

Microsoft® C Compiler

for the MS-DOS® Operating System

User's Guide

Microsoft Corporation

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Corporation. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy this software on magnetic tape, disk, or any other medium for any purpose other than the purchaser's personal use.

© Copyright Microsoft Corporation, 1984, 1985, 1986

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

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

Microsoft, the Microsoft logo, MS, MS-DOS, and XENIX are registered trademarks of Microsoft Corporation. CodeView and The High Performance Software are trademarks of Microsoft Corporation.

IBM is a registered trademark of International Business Machines Corporation.

Intel is a registered trademark of Intel Corporation.

UNIX is a trademark of AT&T Bell Laboratories.

Contents

1 Introduction 1

1.1	Overview	3
1.2	About This Manual	4
1.3	New Features	6
1.4	Notational Conventions	9
1.5	Learning More About C	11
1.6	Reporting Problems	12

2 Getting Started 15

2.1	Introduction	17
2.2	Backing Up Your Disks	17
2.3	Disk Contents	18
2.4	Quick Hard-Disk Setup Procedure	22
2.5	Quick Floppy-Disk Setup Procedure	25
2.6	Understanding the Compiler Software	30
2.7	Setting Up the Environment	34
2.8	Setting Up Your CONFIG.SYS File	38
2.9	Using an 8087 or 80287 Coprocessor	39
2.10	Using an 80186, 80188, or 80286 Processor	40
2.11	Converting Existing C Programs	40
2.12	Organizing Your Software	40
2.13	Practice Session	41
2.14	Using Batch Files	46

3 Compiling 49

3.1	Introduction	51
3.2	Running the Compiler	52
3.3	Listing the Compiler Options	62
3.4	Naming the Object File	63
3.5	Producing Listing Files	64
3.6	Controlling the Preprocessor	70

Contents

3.7	Syntax Checking	76
3.8	Selecting Floating-Point Options	79
3.9	Using 80186, 80188, or 80286 Processors	84
3.10	Understanding Error Messages	85
3.11	Preparing for Debugging	89
3.12	Optimizing	90
3.13	Compiling Large Programs	92
4	Linking	95
4.1	Introduction	97
4.2	Running the Linker	97
4.3	Linking C Program Files	105
4.4	Listing-File Format	107
4.5	Using Overlays	109
4.6	Using Options to Control the Linker	111
4.7	How the Linker Works	123
5	Running C Programs on MS-DOS	129
5.1	Introduction	131
5.2	Passing Command-Line Data to a Program	131
5.3	Returning an Exit Code	136
5.4	Suppressing Null-Pointer Checks	137
6	Managing Libraries	139
6.1	Introduction	141
6.2	Overview of LIB Operation	142
6.3	Running LIB	143
6.4	Library Tasks	150
7	Maintaining Programs with MAKE	157
7.1	Introduction	159
7.2	Using MAKE	159
7.3	Maintaining a Program: an Example	167

8 Working with Memory Models 169

8.1	Introduction	171
8.2	Using the Standard Memory Models	173
8.3	Using the near, far, and huge Keywords	177
8.4	Creating Customized Memory Models	185

9 Advanced Topics 191

9.1	Introduction	193
9.2	Disabling Special Keywords	193
9.3	Packing Structure Members	193
9.4	Restricting Length of External Names	194
9.5	Labeling the Object File	195
9.6	Suppressing Default-Library Selection	195
9.7	Changing the Default char Type	196
9.8	Controlling Stack and Heap Allocation	197
9.9	Controlling Floating-Point Operations	198
9.10	Advanced Optimizing	201
9.11	Controlling the Function-Calling Sequence	203
9.12	Controlling Binary and Text Modes	205
9.13	Setting the Data Threshold	206
9.14	Naming Modules and Segments	207
9.15	Compiling for Windows Applications	209

10 Interfaces with Other Languages 211

10.1	Introduction	213
10.2	Assembly-Language Interface	213
10.3	Mixed-Language Programming	229

Appendixes 267**A ASCII Character Codes 269**

B Command Summary 271

B.1	Introduction	273
B.2	Compiler Summary	273
B.3	Linker Summary	280
B.4	LIB Summary	283
B.5	MAKE Summary	284
B.6	EXEPACK Summary	286
B.7	EXEMOD Summary	287
B.8	SETENV Summary	288

C The CL Command 289

C.1	Introduction	291
C.2	Command Syntax and Options	291
C.3	Linking with the CL Command	294
C.4	Additional Options	296
C.5	XENIX-Compatible Options	297

D Using EXEPACK, EXEMOD, and SETENV 301

D.1	Introduction	303
D.2	The EXEPACK Utility	303
D.3	The EXEMOD Utility	304
D.4	The SETENV Utility	307

E Using Exit Codes 309

E.1	Introduction	311
E.2	Exit Codes with MAKE	311
E.3	Exit Codes with MS-DOS Batch Files	311
E.4	Exit Codes for Programs in the C Compiler Package	312

**F Converting from Previous
Versions of the Compiler 317**

F.1	Introduction	319
F.2	Differences between Versions 3.0 and 4.0	319
F.3	Differences Between Version 4.0 and Versions Prior to 3.0	324

G Writing Portable Programs 345

G.1	Introduction	347
G.2	Program Portability	348
G.3	Machine Hardware	348
G.4	Compiler Differences	354
G.5	Environment Differences	358
G.6	Portability of Data	359
G.7	Byte-Ordering Summary	360

H Error Messages 363

H.1	Introduction	365
H.2	Run-Time Error Messages	365
H.3	Compiler Error Messages	371
H.4	LINK Error Messages	410
H.5	Library-Manager Error Messages	417
H.6	MAKE Error Messages	421
H.7	EXEPACK Error Messages	423
H.8	EXEMOD Error Messages	424
H.9	SETENV Error Messages	425

Index 427

Figures

Figure 10.1	Segment Setup in C Programs	214
Figure F.1	Version 2.03 Stack Frame Setup	338
Figure F.2	Version 3.0 Stack Frame Setup	339
Figure F.3	Version 2.03 Layout for the S and P Models	343
Figure F.4	Layouts for the 3.0 and 4.0 Versions	343

Tables

Table 5.1	Argument Variables	132
Table 8.1	Addressing of Code and Data Declared with near, far, and huge	178
Table 9.1	Using the check_stack Pragma	202
Table 9.2	Segment-Naming Conventions	208
Table 10.1	Segments, Groups, and Classes for Standard Memory Models	217
Table 10.2	C Return Value Conventions	223
Table 10.3	Specifying Calling Conventions	231
Table 10.4	Passing Parameters With C Calling Conventions	232
Table 10.5	Passing Parameters With Pascal Calling Conventions	234
Table 10.6	Passing Parameters With FORTRAN Calling Conventions	234
Table 10.7	Signed 1-Byte Integers	245
Table 10.8	Unsigned 1-Byte Integers	246
Table 10.9	Signed 2-Byte Integers	246
Table 10.10	Unsigned 2-Byte Integers	247
Table 10.11	Signed 4-Byte Integers	247
Table 10.12	Boolean Types	248
Table 10.13	Character Types	248
Table 10.14	Single-Precision Real Numbers	250
Table 10.15	Double-Precision Real Numbers	250
Table 10.16	String and Array Types	252
Table 10.17	Strings	252
Table 10.18	Near Pointers	254
Table 10.19	Far Pointers	254

Table 10.20	Procedure Pointers	255
Table 10.21	Arrays (Lower Bound of Pascal Array Is 0)	258
Table 10.22	Arrays (Lower Bound of Pascal Array Is Nonzero)	258
Table 10.23	Super Array Pointers	259
Table 10.24	Single-Precision Complex Numbers	260
Table 10.25	Double-Precision Complex Numbers	260
Table 10.26	Two-Byte LOGICAL Values	261
Table 10.27	Four-Byte LOGICAL Values	261
Table B.1	Text and Data Segments in Standard Memory Models	278
Table B.2	Pointer and Integer Sizes in Standard Memory Models	279
Table B.3	Segment Names in Standard Memory Models	279
Table C.1	Summary of -F Options	292
Table C.2	Arguments to -F Options	293
Table C.3	XENIX Options Accepted by the CL Command	297
Table G.1	Byte Ordering for Short Types	360
Table G.2	Byte Ordering for Long Types	361
Table H.1	Program Limits at Run Time	371
Table H.2	Limits Imposed by the C Compiler	409

Chapter 1

Introduction

1.1	Overview	3
1.2	About This Manual	4
1.3	New Features	6
1.4	Notational Conventions	9
1.5	Learning More About C	11
1.6	Reporting Problems	12

(

(

(

1.1 Overview

The C language is a powerful general-purpose programming language that can generate 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 five standard memory models (small, medium, compact, large, and huge) 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 transport programs easily from a XENIX development environment to an MS-DOS machine, or vice versa. See the *Microsoft C Compiler Run-Time Library Reference* (included in this package) for more information on the common library for MS-DOS and XENIX.

Note

Since MS-DOS and PC-DOS are essentially the same operating system, Microsoft manuals use the term MS-DOS to include both systems, except in those cases where a utility (such as **SETENV**) is guaranteed to work only under PC-DOS; in those cases, the term PC-DOS is used explicitly.

Compared to other programming languages, C is extremely flexible concerning 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 know how 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 Compiler Language Reference*, included in this package. The *Microsoft C Compiler 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 1.5, "Learning More About C." A brief description of the remaining chapters of the *Microsoft C Compiler User's Guide* is given below.

Chapter 2, "Getting Started," covers installation and organization of 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 chapter also discusses standard memory models (small, medium, compact, large, and huge).

Chapter 4, "Linking," describes the Microsoft Overlay Linker (**LINK**) 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 C Programs on MS-DOS," explains how to run your executable program file, and discusses features specific to the MS-DOS implementation of C. The chapter tells how to pass data from MS-DOS to a program at execution time, and how to return an exit code from your program to MS-DOS.

