'*identifier*' : field type too small for number of bits
The number of bits specified in the bitfield declaration exceeds the number of bits in the given **unsigned** type.

'*identifier*' : fields only in structs
Only structure types may contain bitfields.

function returns array
A function may not return an array. (It may return a pointer to an array.)

function returns function
A function may not return a function. (It may return a pointer to a function.)

'*identifier*' : functions are illegal members
A function cannot be a member of a structure; use a pointer to a function instead.

'*string*' : ignored
The given text appeared out of context and was ignored.

illegal allocation of *segment-type* segment > 64K
A segment exceeds the limit of 64 kilobytes. The *segment-type* indicates whether the code or data segment exceeds the limit. To correct this problem you must use a larger memory model.

illegal break
A **break** statement is legal only when it appears within a **do**, **for**, **while**, or **switch** statement.

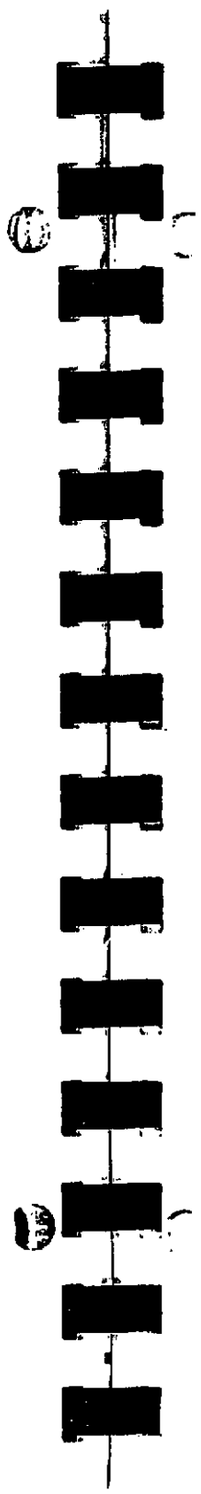
illegal case
The **case** keyword may only appear within a **switch** statement.

illegal cast
A type used in a cast operation is not a legal type.

illegal continue
A **continue** statement is legal only when it appears within a **do**, **for**, or **while** statement.

illegal default
The **default** keyword may only appear within a **switch** statement.

illegal escape sequence
The character(s) after the escape character (\) do not form a valid escape sequence.



illegal expression
An expression is illegal because of a previous error. (The previous error may not have produced an error message.)

'*operator*' : illegal for struct/union
Structure and union type values are not allowed with the given *operator*.

illegal index, indirection not allowed
A subscript was applied to an expression that does not evaluate to a pointer.

illegal indirection.
The indirection operator (*) was applied to a nonpointer value.

illegal initialization
An initialization is illegal because of a previous error. (The previous error may not have produced an error message.)

'*operator*' : illegal pointer combination
Pointers that point to different types cannot be used with the given *operator*.

illegal pointer subtraction.
Only pointers that point to the same type may be subtracted.

illegal sizeof operand
The operand of a **sizeof** expression must be an identifier or a type name.

#include expected a file name
An **#include** directive lacks the mandatory filename specification.

identifier : incompatible types
An expression contains types that are not compatible.

'*identifier*': init of a function
Functions may not be initialized.

identifier is an undefined struct/union
The given *identifier* is declared as a structure or union type that has not been defined.

keyword 'enum' illegal
The **enum** keyword appears in a structure or union declaration, or an **enum** type definition is not formed correctly.

identifier : label redefined
The given *identifier* appears before more than one statement in the same function.

label '*identifier*' was undefined.
The function does not contain a statement labeled with the given *identifier*.

left of '->*identifier*' must have a struct/union type
The expression before the member selection operator '->' does not point to a structure or union type.

left of '.*identifier*' must have a struct/union type
The expression before the member selection operator '.' does not evaluate to a structure or union type.

left of '->' specifies undefined struct/union '*identifier*'
The expression before the member selection operator '->' points to a structure or union type that is not defined.

left of '.' specifies undefined struct/union '*identifier*'
The expression before the member selection operator '.' has a structure or union type that is not defined.

operator : left operand must be lval.
The left operand of the given *operator* must be an lvalue.

#line expected a line number
A #line directive lacks the mandatory line number specification.

'*identifier*' : member of enum redefinition
The given *identifier* has already been used for an enumeration constant, either within the same enumeration type or within another enumeration type with the same visibility.

missing '>'
The closing angle bracket ('>') is missing from an #include directive.

missing name following '<'
An #include directive lacks the mandatory filename specification.

missing open paren after keyword 'defined'
Parentheses must surround the identifier to be checked in an #if directive.

'*identifier*' : missing subscript
To reference an element of an array you must use a subscript.

mod by 0
The second operand in a remainder operation (%) evaluates to zero, giving undefined results.

more than one default
A switch statement contains too many default labels (only one is allowed).

'*operator*' needs lvalue.
The given *operator* must have an lvalue operand.

negative subscript
A value defining an array size was negative.

newline in constant
A newline character in a character or string constant must be preceded by the backslash escape character (\).

no closing single quote
A newline character in a character constant must be preceded by the backslash escape character (\).

non-address expression
An attempt was made to initialize an item that is not an lvalue.

non-constant offset
An initializer uses a non-constant offset.

non-integer switch expression
Switch expressions must be integral.

non-integral field initializer *identifier*
An attempt is made to initialize a bitfield member of a structure with a non-integral value.

non-integral index
Only integral expressions are allowed in array subscripts.