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

Chapter 7, "Maintaining Programs with MAKE," describes the Microsoft Program Maintenance Utility (**MAKE**). This utility automates the process of maintaining programs by carrying out the tasks needed to update a program after one or more of its component files have been changed.

Chapter 8, "Working with Memory Models," describes methods of managing memory models. These methods are useful for writing large programs that use more than 64K of code or data. This chapter also discusses "mixed-model" programming (combining features from the five standard memory models).

Chapter 9, "Advanced Topics," describes additional command-line options for the experienced programmer and gives the technical information necessary to use them.

Chapter 10, "Interfaces with Other Languages," covers two main topics: the interface between assembly-language routines and C routines, and mixed-language programming using Microsoft's FORTRAN, Pascal, and C compilers.

Appendix A, "ASCII Character Codes," gives the decimal, octal, and hexadecimal equivalents for ASCII (American Standard Code for Information Interchange) characters.

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

Appendix C, "The CL Command," describes an alternative 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, "Using EXEPACK, EXEMOD, and SETENV," tells how to use three special-purpose utilities that are included with the Microsoft C Compiler package.

Appendix E, "Using Exit Codes," lists the exit codes produced by each of the programs in the Microsoft C Compiler package. The chapter also briefly discusses how exit codes are used in **MAKE** description files and in batch files.

Appendix F, "Converting from Previous Versions of the Compiler," summarizes the differences between Version 4.0 of the Microsoft C Compiler and previous versions. This appendix gives instructions for converting programs written for versions prior to 4.0 to the format accepted by Version 4.0.

Appendix G, "Writing Portable Programs," lists some of the C language features that are implementation dependent, and offers suggestions for increasing program portability.

Appendix H, "Error Messages," lists and describes the error messages generated by the C Compiler and by the other programs in the Microsoft C Compiler package. It also lists and explains run-time error messages produced by executable programs written in C.

1.3 New Features

Several useful new features have been added to Version 4.0 of the Microsoft C Compiler. This section summarizes features added since Version 3.0. For information about differences between Version 4.0 and versions prior to 3.0, see Appendix F, "Converting from Previous Versions of the Compiler."

The new features include the following:

Feature	Description
Compact model	The compact memory model allows programs to access more than one segment of data while limiting code to a single segment. A new compact-model library is provided to support this memory model. See Section 3.13, "Compiling Large Programs."
Huge model	The huge memory model allows programs to have multiple code segments, multiple data segments, and single arrays that are larger than

64K. The huge memory model is supported through the large-model library. See Section 3.13, "Compiling Large Programs."

~
huge keyword

The **huge** keyword allows declarations of individual arrays that are larger than 64K. See Section 8.3, "Using the near, far, and Huge Keywords," in Chapter 8, "Working with Memory Models."

CodeView
debugger

The Codeview symbolic debugger is provided with the C compiler. This powerful debugger has a window interface that allows interactive debugging of C programs. See the separate *Microsoft CodeView* manual.

MAKE utility

The Microsoft Program Maintenance Utility, **MAKE**, is provided with the C compiler. See Chapter 7, "Maintaining Programs with **MAKE**."

~
SETENV utility

The **SETENV** utility allows you to enlarge the PC-DOS environment variable table. See Appendix D, "Using EXEPACK, EXEMOD, and **SETENV**."

Source listings

The **MSC** and **CL** commands can produce source listings showing source lines, errors encountered during compilation, and local and global symbol information. A source listing can be produced either with an **MSC** prompt or command line, or with the new **/Fs** option. See Section 3.2.5 for more information.

Numbered errors

Compiler and run-time error messages are now numbered. See Appendix H, "Error Messages."

~
New **MSC** and
CL options

Option Action

/HELP Lists many of the more commonly used options. This option is not case sensitive: any combination of uppercase and lowercase letters is acceptable; for example, **/hELp**.

/Fs	Creates a source-listing file.
/Gc	Causes compiler to use function entry/exit sequence used by the Microsoft FORTRAN and Microsoft Pascal compilers.
/J	Makes the char type unsigned by default.
/Zi	Produces full symbolic debugging information for use with the Code-View symbolic debugger.

See Chapter 3, "Compiling," and Chapter 9, "Advanced Topics," for descriptions of these options.

New keywords

Keyword Description

signed	Usage is similar to unsigned ; used with the /J compiler option
huge	Allows you to create arrays larger than 64K (as well as pointers to those arrays) in any memory model
cdecl	Similar in usage to the keywords fortran and pascal ; useful in conjunction with the /Gc option; enables C function entry/exit sequence and naming convention, thus allowing functions (including standard library functions) to have an arbitrary number of parameters, even in the presence of the /Gc option

pragmas

The **#pragma** directive has been added, in accordance with the developing ANSI C standard. (The **#pragma** directive is discussed in Chapter 8 of the *Microsoft C Compiler Language Reference*.) The only pragma implemented in Version 4.0 is the **check_stack** pragma, discussed in Section 9.10.1, "Removing Stack Probes."

New LINK options	Option	Action
	/HELP	Displays a list of LINK options
	/EXEPACK	Packs executable files during linking
	/CO	Prepares executable files with the symbolic information needed by the <i>CodeView</i> debugger
		See sections 4.6.1, 4.6.3, and 4.6.6 for more information.
Language changes		The C language syntax and semantics have been modified in certain cases to correspond with recent updates to the ANSI standard for the C language. See Appendix F, “Converting from Previous Versions of the Compiler,” and Appendix A of the <i>Microsoft C Compiler Language Reference</i> .
New library routines		A number of library routines have been added, and some existing routines have been modified and enhanced. See Appendix F, “Converting from Previous Versions of the Compiler,” and the <i>Microsoft C Compiler Run-Time Library Reference</i> .

1.4 Notational Conventions

The following notational conventions are used throughout this manual:

Convention	Meaning
Bold	Bold type indicates text that must be typed as shown. Text that is normally shown in bold type includes operators, keywords, library functions, commands, options, and preprocessor directives. Examples are shown below:

+ = #if defined() int
if /Fa fopen
main sizeof

**BOLD
CAPITALS**

Bold capital letters are used for the names of executable files and files provided with the product, environment variables, 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 programs names such as **MSC**, **LINK**, and **LIB**. However, you are not required to use capital letters when you actually enter these commands.

Italics

Italics mark the places in command-line and option specifications and in the text where specific terms appear in an actual command. Consider the following option line:

/W *number*

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

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

Examples

Programming examples are displayed in a special typeface so that they resemble the output on your screen or the output of commonly used computer printers.

User input

Some examples show both program output and user input; in these cases, input is shown in a darker font.

Ellipsis dots

Vertical ellipsis dots are used in program examples to indicate that a portion of the program is omitted. For instance, in the following excerpt, the ellipsis dots between the statements indicate that intervening program lines occur but are not shown:

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

[Double brackets]

Double brackets enclose optional fields in command-line and option specifications. Consider the following option specification:

/D *identifier*[=[*string*]]

The placeholder *identifier* indicates that you must supply an identifier when you use the /D option. The outer brackets indicate that you are not required to supply an equal sign (=) and a string following the identifier. The inner brackets indicate that you are not required to enter a string following the equal sign, but if you do supply a string, you must also supply the equal sign.

Single brackets are used to indicate brackets used by C-language array declarations and subscript expressions. For instance, a[10] is an example of brackets in a C subscript expression.

"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 refer to command-line prompts. For example, LINK prompts you for the name of the object files; this prompt is called the "Object Modules" prompt.

Some C constructs require quotation marks. Quotation marks required by the language have the form "" rather than ". For example, a C string used in an example would be shown in the following form:

"abc"

SMALL CAPITALS

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

1.5 Learning More About C

The 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 or more 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.

Schustack, Steve. *Variations in C*. Bellevue, Washington: Microsoft Press, 1985.

This is by no means an exhaustive list of the books available for learning C; any book's inclusion in this list should not be taken as a recommendation by Microsoft over other books on the same subject.

1.6 Reporting Problems

If you encounter a problem or you feel you have discovered a problem in the software, please provide the following information to help us in locating the problem:

- The compiler version number (from the logo that is printed when you invoke the compiler with **MSC** or **CL**)
- The version of MS-DOS you are running (use the MS-DOS **VER** command)
- Your system configuration (type of machine you are using and its total memory, total free memory at compiler execution time, as well as any other information you think might be useful)
- The command line used in the compilation
- A preprocessed listing of the program (produced with the **/E**, **/P**, or **/EP** option), or if the problem appears to be in the preprocessor, the C source file or files and *all* include files referenced
- Any nonstandard object files or libraries needed to link, in addition to the standard object files or libraries you linked with at the time of the problem

If your program is very large, please try to reduce its size to the smallest possible program still producing the problem.

Use the Software Problem Report at the back of this manual to send this information to Microsoft.

If you have comments or suggestions regarding any of the manuals accompanying this product, please use the Documentation Feedback Card at the back of this manual.



Chapter 2

Getting Started

2.1	Introduction	17
2.2	Backing Up Your Disks	17
2.3	Disk Contents	18
2.4	Quick Hard-Disk Setup Procedure	22
2.5	Quick Floppy-Disk Setup Procedure	25
2.6	Understanding the Compiler Software	30
2.6.1	Executable Files	30
2.6.2	Include Files	31
2.6.3	Library Files	31
2.6.4	Other Files	33
2.7	Setting Up the Environment	34
2.8	Setting Up Your CONFIG.SYS File	38
2.9	Using an 8087 or 80287 Coprocessor	39
2.10	Using an 80186, 80188, or 80286 Processor	40
2.11	Converting Existing C Programs	40
2.12	Organizing Your Software	40
2.13	Practice Session	41
2.14	Using Batch Files	46

10

11

12

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 constitute 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 operating system 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 Overlay Linker (**LINK**). 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.

To get your C compiler up and running, we suggest the following steps:

1. Back up your disks (see Section 2.2).
2. Check the contents of the disks (see Section 2.3).
3. Read the **README.DOC** file to learn about changes and additions made to the compiler after this manual was printed.
4. Use the “Quick Setup Procedure” applicable to your system (floppy or hard disk) to create directories and copy files from the system disks (see Section 2.4 or Section 2.5).

2.2 Backing Up Your Disks

The first thing you should do after you have unwrapped your system disks is make working copies, using the MS-DOS **COPY** command or the **DISKCOPY** utility. 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

File Name	Description
MSC.EXE	Control program for the compiler
C1.EXE	Preprocessor and language parser
C2.EXE	Code generator
C3.EXE	Optimizer, link text emitter, and assembly-listing generator
LINK.EXE	Microsoft Overlay Linker
LIB.EXE	Microsoft Library Manager
EXEPACK.EXE	Microsoft EXE File Compression Utility
EXEMOD.EXE	Microsoft EXE File Header Utility
SETENV.EXE	Microsoft Environment Expansion Utility
CV.EXE	Microsoft CodeView Window-Oriented Debugger
MAKE.EXE	Microsoft Program Maintenance Utility
CL.EXE	Alternate control program for the compiler

Include Files

File Name	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 MS-DOS interface functions and declares MS-DOS interface functions
ERRNO.H	Defines system-wide error numbers
FCNTL.H	Defines flags used in open functions
FLOAT.H	Defines values used in floating-point operations
IO.H	Declares functions that work on file handles ("low-level" functions)
LIMITS.H	Defines upper and lower limits for various numeric types
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
STDARG.H	Defines macros for handling variable-length argument lists (as outlined in draft of ANSI C standard)
STDDEF.H	Defines standard values such as NULL and errno
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 types used by time functions
VARARGS.H	Defines macros for handling variable-length argument lists (similar to STDARG.H , but XENIX compatible)
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

File Name	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
CLIBC.LIB	Compact-model standard C library
CLIBFP.LIB	Compact-model floating-point math library
CLIBFA.LIB	Compact-model alternate math library
LIBH.LIB	Model-independent code-helper 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	Model-independent emulator floating-point library
87.LIB	Model-independent 8087/80287 floating-point library

Other Files

File Name	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.
CSETARGV.OBJ	Compact-model routine for processing wild-card characters.
LSETARGV.OBJ	Large-model routine for processing wild-card characters.
SVARSTCK.OBJ	Small-model routine for allowing dynamic heap allocation out of unused stack space.
CVARSTCK.OBJ	Compact-model routine for allowing dynamic heap allocation out of unused stack space.
MVARSTCK.OBJ	Medium-model routine for allowing dynamic heap allocation out of unused stack space.
LVARSTCK.OBJ	Large-model routine for allowing dynamic heap allocation out of unused stack space.
EMOEM.ASM	Module for customizing floating-point software.
CV.HLP	Help file for the CodeView debugger.
DEMO.C	Sample C program.
README.DOC	Documentation of changes and additions not appearing in these manuals. 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.

Start-up sources	A group of assembler routines and include files comprising basic start-up code for C programs; see README.DOC for a complete list of these files.
------------------	--

There may be additional sample C programs on the disk. If so, they will be listed in the **README.DOC** file.

2.4 Quick Hard-Disk Setup Procedure

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. However, if you use more than one memory model in your programming, you will probably want to add the appropriate library files from Disk 4, "Libraries Disk (Medium Model and Compact Model)," or Disk 5, "Libraries Disk (Large Model)," 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**.

Note

The following procedure assumes your hard disk is Drive C, and that you begin with **C:** as your current drive and directory.

1. With your system power on, and the MS-DOS prompt showing, enter the following commands (these set the environment variables so the compiler will look for the necessary executable files, libraries, and include files in the directories you will create in Step 2):

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

Note 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.)

To save the time it takes to enter these settings, you can place these commands in a batch file and set up the environment variables by entering the name of the file (see Section 2.14, "Using Batch Files").

2. Enter the following commands in the order shown, following the MS-DOS prompt (these create the directories in which you will store compiler files, libraries, and include files; if you already have any directories named BIN, LIB, INCLUDE, or INCLUDE\SYS on your hard disk, you should skip the commands that create those directories):

```
CD \  
MD BIN  
MD LIB  
MD INCLUDE  
MD INCLUDE\SYS
```

3. Insert Disk 1, "C Compiler Disk," in Drive A and type the following command at the MS-DOS prompt:

```
COPY A:.*.* \BIN
```

4. Replace the disk in Drive A with Disk 2, "Utilities Disk," and enter the following command at the MS-DOS prompt:

```
COPY A:*.EXE \BIN
```

5. Replace Disk 2 with Disk 3, "Include Files and Libraries Disk (Small Model)," and type this command following the MS-DOS prompt:

```
COPY A:LINK.EXE \BIN
```

6. Type

```
CD \BIN  
DIR
```

at the MS-DOS prompt to verify that the following files are now in your BIN directory:

EXEMOD.EXE	MSC.EXE
EXEPACK.EXE	C1.EXE
CV.EXE	C2.EXE
LIB.EXE	C3.EXE
LINK.EXE	SETENV.EXE
MAKE.EXE	

7. With Disk 3 still in Drive A, enter these commands following the MS-DOS prompt:

```
COPY A:*.H \INCLUDE  
COPY A:\SYS\*.H \INCLUDE\SYS
```

8. After the MS-DOS prompt, type the commands

```
CD \INCLUDE  
DIR
```

to verify that the following files have been copied to your INCLUDE directory:

ASSERT.H	FLOAT.H	SEARCH.H	STDLIB.H
CONIO.H	IO.H	SETJMP.H	STRING.H
CTYPE.H	LIMITS.H	SHARE.H	TIME.H
DIRECT.H	MALLOC.H	SIGNAL.H	V2TOV3.H
DOS.H	MATH.H	STDARG.H	VARARGS.H
ERRNO.H	MEMORY.H	STDDEF.H	
FCNTL.H	PROCESS.H	STDIO.H	

9. Next, type these two commands after the MS-DOS prompt:

```
CD SYS  
DIR
```

This confirms that these additional include files have been copied to the INCLUDE directory:

LOCKING.H
STAT.H
TIMEB.H
TYPES.H
UTIME.H

10. With Disk 3 still in Drive A, enter the following commands at the MS-DOS prompt:

```
COPY A:SLIBC.LIB \LIB  
COPY A:SLIBFP.LIB \LIB  
COPY A:EM.LIB \LIB  
COPY A:LIBH.LIB \LIB
```

11. Enter

```
CD \LIB  
DIR
```

to verify that the four files from the preceding step were copied to your LIB directory.

With this sample setup, you can run the compiler and linker (in fact, any of the .EXE files you have just copied) from any directory or disk.

If you use one of the following object files in your program, you can place the file either in your C program file directory or in the LIB directory:

File	Use
<i>xSETARGV.OBJ</i>	Enables wild-card expansion
<i>xVARSTCK.OBJ</i>	Enables stack/heap competition, where <i>x</i> is S, C, M, or L
BINMODE.OBJ	Changes the default text-processing mode

Note, however, that the LIB environment variable is not used to find the ***xSETARGV*** or **BINMODE** file; if it is not in your current working directory you must specify a path name at link time.

2.5 Quick Floppy-Disk Setup Procedure

You will need at least three floppy disks to set up the files so that you can run the compiler. The sample setup given below uses two disks and assumes the following:

- You will swap the two disks named “Compiler” and “Linker/Utilities/Libraries” in and out of Drive A as necessary.
- You will develop your programs and create listing files on a separate disk named “Include/Source Files” in Drive B.
- You will 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).

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 (small-, medium-, compact-, or large-model set). If

all your programs are small model, or if you will not use memory models, 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**.

Each disk drive must have a capacity of 360K for this sample setup procedure to work.

1. With your system power on, and the MS-DOS prompt showing, enter the following commands (these change the current drive to Drive A, and set the environment variables so the compiler will look for the necessary executable files, libraries, and include files in the directories you will create in the steps that follow):

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

Note 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 directory to store them. (**MSC.EXE** deletes the temporary files automatically; you are not responsible for removing them.)

To save the time it takes to enter these settings, you can place these commands in a batch file and set up the environment variables by entering the name of the file (see Section 2.14, "Using Batch Files").

2. Insert Disk 1, "C Compiler Disk," in Drive B, and a formatted disk in Drive A.
3. Type the following command following the MS-DOS prompt:

```
COPY B:*\.*
```

4. Type

```
DIR
```

following the MS-DOS prompt to verify that the following files have been copied to your disk in Drive A:

MSC.EXE
C1.EXE
C2.EXE
C3.EXE

5. Remove the disk in Drive A, label it “Compiler,” and replace it with another formatted disk.
6. Replace the disk in Drive B with Disk 2, “Utilities Disk.”
7. After the MS-DOS command, type the following commands, in sequence:

MD BIN
CD BIN
COPY B:LIB.EXE
COPY B:MAKE.EXE
COPY B:EXEPACK.EXE
COPY B:EXEMOD.EXE
COPY B:SETENV.EXE

8. Replace Disk 2 in Drive B with Disk 3, “Include Files and Libraries Disk (Small Model),” then type these commands, in sequence:

COPY B:LINK.EXE
CD \
MD LIB
CD LIB
COPY B:SLIBC.LIB
COPY B:SLIBFP.LIB
COPY B:EM.LIB
COPY B:LIBH.LIB

Type

DIR

to confirm that the following library files have been copied to the LIB directory on the disk in Drive A:

SLIBC.LIB
SLIBFP.LIB
EM.LIB
LIBH.LIB

Next, enter

CD \BIN
DIR

to confirm that the following utilities have been copied to your BIN directory:

LINK.EXE	EXEPACK.EXE
MAKE.EXE	EXEMOD.EXE
LIB.EXE	SETENV.EXE

9. Remove the disk in Drive A, label it "Linker/Utilities/Libraries," and replace it with another formatted disk.
10. Type the following commands, in the order shown, after the MS-DOS prompt:

```
MD INCLUDE  
CD INCLUDE  
COPY B:INCLUDE\*.H
```

11. Type

```
DIR
```

to confirm that the following files were copied to your INCLUDE directory:

ASSERT.H	FLOAT.H	SEARCH.H	STDLIB.H
CONIO.H	IO.H	SETJMP.H	STRING.H
CTYPE.H	LIMITS.H	SHARE.H	TIME.H
DIRECT.H	MALLOC.H	SIGNAL.H	V2TOV3.H
DOS.H	MATH.H	STDARG.H	VARARCS.H
ERRNO.H	MEMORY.H	STDDEF.H	
FCNTL.H	PROCESS.H	STDIO.H	

12. Type

```
MD SYS
```

to create the SYS subdirectory in INCLUDE.