'*identifier*' : not a function
The given *identifier* was not declared as a function, but an attempt was made to use it as a function.

'*identifier*' : not struct/union member
The given *identifier* is used in a context that requires a structure or union member.

'&' on bit field ignored
Bitfields cannot have their address taken.

'&' on constant
Only variables and functions can have their address taken.

'&' on register variable
Register variables cannot have their address taken.

out of heap space
The compiler has run out of dynamic memory space. This usually means that your program has many symbols and complex expressions. To correct the problem, break down the file into several smaller source files.

out of macro actual parameter space
Arguments to preprocessor macros may not exceed 256 bytes.

parameter allocation exceeds 32K
The storage space required for the parameters to a function exceeds the limit of 32 kilobytes.

parameter has type void
Formal parameters and arguments to functions may not have void type.

pointer + non-integer
Only integral values may be added to pointers.

'*operator*' : pointer on left. Needs integral right.
The left operand of the given *operator* is a pointer; the right operand must be an integral value.

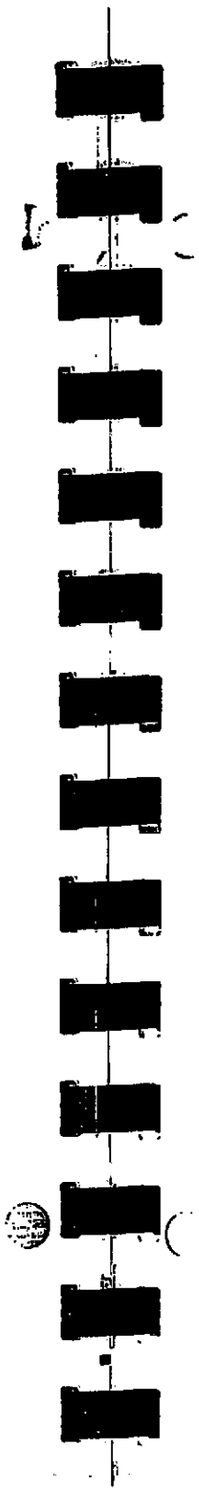
'+' : 2 pointers
Two pointers may not be added.

preprocessor command must start as first non-whitespace
Non-whitespace characters appear before the number sign (#) of a preprocessor directive on the same line.

'*identifier*' : redefinition
The given *identifier* was defined more than once.

redefinition of formal parameter *identifier*
A formal parameter to a function is redeclared within the function body.

'&' requires lvalue
The address-of operator can only be applied to lvalue expressions.



'.' requires struct/union name
The expression before the member selection operator '.' is not the name of a structure or union.

'->' requires struct/union pointer
The expression before the member selection operator '->' is not a pointer to a structure or union.

reuse of macro formal *identifier*
The parameter list in a macro definition contains two occurrences of the same identifier.

'-' : right operand pointer
The right-hand operand in a subtraction operation (-) is a pointer, but the left-hand operand is not.

static procedure '*identifier*' not found.
A forward reference was made to a missing static procedure.

struct/union inits need curly braces
The braces ({}) around a structure or union initializer are missing.

struct/union member needs to be inside a struct/union
Structure and union members must be declared within the structure or union.

struct/union member redefinition
The same identifier was used for more than one structure or union member.

structure/union comparison illegal
You cannot compare two structures or unions. (You can, however, compare individual members of structure and unions.)

subscript on non-array.
A subscript was used on a variable that is not an array.

syntax error
This statement or the preceding statement is not formed correctly.

term does not evaluate to a function
An attempt is made to call a function through an expression that does not evaluate to a function pointer.

'*n*' : too big for char
The number *n* is too large to be represented as a character.

too many chars in constant
: A character constant is limited to a single character or escape sequence. (Multicharacter character constants are not supported.)

too many initializers.
: The number of initializers exceeds the number of objects to be initialized.

typedef specifies different enum
: Two different enumeration types defined with `typedef` are used to declare an item, but the enumeration types are different.

typedef specifies different struct
: Two structure types defined with `typedef` are used to declare an item, but the structure types are different.

typedef specifies different union
: Two union types defined with `typedef` are used to declare an item, but the union types are different.

'typedefs' both define indirection
: Two `typedef` types are used to declare an item and both `typedef` types have indirection. For example, the declaration of *p* in the following example is illegal.

```
typedef int *P_INT;
typedef short *P_SHORT;
/* this declaration is illegal */
P_SHORT P_INT p;
```

'*identifier*' : undefined
: The given *identifier* is not defined.

'*c*' : unexpected in formal list
: The character *c* is misused in a macro definition's list of formal parameters.

'*c*' : unexpected in macro definition
: The character *c* is misused in a macro definition.

unknown character '0x*n*'
: The given hexadecimal number does not correspond to a character.

'*identifier*' : unknown size
: A member of a structure or union has an undefined size.

use of undefined struct/union *identifier*
: The given *identifier* was used to refer to a structure or union type that is not defined.

'void' illegal with all types
: The `void` type cannot be used in operations with other types.