13. Type the following commands, in the order shown:

```
CD SYS  
COPY B:\INCLUDE\SYS\*.H
```

14. Type

```
DIR
```

to verify that the following files were copied to SYS:

LOCKING.H
STAT.H
TIMEB.H
TYPES.H
UTIME.H

15. Remove the disk in Drive A and label it "Include/Source Files."

If you use one of the **xSETARGV.OBJ**, **xVARSTCK.OBJ**, or **BINMODE.OBJ** files (all of which are described in Section 2.4, "Quick Hard-Disk Setup Procedure"), you can place the file either in the directory with your C program files or in the **LIB** directory. Note, however, that the **LIB** environment variable is not used to find the **xSETARGV** or **BINMODE** file; when it is not in your current working directory, you must specify a path name 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. Note that the files stored on your "Compiler" and "Include/Source Files" disks (the compiler passes and the include files) do not change with the memory model, so you can use the same disks in the compiling stage for all five models.

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

Use the same directory structure on all four disks (small, medium, compact, and large) so 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 in the following manner 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 the "Linker/Utilities/Libraries" disk 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. Note 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.

2.6 Understanding the Compiler Software

The software for the Microsoft C Compiler consists of three main categories of files: executable files, include files, and library files. These files are listed in Section 2.3, "Disk Contents." Sections 2.6.1, 2.6.2, and 2.6.3, 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 2.6.4, "Other Files."

2.6.1 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**.

C1.EXE, **C2.EXE**, and **C3.EXE** are the three 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**).

Note

Version 3.0 of the Microsoft C Compiler had four passes. Pass 0, the preprocessor, and pass 1, the language parser, have been combined in Version 4.0.

The file **LINK.EXE** is the linker utility. 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. Invoke this utility by typing **LIB**.

EXEPACK.EXE and **EXEMOD.EXE** are special programs you can use to modify your executable program files. **SETENV.EXE** is a utility to modify the size of the DOS environment table. These functions are discussed in Appendix D, "Using EXEPACK, EXEMOD, and SETENV."

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 three passes of the compiler for you. You can also invoke the linker through **CL.EXE**.

2.6.2 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 with 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.

2.6.3 Library Files

Library files contain compiled run-time library routines to be linked with your program. Four separate sets of library files are included: small-model library files, medium-model library files, compact-model library files, and large-model library files. Huge-model programs use the large-model library files. The terms “small model,” “medium model,” “compact model,” “large model,” and “huge model” refer to the 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.

Three additional library files, **EM.LIB**, **LIBH.LIB**, and **87.LIB**, are model independent; they can be used with all five memory models. **EM.LIB** is the floating-point emulator, used to perform floating-point operations. **LIBH.LIB** is a library of model-independent “compiler helper” functions; the compiler generates references to these functions to handle complex operations such as 32-bit multiplication and division. **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 3.8, “Selecting Floating-Point Options,” in Chapter 3, “Compiling,” and in Section 9.9 “Controlling Floating-Point Operations,” in Chapter 9, “Advanced Topics.”

The library files beginning with **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 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 3.8, “Selecting Floating-Point Options,” in Chapter 3, “Compiling,” and in Section 9.9, “Controlling Floating-Point Operations,” in Chapter 9, “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**), the code-helper library (**LIBH.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 3.8 of Chapter 3, “Compiling,” and in Section 9.9 of Chapter 9, “Advanced Topics.”

The files beginning with **M** are medium-model library files, the files beginning with **C** are compact-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. **CLIBC.LIB**, **LLIBC.LIB**, and **MLIBC.LIB**, like **SLIBC.LIB**, each contain a start-up routine named **CRT0.OBJ**.

Note

Throughout the remainder of this manual, the convention ***xLIBC.LIB*** or ***xLIBFP.LIB***, where *x* is **S**, **C**, **M**, or **L**, will be used to refer to the standard library (small, compact, medium, or large) that is appropriate for the memory model chosen by the user.

This convention will also be used for other files that are supplied in sets of four, such as ***xSETARGV.OBJ***, in order to handle the five standard memory models in Microsoft C.

If you specify the medium, compact, or large model when you process your program, the compiler uses the appropriate standard library (***xLIBC.LIB***), floating-point libraries (by default, **EM.LIB** plus ***xLIBFP.LIB***), and the code-helper library (**LIBH.LIB**) when placing information in the object file for the linker. Otherwise, the compiler uses the small-model files.

2.6.4 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 five memory models (see Section 9.12, “Controlling Binary and Text Modes,” of Chapter 9, “Advanced Topics,” for details on **BINMODE.OBJ**).

The ***xSETARGV.OBJ*** files provide a routine that expands the MS-DOS wild-card characters **?** and ***** in file-name arguments passed to C programs from the command line. Wild-card expansion is performed only if you explicitly link with the appropriate **SETARGV** file. See Section 5.2, “Passing Command-Line Data to a Program,” for more information.

Linking with the **xVARSTCK.OBJ** files allows the heap to compete with the stack for memory space. In this way, the heap can allocate memory from unused stack space. See Section 9.8, "Controlling Stack and Heap Allocation," in Chapter 9, "Advanced Topics," for more information about the **xVARSTCK.OBJ** files.

The **EMOEM.ASM** allows you to customize floating-point software. See Section 3.8.3, "If Your Computer Is Not IBM Compatible," in Chapter 3, "Compiling."

The **CV.HLP** file is a help file for the CodeView symbolic debugger. The **COUNT.*** files are used in the practice session for the debugger (see your *Microsoft CodeView* manual for more information about these files).

The **README.DOC** file, if present, contains documentation of recent changes that may not be included in this manual, as well as documentation of the sources for the C start-up routines. 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.

DEMO.C, which is discussed in Section 2.13, "Practice Session," is a sample C program. Other demonstration programs may be included on your distribution disks. If so, they will be described in the **README.DOC** file.

2.7 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:

Files	Purpose
Executable files	These are the files the control program executes as it processes your program. The names of these files are C1.EXE , C2.EXE , and C3.EXE . When using CL.EXE , the alternate control program, LINK.EXE may also be executed by the control program. Note 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 or not 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. If you do set these variables, the compiler still searches the current working directory first. Then, if it does not find the file or files in the current working directory, it checks the appropriate environment variable for the path to the file. Exceptions to this sequence are **#include** files enclosed in angle brackets (`< >`). (See Section 8.3, "Include Files," in Chapter 8, "Preprocessor Directives and Pragmas," of the *Microsoft C Compiler Language Reference*.) An error is produced if the files are not found or if insufficient space is available in the specified directory or directories to create temporary files.

MSC.EXE looks for three environment variables: **PATH**, **INCLUDE**, and **TMP**. **LINK.EXE** uses one environment variable, **LIB**. (Like the compiler, **LINK** also checks the current working directory first for the libraries it needs, unless a library is specified with an absolute path name.) The alternate control program, **CL.EXE**, uses all four environment variables.

PATH tells the compiler and the operating system where to look for executable files, and **INCLUDE** tells them 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 and the operating system 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 double 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, commonly referred to as a "RAM disk," you can expedite processing by assigning that path 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.

Whereas the **TMP** variable can be assigned only one path name, the **INCLUDE**, **PATH**, and **LIB** variables can each contain more than one path name. Each path name is separated from the next path name 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, executable 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, as the following shows:

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

First the compiler will 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, you might use the following command line:

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

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

Note

The environment table is 160 bytes by default. If you want to set up a complex environment, this may not be enough space. You can use the **SETENV** program to increase the size of the environment table. See Section D.4 for more information.

MSC searches the current working directory, then all directories specified in the **PATH** command, 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 path name 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 path name 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 between different environments, you can save time by setting up batch files that contain the **SET** and **PATH** commands for each environment, thus allowing you to simply 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 3.6.6 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.

2.8 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 15 files. Check this by looking in your **CONFIG.SYS** file for the following line:

```
files=number
```

If *number* is less than 15, edit **CONFIG.SYS** to set *number* to an integer between 15 and 20. (Setting a number higher than 20 has no effect on the number of files per process. See your *Microsoft MS-DOS Programmer's Reference Manual* for more information.) If you do not currently have a **CONFIG.SYS** file, create a file by that name on your system disk (or root directory if you have a bootable hard disk) and insert the following line:

```
files=15
```

Note

If you do not specify enough files in the **CONFIG.SYS** file, you may see one of the following fatal error messages during compilation:

Cannot open compiler intermediate file — no more files

or

Cannot find '*includefile*'

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

`buffers=number`

If *number* 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.

2.9 Using an 8087 or 80287 Coprocessor

If you have an 8087 or 80287 coprocessor, you should read Section 3.8, "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 3.8.1, "If You Have an 8087 or 80287 Coprocessor," to take maximum advantage of your processor's capabilities.

2.10 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 3.9 of Chapter 3, "Compiling").

2.11 Converting Existing C Programs

If you are using an earlier version of the Microsoft C Compiler, or if you have programs written for such a compiler, turn to Appendix F, "Converting from Previous Versions of the Compiler," for a discussion of differences between this compiler and earlier versions. Some programs may need modification to compile correctly on Version 4.0.