'*expression*' was the use of the struct/union
: An undefined structure or union type variable is used in the given *expression*.

E.3.4 Command Line Messages

The following messages indicate errors on the command line used to invoke the compiler. If possible, the compiler continues operation, printing a warning message. In some cases, command line errors are fatal and the compiler terminates processing.

80286 selected over 8086 for code generation
: Both /G1 and /G2 were specified; /G2 is selected.

a previously defined model specification has been overridden
: Two different memory models are specified; the model specified later is used.

argument list for *name* too big
: The combined length of all arguments on the command line (including the program name) may not exceed 128 bytes.

assembly files are not handled
: You cannot give assembly source files as input to the compiler.

-C ignored (must also specify -P or -E or -EP)
: The -C option takes effect only when you are creating a preprocessed listing using -P, -E, or -EP.

could not execute *name*. Please insert diskette and hit any key.
: One of the compiler passes cannot be found on the current disk. Insert the disk containing the named file and press any key.

ignoring unknown switch *identifier*
: One of the options given on the command line is not recognized and is ignored.

incomplete model specification
: Either one capital letter (S, M, or L) or a string of three lowercase letters must be specified with the /A option.

listing has precedence over assembly output
: Two different listing options were chosen; the assembly listing is not created.

-ND not allowed with -Ad
: The -ND option cannot be used when "d" or "w" is specified with the -A option.

-ND not allowed with -Aw
: The -ND option cannot be used when "d" or "w" is specified with the -A option.

only one floating-point model allowed
: You can only give one of the five floating-point (/FP) options on the command line.

only one of -P/-E/-EP allowed, -P selected
: Each of these options produces a different kind of preprocessed listing; only one can be used at a time.

only one memory model allowed
: You must choose one memory model; you cannot specify more than one.

optimizing for space over time
: This message confirms that the /Os option is used for optimizing.

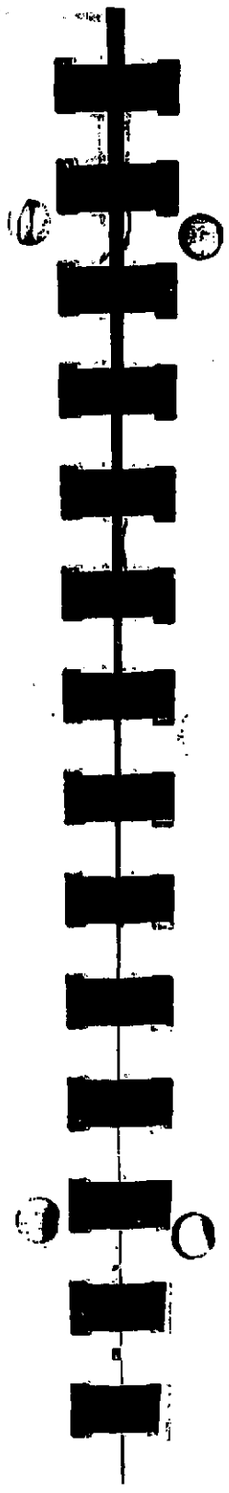
threshold only for far/huge data, ignored
: The threshold option (/Gt) is effective only in large model programs.

too many linker flags on command line
: Too many linker options are specified with the CL command; not all of them can be passed to the linker.

too many symbols predefined with -D
: The limit on command line definitions is normally 16; the /U or /u option can be used to increase the limit to 20.

unknown -A subswitch '*c*'
: One of the letters given with the -A option is not recognized.

unknown floating-point option
: The specified floating-point option (an /FP option) is not one of the five valid options.



unknown -M substring '*c*'
: One of the letters given with the -M option is not recognized. (The -M option is a XENIX-compatible option.)

unknown option (*c*) in *option*
: One of the letters in the given option is not recognized.

E.3.5 Compiler Limits

To operate the Microsoft C Compiler, you must have sufficient disk space available for the compiler to create temporary files used in processing. The space required is approximately two times the size of the source file.

Table E.2 summarizes the limits imposed by the C compiler. If your program exceeds one of these limits, an error message will inform you of the problem.

Table E.2
Limits Imposed by the C Compiler

Program Item	Description	Limit
String Literals	Maximum length of a string, including the terminating null character (\0).	512 bytes
Constants	Maximum size of a constant is determined by its type; see the *Microsoft C Language Reference* for a discussion of constants.	
Identifiers	Maximum length of an identifier.	31 bytes (additional characters are discarded)
Declarations	Maximum level of nesting for structure/union definitions.	5 levels
Preprocessor Directives	Maximum size of a macro definition.	512 bytes
	Maximum number of actual arguments to a macro definition.	8 arguments
	Maximum length of an actual preprocessor argument.	256 bytes
	Maximum level of nesting for #if, #ifdef, and #ifndef directives.	32 levels
	Maximum level of nesting for include files.	10 levels

The compiler does not set explicit limits on the number and complexity of declarations, definitions, and statements in an individual function or in a program. If the compiler encounters a function or program that is too large or too complex to be processed, it produces an error message to that effect.

E.4 Linker Error Messages

All error messages, except for warning messages, cause the link session to end. After you locate and correct a problem, you must relink.