2.12 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. (See Section 2.4, "Quick Hard-Disk Setup Procedure," and Section 2.5, "Quick Floppy-Disk Setup Procedure," for examples of how to create these 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 with the file name when you use a "system-level" include file in your program. For example, if you want the compiler to find and use the

include file **TIMEB.H** from the subdirectory **SYS** in the directory specified by the **INCLUDE** variable, use the following line in your program:

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

On the other hand, if you do not use the **SYS** convention, the following line is sufficient:

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

Note 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, unlike the XENIX operating system, where these two names would *not* be equivalent.

Sample setups for hard-disk systems and floppy-disk systems are given in Sections 2.4 and 2.5. Refer to the section that applies to your system.

2.13 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 the current value of environment settings. You can examine the **DEMO.C** source file to see how this is done. 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 C Programs on MS-DOS."

This practice session assumes that you are using the sample disk setup and environment that is appropriate for your system. See Section 2.4, "Quick Hard-Disk Setup Procedure," or Section 2.5, "Quick Floppy-Disk Setup Procedure," for examples of how to set up your disks.

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

PATH=C:\BIN

INCLUDE=C:\INCLUDE

LIB=C:\LIB

TMP=C:\

Floppy-Disk Settings

PATH=A:\;A:\BIN

INCLUDE=B:\INCLUDE

LIB=A:\LIB

TMP=B:\

If your settings do not match the above settings, turn back to Section 2.4 or 2.5 to review the disk setup and environment settings relevant to your system.

Once you have set up the environment, you are ready to begin processing **DEMO.C**. Follow steps 1–14 below:

1. First, set up a directory to hold program files. The directory can be on the hard disk or on the floppy disk named “Include/Source Files” created in Section 2.5. You can give the directory any name you like; for this session, the name **PROG** will be used. Next, copy **DEMO.C** from Disk 3, “Include Files and Libraries Disk (Small Model),” into the **PROG** directory.

Important

If you are using a floppy-disk setup, the disk containing the compiler executable files (“Compiler” from Section 2.5) should now be in Drive A.

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 type 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 (R) C Compiler Version 4.00  
Copyright (C) Microsoft Corp 1984, 1985, 1986. All rights reserved.  
Source file name [.C]:
```

Following the "Source file name" prompt, specify the name of the file or files to be compiled. (If you don't include the file-name extension when responding to this prompt, **MSC.EXE** assumes that the extension is **.C**. For this reason, your source file *must* have the file extension **.C** or **.c**.) Type

DEMO

in response to this prompt.

4. The next prompt is

```
Object file name [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 file name" 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

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

This prompt lets you create a source listing containing the source code on numbered lines and a table of symbols in the program. If errors are encountered during compilation, they will be shown immediately following the source lines that caused the error. Type

DEMO

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

6. 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**).

7. **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**, a source-listing file named **DEMO.LST**, and an object-listing file named **DEMO.COD** in your current working directory.

8. Next, you need to link your program.

Note

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, then insert the disk containing the **LINK** utility and the library files ("Linker/Utilities/Libraries" from Section 2.5).

To link your file, simply type

LINK

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

```
Microsoft (R) Overlay Linker Version 3.50
Copyright (C) Microsoft Corp 1983, 1984, 1985, 1986. All rights reserved.
```

9. 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. **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 path name to enable **LINK** to find it.

10. 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. If you don't supply a different name for the executable file, the linker uses the default name shown in brackets. The executable file is created in the current working directory (PROG).

11. The next prompt is

List File [NUL.MAP] :

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

12. 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.

13. **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 Library Manager, **LIB**, to organize object files into libraries of useful functions.

14. You can run the sample program by simply 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 processes).

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. Note that the **CL** command, described in Appendix C, "The CL Command," can be used to perform the same tasks as **MSC** and **LINK**.

2.14 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 more fully in your MS-DOS manual. This section is intended only to demonstrate a few of the 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 file name 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 **LINK**. The command-line method lets you give all responses to the prompts on a single line instead of waiting for the individual prompts. This method is discussed in Section 3.2.9 of Chapter 3, "Compiling," and in Section 4.2.9 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 NOT ERRORLEVEL 1 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. To run the batch file, type the following line:

```
MYCOMP THIS
```

The file name **THIS** is substituted for **%1**, and **THIS.C** is compiled, producing the object file **THIS.OBJ**.

The second line of the batch file 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 other exit codes, see Appendix E, "Using Exit Codes." The MS-DOS batch command **IF ERRORLEVEL** is used to test whether the exit code is 1 or greater. 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.

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

Chapter 3

Compiling

3.1	Introduction	51
3.2	Running the Compiler	52
3.2.1	File-Name Conventions	53
3.2.2	Special File Names	54
3.2.3	“Source file name” Prompt	55
3.2.4	“Object file name” Prompt	55
3.2.5	“Source listing” Prompt	55
3.2.6	“Object listing” Prompt	56
3.2.7	Selecting Default Responses	57
3.2.8	Swapping Disks	57
3.2.9	Using the Command Line	57
3.2.10	Options	60
3.3	Listing the Compiler Options	62
3.4	Naming the Object File	63
3.5	Producing Listing Files	64
3.6	Controlling the Preprocessor	70
3.6.1	Defining Constants and Macros	71
3.6.2	Predefined Identifiers	72
3.6.3	Removing Definitions of Predefined Identifiers	73
3.6.4	Producing a Preprocessed Listing	74
3.6.5	Preserving Comments	75
3.6.6	Searching for Include Files	75

3.7	Syntax Checking	76
3.7.1	Identifying Syntax Errors	77
3.7.2	Generating Function Declarations	77
3.8	Selecting Floating-Point Options	79
3.8.1	If You Have an 8087 or 80287 Coprocessor	80
3.8.2	If You Don't Have a Coprocessor	81
3.8.3	If Your Computer is not IBM Compatible	82
3.8.4	Compatibility Between Floating-Point Options	83
3.9	Using 80186, 80188, or 80286 Processors	84
3.10	Understanding Error Messages	85
3.10.1	C Compiler Messages	86
3.10.2	Setting the Warning Level	88
3.11	Preparing for Debugging	89
3.12	Optimizing	90
3.13	Compiling Large Programs	92

3.1 Introduction

You need only one basic command, **MSC**, to compile your C source files with the Microsoft C Compiler. The **MSC** command executes the three compiler passes for you.

With 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; you can specify the definitions for manifest (symbolic) constants and macros, and the kinds of warning messages you want to see.

Note

The options available with **MSC** are documented extensively in this chapter, as well as Chapter 8, "Working with Memory Models," Chapter 9, "Advanced Topics," and Appendix B, "Command Summary." For a quick overview of the more commonly used options, type

MSC /HELP

after the MS-DOS prompt. The **/HELP** option is described in greater detail in Section 3.3, "Listing the Compiler Options."

The **MSC** command automatically optimizes your program. You never have to give an optimizing instruction unless you either want to change the way **MSC** optimizes or disable optimization altogether. See Section 3.12, "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 9, "Advanced Topics." A summary of the **MSC** command and all available options is provided in Section B.2 of Appendix B, "Command Summary." Appendix C, "The **CL** Command," is a summary of the **CL** command, an alternative to the **MSC** command. **CL** is similar to the **cc** command on XENIX and UNIX systems, and is included for users who are accustomed to 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 Compiler Language Reference*. For questions relating to debugging C programs, see the *Microsoft CodeView* manual.

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 the following command at the MS-DOS command level:

MSC

MSC prompts for the input it needs by displaying the following four messages, one at a time:

```
Source file name [.C] :  
Object file name [basename.OBJ] :  
Source listing [NUL.LST] :  
Object listing [NUL.COD] :
```

where *basename* is the response (minus the .C extension – if any) you make to the first prompt.

The responses you make to each prompt are explained in the sections that follow.

If you want to stop a compiling session for any reason, type CONTROL-C. You will be returned to the MS-DOS command level, where you can start **MSC** from the beginning. If after doing this you discover new files beginning with **00** or **01** in the directory specified by the **TMP** environment variable, you can safely delete them; these are temporary compiler files that were not deleted because the compiling session was interrupted.

Note

Certain nonstandard MS-DOS environments (including some commonly used networks) often intercept some or all of the MS-DOS system calls and handle the calls themselves to provide additional or different capabilities. When running the compiler under such environments, the different operation of the system calls may cause some **MSC** functions to differ from their documented behavior. For example, compiler temporary files may not always be removed when you use CONTROL-C to terminate a compilation.

3.2.1 File-Name Conventions

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

abcde.fgh
AbCdE.FgH
ABCDE.fgh

You can include spaces before or after file names, but not within them. Options (see Section 3.2.10) can appear anywhere spaces can appear.

MSC uses the default file extensions **.C**, **.OBJ**, **.LST**, and **.COD** when you do not supply extensions with your file names. You can override the default extension for a particular prompt by specifying a different extension. To enter a file name 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 A: in response to the prompt. The output file is written on Drive A with the default file name.

Note that if you type any part of a legal path name following the "Source listing" prompt, **MSC** produces a source-listing file. The default name is the base name of the source file with the extension **.LST**. The base name 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 "Source listing" prompt, MSC produces a listing file on Drive A with the name **A:TEST.LST**.

MSC handles your response to the "Object listing" prompt in the same manner, using the extension **.COD** in place of **.LST** for the object listing.

3.2.2 Special File Names

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

Name	Device
AUX	Refers to an auxiliary device (usually the same as COM1).
CON	Refers to the console (terminal).
PRN	Refers to the printer device (usually the same as LPT1).
NUL	Specifies a "null" (nonexistent) file. Giving NUL as a file name means that no file is created.

Even if you add device designations or file-name extensions to these special file names, 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.

Notes

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

When using device names, do not follow them with a colon. The Microsoft C Compiler does not recognize the colon. For example, use CON or PRN, not CON: or PRN:, in your responses to MSC prompts.

3.2.3 “Source file name” Prompt

Following the “Source file name” 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.

Path names are allowed with the source-file name. Therefore, you can specify the path name 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-file name.

3.2.4 “Object file name” Prompt

Following the “Object file name” prompt, you can supply a name for the object file produced when you compile a source file. You are free to give any name and any extension you like. However, using the conventional **.OBJ** extension simplifies operation of **LINK** and **LIB**, both of which use **.OBJ** as the default extension when processing object files.

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

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

3.2.5 “Source listing” Prompt

If you supply a file name following the “Source listing” prompt, **MSC** creates a source listing, using the file name you supply. By convention, these listings are given the extension **.LST**, but you are free to choose any extension you like. If you do not supply a file name, the default is the special name **NUL.LST**, which tells **MSC** not to create a listing.

Note

Source listings were not available in Microsoft C Version 3.0, and the “Source listing” prompt and its corresponding place in the command line did not exist. If you are upgrading from Version 3.0, command lines in batch files or **MAKE** description files may need to be revised slightly to work correctly with Microsoft C Version 4.0. Specifically, if a Version 3.0 command line specifies an object listing, it will produce a source listing instead with versions 4.0 and higher.

Specifying a source listing at the “Source listing” prompt has the same effect as using the **/Fs** option. See Section 3.5, “Producing Listing Files.”

3.2.6 “Object listing” Prompt

By supplying a file name 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 assembler code for your program.

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

If you do not supply a file name, the default is the special name **NUL.COD**, which tells **MSC** not to create an object listing.

An object listing (unlike a source listing) can only be produced if the source file is compiled with no errors. 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 3.11, “Preparing for Debugging.”

Specifying an object listing at the “Object listing” prompt has the same effect as using the **/Fl** option. See Section 3.5, “Producing Listing Files,” for more information and an example of listing files. Section 3.5 also tells how to produce two variations of the object-listing file: assembly listings, and combined source and assembly listings.

3.2.7 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 (;) to terminate the line. Once the semicolon has been entered you cannot respond to any of the remaining prompts for that compiling session. Any text following 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 file name." You must enter a source-file name. The default for the "Object file name" is the base name of the source file with an **.OBJ** extension. The default for the "Source listing" prompt is the special name **NUL.LST**, which tells **MSC** not to create a source-listing file. The default for the "Object listing" prompt is the special name **NUL.COD**, which tells **MSC** not to create an object-listing file.

3.2.8 Swapping Disks

MSC suspends execution and displays a prompt whenever it cannot find one or more of the executable files that constitute the compiler: **C1.EXE**, **C2.EXE**, and **C3.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 file name" prompt with a nonexistent file name, or to the "Object file name," "Source listing," or "Object listing" prompt with an invalid path name, **MSC** displays an error message and terminates. You must restart **MSC** with the correct information.

3.2.9 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 file names on the line used to start **MSC**. The command-line method has the following form:

MSC *sourcefile* [[,][*objectfile*] [,][*sourcelistfile*][,][*objectlistfile*]]]]] [*options*] [;]

You can include spaces before or after file names, but not within them. Options (described in Section 3.2.10) can appear anywhere spaces can appear in the command line.

You can leave the *objectfile*, *sourcelistfile*, and *objectlistfile* fields blank to cause **MSC** to select the default file names. The semicolon (;) character has the same effect on 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 is ignored.

The comma (,) serves as a separator and also has a special function in the command line. If you place a comma after the *objectfile* field in the command line (whether or not *objectfile* is actually given), the default for the source-listing field is changed from **NUL.LST** to the base name of the source file plus **.LST**. Similarly, if a comma follows the source-listing field, the default for the object-listing field is changed from **NUL.COD** to the base name of the source file plus **.COD**. For example, the following 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 three files are produced: TEST.OBJ, TEST.LST, and TEST.COD.

In the second command line, only the source-file name is supplied. The default name TEST.OBJ is used for the object-file name, since none is specified. The comma following the object-file-name field causes the default for the listing files to be changed to TEST.LST and TEST.COD. Since no name is provided for the source and object listings, the default files are created.

The following line has a different effect:

```
MSC TEST;
```

This command creates an object file named TEST.OBJ, but does not create listing files, since no comma is present in the command line to change the defaults.

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 below:

Entry	Result
Semicolon (;	MSC uses the default responses for the remaining prompts.

file name	MSC prompts you for the remaining responses, if any.
Comma (,)	If you give just a source-file name followed by a comma, MSC prompts for the object-file name, source-listing name, and object-listing name, as usual. However, if you supply both a source-file name and an object-file name, and then terminate the command line with a comma, MSC changes the default source-listing name from NUL.LST to the base name of the source file plus .LST. MSC then prompts you for an object-listing name to allow you to override the default. (You can give the name NUL.LST to suppress the creation of a source listing.) The default object-listing name is changed in a similar fashion if the command line ends with a source-listing name followed by a comma.

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

MSC ASK.C, TELL.OBJ

MSC ASK, TELL;

MSC ASK.C, TELL.OBJ,

MSC ASK

The first example causes **MSC** to prompt with

Source listing[NUL.LST]

since you supplied the source-file name and object-file name but not the source- or object-listing file names.

Note the difference between the first example and the second example, which tells **MSC** to use the default response (no file) for the source and object listings. No further prompts appear in this case.

In the third example, the trailing comma (after TELL.OBJ) has a special meaning. It causes **MSC** to prompt as follows:

Source listing[TELL.LST] :

Note that the default name in brackets is TELL.LST rather than NUL.LST. In this case a source listing is created by default, unless you override the default to specify a different listing name (or the name NUL.LST, to suppress the listing).

In the final example, **MSC** starts prompting with the "Object file name" prompt, since only the source-file name is supplied.

3.2.10 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 dash 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.

Note

Although file names can be given in either uppercase or lowercase letters, 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 four file names (source-file name, object-file name, source listing, 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 file names, strings, or numbers. In most 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

With the **/NM**, **/NT**, and **/ND** options (discussed in greater detail in Section 9.14, “Naming Modules and Segments,” in Chapter 9, “Advanced Topics”), a space is *required* between the option and its argument. For example, **/NM testmodule** is acceptable, but **/NMtestmodule** is not, and will produce a command-line error.

The **/Gt** option and **/F** family of options (**/Fs**, **/Fa**, **/Fc**, **/Fl**, and **/Fo**, plus **/Fe** and **/Fm** with the **CL** command) are the only exceptions to allowing or requiring spaces between options and their arguments. The **/Gt** option accepts an optional numerical argument, while the **/F** options accept an optional path-name argument or partial path-name argument. When you supply an argument to one of these options, no spaces can 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 **/FcMINGLE** would also be an unacceptable form for the preceding option.

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 without a file name in response to the prompt.

The compiler options and the tasks they perform are discussed in the remainder of this chapter, in Chapter 8, “Working with Memory Models,” and in Chapter 9, “Advanced Topics.” The command-line form of the **MSC** command is used for the examples of options in this manual. Remember, you can use options with the prompts as well, as shown below.

Examples

MSC

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

The prompts and responses above produce exactly the same effect as the following command line:

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

In each case, the source file LOAD.C on Drive A is compiled, the object file is named OUT.OBJ and the source listing is named LOAD.SRC. 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 base name of the source file. The object file, source listing, and combined listing are created on the default drive, since no drive was specified. The /Oas option tells the compiler how to optimize the object file. The /Fc, /Fs, and /Oas options are discussed in detail in Section 3.5, "Producing Listing Files," and Section 3.12, "Optimizing."

3.3 Listing the Compiler Options

Option

/HELP
/help

This option prints on the console a list of the most commonly used compiler options. You can specify /HELP or /help as part of the MSC command line, or as part of the response to an MSC prompt. In either case, MSC processes all information on the line containing the /help option, prints the command list, and then, if needed, reissues the current prompt for further input. Note that all input you have given up to this point has been processed. For example, if you have typed a file name followed by /help, that file name will appear as the default value when the prompt is reissued.

The only exception to these rules concerns source-file names. If you type the source-file name with /help, the source-file prompt is not reissued. Instead, the object-file prompt is displayed following the command list.

This option is not case sensitive: any combination of uppercase and lowercase letters is acceptable; for example, /hELP.

3.4 Naming the Object File

Option

/Fo*objectfile*

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 file name at the “Object file name” prompt. When using the /Fo option, the *objectfile* 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 *objectfile*. However, it is recommended that you use the conventional .OBJ extension because it simplifies operation of LINK and LIB, both of which use .OBJ as the default extension when processing object files. If you give an object-file name without an extension, MSC automatically appends the .OBJ extension.

If you give only a drive or directory specification following the /Fo option, MSC creates the object file on the given drive or directory and uses the default file name (the base name 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 only a directory specification, the directory specification must end with a backslash (\) so that MSC can distinguish between a directory specification and a file name.

If you give a name following the “Object file name” prompt and also use the /Fo option, the name you give after the /Fo option overrides the name you give following the prompt.

Examples

MSC THIS, B:\OBJECT\;

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.

3.5 Producing Listing Files

Options

<code>/Fs[listfile]</code>	Produces source listing
<code>/Fl[listfile]</code>	Produces object listing
<code>/Fa[listfile]</code>	Produces assembly listing
<code>/Fc[listfile]</code>	Produces mixed source and assembly listing

In addition to the command-prompt method of creating listing files, you can use options to create source and object listings. You can also use options to create two kinds of listings that are not available through prompts: assembly listings and mixed source and assembly listings.

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

The list below shows the kinds of entries that can follow one of the listing file options:

Entry	Result
File name	MSC uses the given file name, appending the default extension if the file name has no extension. The file name can include a path to tell MSC where to create the listing.
Directory specification	MSC creates the listing in the given directory, using the default listing name, which is formed by appending the default extension to the base name of the source file. The directory specification must end with a backslash (\) so that MSC can distinguish between a directory specification and a file name.
Omitted	When no <i>listfile</i> is given, MSC uses the default listing name (base name of the source file plus the default extension) and creates the listing in the current working directory.