Messages appear in the map file and are displayed on the screen. If you direct the map file to CON, the error messages will not be displayed on the screen.

About to generate .EXE file
Change diskette in drive A: and press <ENTER>.
: This message appears before the ".EXE" is written if the /P switch is given. Insert the disk the ".EXE" file is to be written to into the specified drive (A: for example).

Ambiguous switch error: *switch*
: You did not enter a unique switch name prefix after the switch indicator /, such as LINK /N ALPHA; linker will abort.

Array element size mismatch
: A far communal array is declared with two or more different array element sizes (e.g., declared once as an array of characters and once as an array of floating-point values). Reconcile the definitions and recreate object module.

Attempt to put segment *name* in more than one group in file *filename*
: A segment is declared to be a member of two different groups. Correct the source and recreate the object files.

Bad value for cparMaxAlloc
: The number specified using the /CPARMAXALLOC switch does not lie in the range 1 to 65,535. Try again.

Cannot find library: *name*.lib
Enter new file spec.
: The linker cannot find the given library and is giving you a chance to specify a new file name or a new path specification or both. You should respond to the prompt with a new file name or a new path specification or both.

Cannot nest response files
: You have named a response file within a response file, which is illegal.

Cannot open list file
The directory or disk is full. Make space on the disk or in the directory.

Cannot open response file
You named a response file the linker cannot open. Try again.

Cannot open run file
The directory or disk is full. Make space on the disk or in the directory.

Cannot open temporary file
The directory or disk is full. Make space on the disk or in the directory.

Cannot reopen list file
You did not actually replace the original disk when prompted. Restart the linker.

Common area longer than 65536 bytes
Your program has more than 64 kilobytes of communal variables. NOTE: at the present time, only Microsoft C programs can cause this message to be displayed. Rewrite your program using fewer communal variables or making some of your communal variables far; or recompile your program large model.

Dup record too large
LIDATA record contains more than 512 bytes of data. Most likely, an assembly module contains a struc definition that is very complex, or a series of deeply nested DUP statements (e.g. alpha db 10 dup (11 dup (12 dup (13 dup (...))))). Simplify and reassemble.

Fixup overflow near *address* in segment *name* in *file*.OBJ(*file*) offset *offset*
Some possible causes:

A group is larger than 64 kilobytes.

The user's program contains an intersegment short jump or intersegment short call.

The user has a data item whose name conflicts with that of a subroutine in a library included in the link.

In an assembly language source file, the user has an EXTRN declaration inside the body of a segment. For example,


```
beta    segment public 'code'
extrn   bar:far
alpha   proc    far
        call    bar
        ret
alpha   endp
beta    ends
```

The following construction is preferable.

```
extrn   bar:far
beta    segment public 'code'
alpha   proc    far
        call    bar
        ret
alpha   endp
beta    ends
```

To correct the problem, revise the source and recreate the object file.

Incorrect DOS version, use DOS 2.0 or later
Linker will run only on MS-DOS Version 2.0 or later. Reboot your system with MS-DOS 2.0 or later and try linking again.

Insufficient stack space
There is not enough memory to run the linker.

Interrupt number exceeds 255
A number greater than 255 has been given after the /OVERLAYINTERRUPT switch. Try again with a number in the range 4 to 255.

Invalid numeric switch specification
You made a typographical error entering a value for one of the linker switches (for example, entering a character string for a switch that requires a numeric value). Linker will abort.

Invalid object module
One of the object modules is invalid. If the error persists, contact Microsoft via the Software Problem Report at the back of this manual.

name is not a valid library
The file specified as a library is not a valid library file. Linker will abort.

LEDATA record contains more than 1024 bytes of data.
Please note the translator (compiler or assembler) that produced the incorrect object module and the circumstances under which it was produced; report the information to Microsoft via the Software Problem Report at the back of this manual.

Link failed: status (-1)
LINK.EXE could not be found or could not be executed.

Nested left parentheses
You have made a typing mistake while specifying the contents of an overlay on the command line. Try again.

No object modules specified
You have failed to supply the linker with any object file names. Try again.

Out of space on list file
Disk on which list file is being written is full. Free more space on the disk and try again.

Out of space on run file
Disk on which ".EXE" is being written is full. Free more space on the disk and try again.

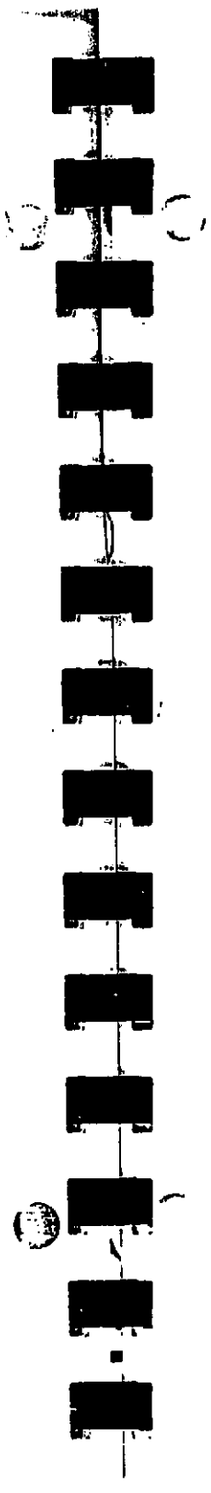
Out of space on scratch file
Disk in default drive is full. Delete some files on that disk, or replace with another disk, and restart the linker.

Overlay manager symbol already defined: *name*
You have defined a symbol name that conflicts with one of the special overlay manager names. Change the offending name and relink.

Please replace original diskette in drive A: and press <ENTER>
This message appears after the ".EXE" has been written if the /P switch is given. Insert the disk with the list file so that it can be reopened.

Relocation table overflow
More than 16,384 long (far) calls, long jumps or other long pointers occur 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 debugging.

Segment limit set too high
The limit on the number of segments allowed was set too high using the /SEGMENTS switch. Linker will abort.



Segment limit too high
There is insufficient memory for the linker to allocate tables to describe the number of segments requested (either the value specified with /SEGMENTS or the default: 128). Either try the link again using /SEGMENTS to select a smaller number of segments (for example, 64, if the default was used previously) or free some memory.

Segment size exceeds 64K
You have a small model program with more than 64 kilobytes of code, or a medium model program with more than 64 kilobytes of data. Try compiling and linking with the medium or large memory model.

Stack size exceeds 65536 bytes
The size specified for the stack using the /STACK switch is more than 65,536 bytes. Try again.

Symbol table overflow
Your program has more than 256 kilobytes of symbolic information (publics, externs, 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
You entered CONTROL-C, causing the linker to terminate.

Too many external symbols in one module
Your object module specified more than the allowed number of external symbols. Break up the module.

Too many group-, segment-, and class-names in one module
Your 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
Your program defines more than nine groups. Reduce the number of groups.

Too many GRPDEFs in one module
Linker encountered more than nine GRPDEFs in a single module. Reduce the number of GRPDEFs or split up the module.

Too many libraries
You tried to link with more than 16 libraries. Combine libraries or link modules that require fewer libraries.

Too many overlays
: Your program defines more than 63 overlays. Reduce the number of overlays.

Too many segments
: Your program has too many segments. Relink using the /SEGMENTS switch with an appropriate number of segments specified.

Too many segments in one module
: Your object module has more than 255 segments. Split the modules or combine segments.

Too many TYPDEFs
: TYPDEFs are records emitted by the compiler to describe communal variables. Create two sources from the old source, dividing the communal variable definitions between them; recompile and relink.

Unexpected end-of-file on library
: The disk containing the library has probably been removed. Try again after inserting the disk containing the library.

Unexpected end-of-file on scratch file
: Disk containing VM.TMP was removed. Restart linker.

Unmatched left parenthesis
: You have made a typing mistake while specifying the contents of an overlay on the command line. Try again.

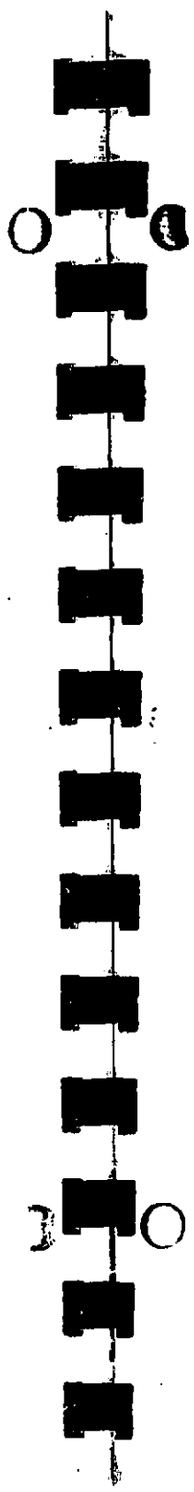
Unmatched right parenthesis
: You have made a typing mistake while specifying the contents of an overlay on the command line. Try again.

Unrecognized switch error:
: You entered an unrecognized character after the switch indicator (/), such as LINK /ABCDEF BETA; linker will abort.

VM.TMP is an illegal file name and has been ignored
: You have used VM.TMP as an object file name. Rename file and link again.

Warning: no stack segment
: Your program contains no segment of combine-type stack.

Warning: too many local symbols
: You asked for a sorted listing of local symbols in the list file, but there are too many symbols to sort. The linker will produce an unsorted listing of the local symbols.



Warning: too many public symbols
: You asked for a sorted listing of public symbols in the list file, but there are too many symbols to sort. The linker will produce an unsorted listing of the public symbols.

E.5 Library Manager Error Messages

The following are Microsoft LIB error messages.

symbol is a multiply defined PUBLIC. Proceed?
: Two modules define the same public symbol. You are asked to confirm the removal of the definition of the old symbol. To correct the source, remove the PUBLIC declaration from one of the object modules and recompile or reassemble. If you respond "no", the library will be left in an indeterminate state.

Allocate error on VM.TMP
: Out of disk space. Make space on the disk or in the directory.

Cannot create extract file
: No room in directory for extract file. Make space on the disk or in the directory.

Cannot create list file
: No room in directory for library file. Make space on the disk or in the directory.

Cannot nest response file
: @*filespec* within response file.

Cannot write library file
: Out of disk space. Make space on the disk or in the directory.

Close error on extract file
: Out of disk space. Make space on the disk or in the directory.