At most, one source-listing file and one variation of the object listing is produced each time you compile. Therefore, if you use both the **/Fa** and the **/Fl** options in one command line, only one file will be produced. The **/Fc** option overrides other listing options; whenever you use **/Fc** a combined listing is produced. If you give conflicting names for a listing file (for example, one following the prompt and one with the option), the last name specified has precedence.

The **/Fs** option produces a source-listing file. The information in the source listing is helpful in debugging programs as they are being developed, and is also useful for documenting the structure of a finished program. The source listing contains the numbered source-code lines, embedded error messages, and symbol tables. Error messages appear in the listing after the line that caused the error. The line number given in the error message corresponds to the number of the source line immediately above the message in the source listing. Include files are not expanded in the source listing; any errors detected in an include file are placed in the source listing immediately following the **#include** directive for that file.

The example below shows a section of code from a source listing:

```

8 FILE *infile;
9 char *name, line[100];
10 int nlines;
11
12 if (argc > 1) {
13     name = argv[argc - 1];
***** count.C(13) : error 65: 'name' : undefined
14     if ((infile = fopen(name, "r")) == NULL) {
15         fprintf(stderr, "%s couldn't open file %s\n",
16                 argv[0], name);
17         exit(1);
18     }
19 }
```

The error message shows that the variable `name` was used without being defined in line 27 of the source file `COUNT.C`. From the context, it is apparent that the variable `name` was intended, but typed incorrectly.

If the source file compiles with no errors more serious than warning errors, tables of segments, local symbols, and global symbols will be included in the source listing. Symbol tables will not be included if the compiler is unable to finish compilation.

At the end of each function, a table of local symbols is given, as shown below for the function main:

main Local Symbols

Name	Class	Offset	Register
name.	auto	-006a	
line.	auto	-0068	
infile.	auto	-0004	
nlines.	auto	-0002	
argc.	param	0004	
argv.	param	0006	

The Name column lists the name of each local symbol in the function. The Class column contains either auto if the symbol is a nonstatic local variable, or param if the symbol is a formal parameter. The Offset column shows the symbol's offset address relative to the frame pointer (that is, the **BP** register). The Offset number is positive for param symbols and negative for auto symbols with **auto** storage class. The Register column is blank unless the variable is stored in a register. If the variable is in a register, the column indicates the register (**SI** or **DI**).

At the end of the source code, a table of global symbols is given, as shown below:

Global Symbols

Name	Type	Size	Class	Offset
_iob.	struct/array	160	extern	***
exit.	near function	***	extern	***
fdopen.	near function	***	extern	***
fgets.	near function	***	extern	***
fopen.	near function	***	extern	***
fprintf.	near function	***	extern	***
main.	near function	***	global	0000
printf.	near function	***	extern	***

The Name column lists each global symbol, external symbol, and statically allocated variable declared in the source file.

The Type column shows a simplified version of the symbol's type as declared in the source file. The Type entry for a function is either near function or far function, depending on the memory model and how the function was declared. The Type entry for a pointer is near pointer, far pointer, or huge pointer. For enumeration variables, the Type entry is int. For structures, unions, and arrays, the Type entry is struct/array.

The **Size** column is only used for variables. This column specifies the number of bytes of storage allocated for the variable. Note that the amount of storage allocated for an external array may not be known, so its **Size** field may be undefined.

The **Class** column contains either **global**, **common**, **extern**, or **static**, depending on how the symbol was defined in the source file.

The **Offset** column is only used for symbols with an entry of **global** or **static** in the **Class** field. For variables, the **Offset** field gives the relative offset of the variable's storage in the logical data segment for the program file being compiled. Since the linker will, in general, combine several logical data segments into a physical segment, this number is useful only for determining the relative position of storage of variables. For functions, the **Offset** field gives the relative offset of the start of the function in the logical code segment. For small-model programs, logical code segments from different program files are combined into a single physical segment by the linker, so the **Offset** field is again useful, primarily to determine the relative positions of different functions defined in the same source file. However, for medium-, large-, and huge-model programs, each logical code segment becomes a unique physical segment. In these cases, the **Offset** field gives the actual offset of the function in its run-time code segment.

The last table in the source listing shows the segments used and their size, as shown below:

```
Code size = 0095 (149)
Data size = 003c (60)
Bss size = 0000 (0)
```

The byte size of each segment is given first in hexadecimal, and then in decimal (in parentheses). See Section 10.2.1 "Segment Model," in Chapter 10, "Interfaces with Other Languages," for a description of the segment model.

The **/F1** option produces an object-listing file. The object listing contains the machine instructions and assembly code for your program, as shown in the sample below:

```
; Line 12
*** 00000a    83 7e 04 01      cmp      WORD PTR [bp+4],1 ;argc
*** 00000e    7e 44          jle      $165
; Line 13
*** 000010    8b 76 04      mov      si,[bp+4];argc
*** 000013    d1 e6          shl      si,1
*** 000015    8b 5e 06      mov      bx,[bp+6];argv
*** 000018    8b 40 fe      mov      ax,[bx-2][si]
*** 00001b    89 46 96      mov      [bp-106],ax ;name
```

```
; Line 14
*** 00001e    b8 00 00      mov     ax,OFFSET DGROUP:$SG67
*** 000021    50          push    ax
*** 000022    ff 76 96      push    WORD PTR [bp-106] ;name
*** 000025    e8 00 00      call    _fopen
*** 000028    83 c4 04      add    sp,4
*** 00002b    89 46 fc      mov    [bp-4],ax;infile
*** 00002e    0b c0          or     ax,ax
*** 000030    75 32          jne    $I70
```

The line numbers are shown in the listing as comments. The machine instructions are on the left and assembly code on the right.

The **/Fa** listing produces an assembly listing of your program. The assembly listing contains the assembly code corresponding to your C file, as shown below:

```
; Line 12
        cmp     WORD PTR [bp+4],1      ;argc
        jle    $I65
; Line 13
        mov     si,[bp+4]           ;argc
        shl     si,1
        mov     bx,[bp+6]           ;argv
        mov     ax,[bx-2][si]
        mov     [bp-106],ax         ;name
; Line 14
        mov     ax,OFFSET DGROUP:$SG67
        push   ax
        push   WORD PTR [bp-106]      ;name
        call   _fopen
        add    sp,4
        mov    [bp-4],ax            ;infile
        or     ax,ax
        jne    $I70
```

Note that the sample shows the same code as in the object listing sample, except that the machine instructions are omitted. This is to ensure the listing will be suitable as input for the Microsoft Macro Assembler (**MASM**).

(however - see `readme.doc`)

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, as shown below:

```
;|*** if (argc > 1) {
; Line 12
*** 00000a    83 7e 04 01      cmp     WORD PTR [bp+4],1 ;argc
*** 00000e    7e 44          jle    $I65
;|*** name = argv[argc - 1];
; Line 13
```

```

*** 000010    8b 76 04      mov     si,[bp+4] ;argc
*** 000013    d1 e6        shl     si,1
*** 000015    8b 5e 06      mov     bx,[bp+6] ;argv
*** 000018    8b 40 fe      mov     ax,[bx-2][si]
*** 00001b    89 46 96      mov     [bp-106],ax   ;name
;|***          if ((infile = fopen(name,"r")) == NULL) {
; Line 14
*** 00001e    b8 00 00      mov     ax,OFFSET DGROUP:$SG67
*** 000021    50             push    ax
*** 000022    ff 76 96      push    WORD PTR [bp-106] ;name
*** 000025    e8 00 00      call   _fopen
*** 000028    83 c4 04      add    sp,4
*** 00002b    89 46 fc      mov     [bp-4],ax ;infile
*** 00002e    0b c0        or     ax,ax
*** 000030    75 32        jne    $I70

```

Note that this sample is like the object listing sample, except that the C source line is provided in addition to the line number.

When you examine a listing file, you will notice that the names of globally visible functions and variables begin with an underscore, as shown below (this part of the listing is the same for all three kinds of listings):

```

PUBLIC  _bytecount
PUBLIC  _charcount
PUBLIC  _wordcount
PUBLIC  _linecount
EXTRN  _fread:NEAR
EXTRN  _fopen:NEAR
EXTRN  _gets:NEAR
EXTRN  __chkstk:NEAR
EXTRN  _printf:NEAR
EXTRN  _perror:NEAR

```

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 10.2.7 of Chapter 10, "Interfaces with Other Languages."

The listing may also contain names that begin with more than one underscore (for example, __chkstk in the sample). Identifiers with more than one leading underscore are reserved for internal use by the compiler, and should not be used in your programs, except for those documented in the *Microsoft C Compiler Run-Time Library Reference*, such as **_psp**, **_ambblksize**, and **_fpreset()**. 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 will have two leading underscores and might conflict with the names reserved by the compiler.

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 codes. To produce a listing without optimizing, use the **/Od** option (discussed in Section 3.12, "Optimizing") with the listing option.

Examples

```
MSC HELLO.C /FsHELLO.SRC /FcHELLO.CMB;  
MSC HELLO /FsHELLO.SRC, ,HELLO.LST, HELLO.COD;
```

In the first example, **MSC** creates a source listing called **HELLO.SRC** and a combined source and assembly listing called **HELLO.CMB**. The object file has the default name **HELLO.OBJ**.

The second example produces a source listing called **HELLO.LST** rather than **HELLO.SRC**, since the last name provided has precedence. This example also produces an object-listing file named **HELLO.COD**. The object file in this example has the default name **HELLO.OBJ**.

3.6 Controlling the Preprocessor

The **MSC** command provides several 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 Compiler Language Reference* for a complete discussion of C preprocessor directives.