Error: An internal error has occurred
: Contact Microsoft Corporation via the Software Problem Report at the back of this manual.

Fatal Error: Cannot open input file
You mistyped an object filename.

Fatal Error: Module is not in the library
You tried to delete a module that is not in the library.

Input file read error
Bad object module or faulty disk.

Invalid object module/library
Bad object module and/or library.

Library Disk is full
No more room on disk. Make space on the disk or in the directory.

Listing file write error
Out of disk space. Make space on the disk or in the directory.

MS-LIB cannot open VM.TMP
No room for VM.TMP in disk directory. Make space on the disk or in the directory.

No library file specified
No response to "Library name" prompt.

Read error on VM.TMP
Disk not ready for read.

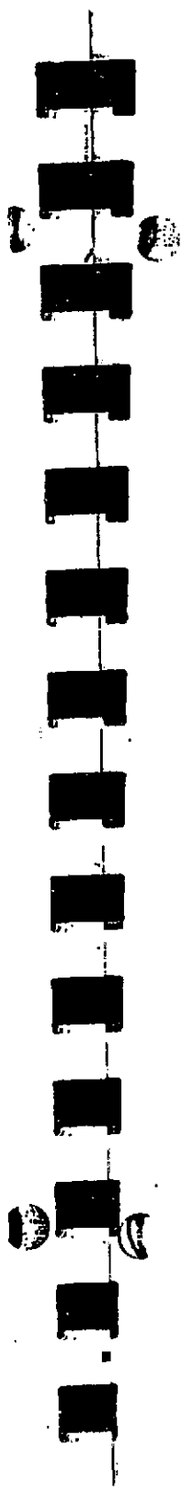
Symbol table capacity exceeded
Too many public symbols (the limit is approximately 30 kilobytes in symbols).

Too many object modules
More than 500 object modules.

Too many public symbols
1,024 public symbols maximum.

Write error on library/extract file
Out of disk space. Make space on the disk or in the directory.

Write error on VM.TMP
Out of disk space. Make space on the disk or in the directory.



E.6 EXEPACK Error Messages

The EXEPACK utility 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 given file cannot be found.

exepack: *filename*: Permission denied
The given file is a read-only file.

exepack: out of memory
The EXEPACK utility does not have enough memory to operate.

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.

You may also encounter MS-DOS error messages if the EXEPACK program cannot read, write to, or create a file.

E.7 EXEMOD Error Messages

The EXEMOD utility 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: *filename*
EXEMOD automatically appends the ".EXE" extension to any filename 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 given 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. (Note: 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. (Note: this is a warning message only; the modification is still performed.)

exemod: *filename*: No such file or directory
The given file cannot be found.

exemod: *filename*: Permission denied
The given file is a read-only file.

usage: exemod file [-/h] [-/stack n] [-/max n] [-/min n]
The EXEMOD command line was not specified properly.

The EXEMOD utility also produces error messages when the file header is not in recognizable ".EXE" format, or if errors occur in reading or writing to a file.

Appendix F
Working with Microsoft Products

F.1 Introduction

This appendix gives an overview of how the Microsoft C Compiler works with other Microsoft languages and tools. In particular, Section F.5 discusses how the compiler works with SYMDEB, the Microsoft Symbolic Debug Utility.

F.2 Microsoft LINK, Microsoft LIB, EXEPACK, and EXEMOD

The programs that accompany the Microsoft C Compiler (the linker, library manager, EXEPACK utility, and EXEMOD utility) work with other Microsoft languages as well as with C. Once you become familiar with the procedures for running these programs, you can use them with other Microsoft languages without having to learn a new program. However, when using these tools, you should always consult the appropriate language documentation to find out what version of the program you have and to learn about any special requirements that may apply when using the tools with a particular language.

The Microsoft Object Code Linker (LINK), which accompanies the C compiler, is the same linker used with all Microsoft compilers, and it offers a wide variety of options to suit the needs of programmers in different languages.

The Microsoft Library Manager (LIB) also accompanies the C compiler. It is used to create and maintain libraries of object files. The library manager accepts MS-DOS object files produced by any Microsoft C compiler. In addition, LIB accepts 286 XENIX archives and Intel-style libraries and allows you to merge these libraries with MS-DOS libraries.

The EXEPACK and EXEMOD utilities (provided with the C compiler) can be used on MS-DOS executable files in any Microsoft language, not just C program files. However, the EXEMOD utility should be used with care, and you should read the discussion of EXEMOD in Section 7.9.2 of Chapter 7, "Advanced Topics," for information about the constraints that apply to executable files in different languages.

F.3 286 XENIX Operating System

The Microsoft C Compiler for MS-DOS shares its design with the C compiler for 286 XENIX systems, and was developed to facilitate portability of C programs between XENIX and MS-DOS systems. If you are interested in writing programs that can be ported to XENIX systems, refer to the *Microsoft C Run-Time Library Reference,* which contains information on portability between the MS-DOS and XENIX C run-time libraries.

F.4 Microsoft FORTRAN and Microsoft Pascal

Versions 3.0 and later of the C compiler allow you to declare and call routines written in Microsoft FORTRAN or Microsoft Pascal. To do this you must have Version 3.3 or later of the Microsoft FORTRAN or Pascal compilers. Section 8.2 of Chapter 8, "Interfaces with Other Languages," describes the syntax for declaring Pascal and FORTRAN routines; however, before attempting to use this feature, you should read the discussion of mixed language programming provided with the Microsoft FORTRAN or Microsoft Pascal Compiler, Version 3.3 or later.

F.5 Microsoft Macro Assembler (MASM) and Symbolic Debug Utility (SYMDEB)

The Microsoft Macro Assembler (MASM) Version 3.0 and later can be used to create assembly language routines to work with C programs. The listing file output by the /Fa option of the C compiler is suitable for input to MASM. See Section 8.1 of Chapter 8, "Interfaces with Other Languages," for a discussion of the assembly language interface to C programs.

The symbolic debug utility (SYMDEB) provided with the Macro Assembler Version 3.0 or later can also be used to debug C programs. SYMDEB can access program locations through addresses, global symbols, or line number references, making it easy to locate and debug specific sections of code. See your Macro Assembler (MASM) documentation for a full description of SYMDEB's capabilities. An introduction to using SYMDEB with C source files is given below.

F.5.1 SYMDEB Procedures

To use SYMDEB with C programs, follow these procedures.

1. When you compile a C source file, use the /Zd and /Od options. The /Zd option produces line numbers in the object file and the /Od option disables optimization so that your code will not be rearranged. (Note: these options are not mandatory for debugging, but they are usually helpful.)
2. When you link, use the /MAP and /LINENUMBERS options to produce a map file that contains line number information.
3. Convert the resulting map file (.MAP) to a symbol file (.SYM) using MAPSYM. MAPSYM is described in the SYMDEB section of the Macro Assembler documentation.
4. Invoke SYMDEB with the symbol file and executable file as arguments (followed by any arguments to the executable file).

When using SYMDEB to debug C programs, you should keep in mind that the C compiler automatically adds a leading underscore to the beginning of every global name. Be sure to include the leading underscore when you specify global symbol names to SYMDEB.

You should also remember that all C programs are linked with a start-up routine (CRT0.OBJ) from the standard C library for the appropriate model. Program execution actually begins with the start-up routine, which then calls the program's "main" function. When you first invoke SYMDEB, the current position will be the beginning of the program, corresponding to the beginning of the start-up routine. Thus, after you invoke SYMDEB, you will probably want to give a command to move to the beginning of the "main" function and start your debugging there.

F.5.2 Sample SYMDEB Session

This sample session with SYMDEB gives examples of some commonly used SYMDEB instructions and shows how to display lines from your source file. For a complete list of SYMDEB options, see your Macro Assembly (MASM) documentation.

A sample program for use in the SYMDEB session is listed below. The sample program contains two functions, *main* and *add,* which are stored in two separate source files named MAIN.C and ADD.C.

```
                /* CONTENTS OF MAIN.C */

double f = 32.5, g = 16.2;

main()
{
        double a;
        double add(double, double);

        a = add(f, g);
}
                /* CONTENTS OF ADD.C */
double add(x, y)
double x, y;
{
        return (x + y);
}
```

Give the following commands to compile and link the files.

```
MSC /Zd /Od ADD.C;
MSC /Zd /Od MAIN.C;

LINK /MAP /LINE ADD.OBJ+MAIN.OBJ;
```

You now have a map file (ADD.MAP) and an executable file (ADD.EXE). To create a symbol file, type

```
MAPSYM ADD.MAP
```

This command creates a symbol file named ADD.SYM that the debugger uses to access global symbols. To start the debugger, type

```
SYMDEB ADD.SYM ADD.EXE
```

This command invokes SYMDEB and passes it the names of the symbol file and executable file for the program to be debugged. (In this case, the program ADD.EXE does not take any command line arguments; if it did, they would be given after ADD.EXE on the SYMDEB command line.) The SYMDEB prompt is the hyphen character (-); it appears whenever SYMDEB is waiting for instructions from you.

You can get a quick demonstration of some of SYMDEB's features by following the steps in this list.

Step	Command	Task
1.	`?`	Displays a listing of SYMDEB commands.
2.	`G _main`	Executes all lines up to the beginning of the "main" function. The start-up code precedes the "main" function.
3.	`R`	Displays the current values of all registers.
4.	`u`	Lists program lines. By default, the listing is an assembly listing, so the assembler instructions that correspond to your program code are displayed. When you type "u" without any further arguments, lines are displayed starting at the current location, which corresponds to the beginning of the *main* function.
5.	`S&`	Changes the default listing type to a combined source and assembly listing. After you give this command, type "u _add" to produce the listing. The global symbol "_add" specifies the starting point for the listing. SYMDEB prompts you for the name of the appropriate source file (the one containing the *add* function). Type ADD.C in response to the prompt, and the source file lines corresponding to the *add* function will be displayed.
6.	`S+`	Changes the default listing type to just source file lines. Type "u _add" after giving this command and the source file lines for the *add* function are displayed. Since the source file, ADD.C, was already specified, you do not have to give the filename again.
7.	`BP _add`	Sets a break point at the beginning of the *add* function.
8.	`G`	Executes the program up to the break point at *add*.
9.	`DL _f`	Displays the value of the global variable *f* as a `long float`. The bytes corresponding to the value of *f* are displayed, and the value is also given in exponential notation.
10.	`Q`	Exits the debugger.

Appendix G
Microsoft LINK Technical Summary

G.1 Introduction

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

You can tell LINK how to link a program's segments by using command line options with the Microsoft C compiler or by using SEGMENT and GROUP directives in the Microsoft Macro Assembler (MASM). See Section 8.1.1 of Chapter 8, "Interfaces with Other Languages," for a discussion of the segment model for C programs and for a listing of class names, align types, and combine types.

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

G.2 Alignment of Segments

LINK uses a segment's alignment type to set the starting address for the segment. The alignment types are BYTE, WORD, PARA, and PAGE. These correspond to starting addresses at byte, word, paragraph, and page boundaries, representing addresses that are multiples of 1, 2, 16, and 256, respectively.

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

G.3 Frame Addresses

LINK computes a starting address for each segment in a program. The starting address is based on a segment's alignment and the size of the segments already copied to the executable file. The address consists of an offset and a "canonical frame number."

The canonical frame number for a particular segment can be determined from the list file produced by LINK as follows. The list file gives the starting address for each segment as a five-digit hexadecimal number. The four most significant digits taken as a four-digit hexadecimal number give the canonical frame number for the segment. The offset is given by the least significant digit of the starting address. The frame number selects a particular paragraph. A paragraph is a set of 16 contiguous bytes such that the address of the first byte in the set is a multiple of 16. The offset is the number of bytes from the first byte in the paragraph to the first byte in the segment.

For BYTE and WORD alignments, the offset may be nonzero. The offset is always zero for PARA and PAGE alignments.

G.4 Order of Segments

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

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


```
DATAX   segment 'DATA'
DATAX   ends

TEXT    segment 'CODE'
TEXT    ends

DATAZ   segment 'DATA'
DATAZ   ends
```

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

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

Note

The Microsoft C Compiler, Versions 3.0 and later, and the Microsoft FORTRAN and Pascal Compilers, Versions 3.3 and later, use the segment ordering specified by the /DOSSEG linker option. This imposes additional constraints on the segment loading order. See the discussion of the /DOSSEG option in Section 4.5.10, "Ordering Segments," of Chapter 4, "Linking," for details.

G.5 Combined Segments

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

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

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

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

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

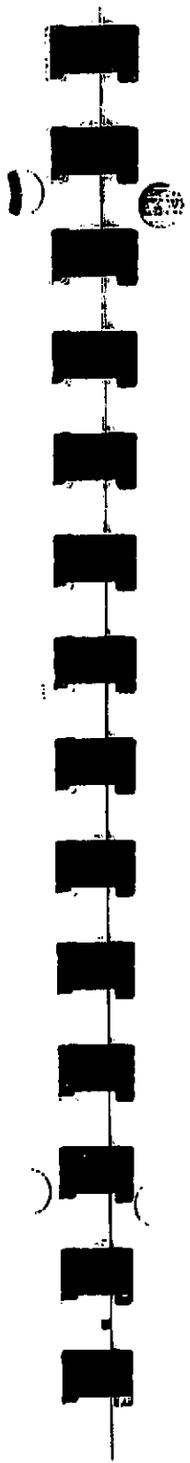
A segment has private type only if no explicit combine type is defined for it in the source file. LINK does not combine private segments.

G.6 Groups

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

Segments in a group do not have to be contiguous, do not have to belong to the same class, and do not have to have the same combine type. The only requirement is that all segments in the group fit within 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, LINK 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, LINK is likely to encounter a fix-up overflow error if this requirement is not met.



G.7 Fix-ups

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

LINK carries out fix-ups 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 fix-up 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. LINK computes a signed, 8-bit number for this reference. It displays an error message if the target instruction belongs to a different segment or group (has a different frame address) or the target is more than 128 bytes distant (either direction).

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

A near segment-relative reference occurs in instructions that 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. LINK displays an error message if the computed offset is greater than 64K or less than zero or if the beginning of the canonical frame of the target is not addressable.

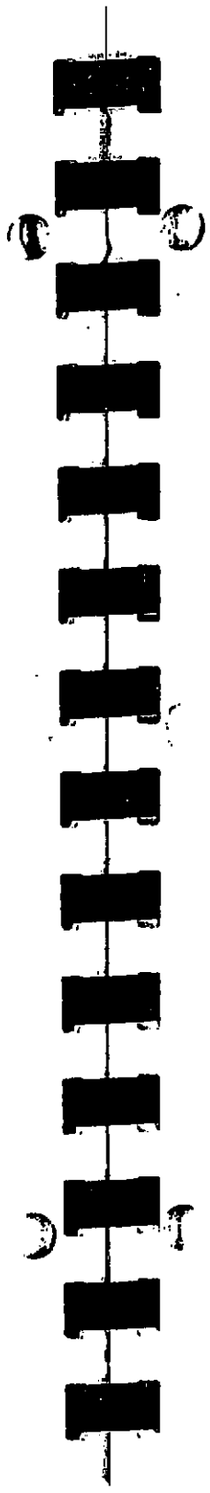
G.8 Controlling the Loading Order

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

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

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

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



Important

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

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

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

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

Example

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

Index

User's Guide Index