3.6.1 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, to remove all occurrences of `register`, use the following option:

`/Dregister=`

Note 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 remains in effect until the identifier is redefined 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 3.6.3, “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`. Note 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. Note that NEED does not have to be set to a specific value to be considered defined. The following command, in contrast, causes the statements in the **#if** block to be ignored (not compiled):

```
MSC MAIN.C;
```

3.6.2 Predefined Identifiers

The compiler defines four identifiers that are useful in writing portable programs. You can use these identifiers to conditionally compile code sections, depending on the processor and operating system being used. 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_I86xM	Always defined. Identifies memory model, where <i>x</i> is either S (small model), C (compact model), M (medium model), L (large model), or H (huge model). Small model is the default.

Memory models are discussed in Chapter 8, “Working with Memory Models.”

NO_EXT_KEYS

“No Extended Keywords.” Defined only when the /Za switch is given, thus disabling special keywords such as **far** and **fortran**. See Section 9.2, “Disabling Special Keywords,” in Chapter 9, “Advanced Topics.”

3.6.3 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 /U MSDOS /U M_I86 /U M_I86SM WORK.C;
```

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

3.6.4 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 file name" or "Object listing" prompt.

The /P option writes the preprocessed listing to a file with the same base name 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, and also around lines removed by conditional compilation preprocessor commands. 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

```
MSC MAIN.C /P;  
MSC ADD.C /E ; > PREADD.C  
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.

3.6.5 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.

3.6.6 Searching for Include Files

Options

`/Idirectory`
`/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 2.7, “Setting Up the Environment,” of Chapter 2, “Getting Started,” for a discussion of environment variables.)

You can add to the list of directories searched for include files by using the /I (for "include") option. This option causes the compiler to search the directory or directories 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

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

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

In the first example, **MSC** looks for the include files requested by **MAIN.C** in the following order: first in the directory **A:\INCLUDE**, then 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.

3.7 Syntax Checking

The options described in this section are useful in the early stages of program development. With the /Zs option, you can quickly check your program for syntax errors; with the /Zg option, you can automatically generate function declarations, which can then be used to enhance the syntax-checking capabilities of the compiler.

3.7.1 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.

3.7.2 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 many existing C programs will not have argument-type lists. See the *Microsoft C Compiler 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 otherwise identical.

Note

If you use the **/Zg** option and your program contains formal parameters that have structure, enumeration, or union type (or pointers to such types), then the declaration for each struct, enum, or union must have a tag. For example, use the following form:

```
struct tagA {  
    .  
    .  
    .  
} A;
```

Your program can 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 **#ifdefed()** blocks and are included only if you define the special identifier **LINT_ARGS**. You can define **LINT_ARGS** either by placing a **#define** directive before any **#include** directives in your program, or by using the **/D** option when you compile.

Example

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

The above command causes the compiler to generate argument-type lists for functions defined in FILE.C. The list of declarations is redirected to FILE.DEC.

3.8 Selecting Floating-Point Options

Option	Action
/FPa	Generates floating-point calls; selects alternate math library
/FPc	Generates floating-point calls; selects emulator library
/FPc87	Generates floating-point calls; selects 8087/80287 library
/FPi	Generates in-line instructions; selects emulator library (default)
/FPi87	Generates in-line instructions; 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 9.9, "Controlling Floating-Point Operations," in Chapter 9, "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 an 8087/80287 is not present. 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 generating in-line 8087/80287 instructions (this is the **/FPi** 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. Therefore, 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 determine which libraries it will link with. You can override the library name given in the object file at link time and link with a different library instead; see Section 9.9.1, "Changing Libraries at Link Time," in Chapter 9, "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.

3.8.1 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 **xLIBFP.LIB**, where *x* indicates the memory model chosen. 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. It selects the 8087/80287 library (**87.LIB**) and **xLIBFP.LIB**. The **/FPc87** option is slower than **/FPi87** because it makes function calls instead of using in-line instructions, but **/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 9.9.1 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 following message will appear:

Floating point not loaded

The **/FPi** option generates in-line instructions for an 8087/80287 and selects the emulator library (**EM.LIB**) and **zLIBFP.LIB**. If an 8087/80287 coprocessor is present at run time, it is used; if one is not present, 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 **/FPi** (default) or **/FPc** 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 9.9.2 of Chapter 9, "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.

3.8.2 If You Don't Have a Coprocessor

You have several options for generating floating-point calls without an 8087/80287 coprocessor. You can use the emulator library (**EM.LIB**) either with in-line instructions (**/FPi**), or with function calls (**/FPc**). Or you can use one of the alternate math libraries (**/FPa**). If you use the emulator library, the 8087/80287 coprocessor will be used if one is present at run time; if not, the emulator library will mimic the operation of an 8087. If you use the alternate math library, the 8087/80287 will be ignored if present.

The **/FPi** option is the default when you do not specify a floating-point option. It generates in-line instructions for an 8087/80287 coprocessor and selects the emulator library (**EM.LIB**) and **zLIBFP.LIB**. 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 generates floating-point calls to the emulator library and selects the emulator library (**EM.LIB**) and **xLIBFP.LIB**. The **/FPc** option is slower than **/FPI** because it makes function calls instead of using in-line instructions, but **/FPc** is more flexible than **/FPI**: 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 9.9.1 for instructions on changing libraries at link time.

The **/FPa** option generates floating-point calls and selects the alternate math library (**xLIBFA.FP**). 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, NaNs, and denormal numbers are not used.) Calls to this library provide the 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 9.9.1, "Changing Libraries at Link Time," 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 3.8.1, "If You Have an 8087 or 80287 Coprocessor," for a description of the appropriate options.

3.8.3 If Your Computer is not IBM Compatible

The exception handler in the libraries for 8087 or 80287 floating-point calculations (**EM.LIB** and **87.LIB**) are designed to work without modification on the IBM PC family of computers, and on closely compatible computers, including the Wang PC, the AT&T 6300, and the Olivetti personal computers. The libraries also need not be modified for the Texas Instruments Professional Computer, even though it is not compatible. Any machine that uses nonmaskable interrupts (NMI) for 8087 exceptions should work with the unmodified libraries. However, if your computer is not one of these, and if you are not sure if it is completely compatible, you may need to modify the 8087 libraries.

All Microsoft languages that support the 8087 intercept 8087 exceptions in order to produce accurate results and properly detect error conditions.

In order to make the libraries work correctly on noncompatible machines, you can modify the libraries. To make this easier, an assembly-language source file, **EMOEM.ASM**, is included on the distribution disk. Any machine that sends the 8087 exception to an 8259 Priority Interrupt Controller (master or master/slave) should be easily supported by a simple table change to the **EMOEM.ASM** module. The source file contains further instructions on how to modify **EMOEM.ASM** and patch libraries and executable files.

3.8.4 Compatibility Between Floating-Point Options

Each time you compile a source file, you can specify a floating-point option. When you link two or more source files 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** floating-point option is recommended for all compilations. The **/FPc** option offers the greatest 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 files **87.LIB** and **xLIBFP.LIB** 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 **xLIBFP.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** option, 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 (**xLIBC.LIB**), as well as the code-helper library, **LIBH.LIB**. Always give the names of the floating-point libraries *before* the names of other libraries in the "Libraries" field.

3.9 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 the 80186 and 80188 processors; use **/G2** for the 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 80186/80188 or 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. Programs compiled this way will also run on the machines with the 80186, 80188, or 80286 processor.

3.10 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 may encounter while using the **MSC** command. For a list of actual error messages, see Appendix H, "Error Messages."

Error messages produced by the compiler are sent to the standard output, which is usually your terminal. You can redirect the messages to a file or printer by using an MS-DOS redirection symbol, **>** or **>>**. (For more information on redirection, see your *Microsoft MS-DOS Programmer's Reference Manual*.) This is especially useful in batch-file processing. For example, the following command redirects error messages to the printer device (designated by **PRN**):

```
MSC RM.C; > PRN
```

The following command redirects error messages to the file **RM.ERR**:

```
MSC RM.C; > RM.ERR
```

Note that only output ordinarily sent to the console screen is redirected.

Example

Contents of **RM.C**:

```
#include <stdio.h>

main(argc, argv)
    int argc;
    char argv[];

{
    register int i;
    char *name;
```

```
for (i = 1; i < arg; ++i)
    if (unlink(name = argv[i])) {
        printf("couldn't delete %s : ", name);
        perror("");
    }
}
```

Contents of error-message file RM.ERR:

```
rm.c(11) : error 65: 'arg' : undefined
rm.c(12) : warning 47: '=' : different levels of indirection
```

Corrected version of RM.C:

```
#include <stdio.h>

main(argc, argv)
    int argc;
    char *argv[];

{
register int i;
char *name;

for (i = 1; i < argc; ++i)
    if (unlink(name = argv[i])) {
        printf("couldn't delete %s : ", name);
        perror("");
    }
}
```

3.10.1 C Compiler Messages

The C compiler displays messages about syntactic and semantic errors in a source file, such as misplaced punctuation, illegal use of operators, and undeclared variables. 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 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 for your information 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 3.10.2.

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 a message about the fatal error, the compiler terminates without producing an object file or checking for further errors. A source listing is produced if one was requested, but it will not contain a symbol table.

Compilation error messages identify actual program errors. No object file is produced for a source file that has such errors. A source listing is produced if one was requested, but it will not contain a symbol table. 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. A source listing is produced if one was requested, but it will not contain a symbol table. In some cases, command-line errors are fatal, and the compiler terminates processing without producing an object file, a source listing, or an object listing.

Compiler internal error messages indicate an error on the part of the compiler rather than your program. If you get one of these messages, note the conditions of the error and notify Microsoft, using the Software Problem Report at the back of this manual.

Error messages of all types have the following basic form:

filename(linenumber) : messagetype number: message

In this syntax, *filename* is the name of the source file being compiled and *linenumber* identifies the line of the source file containing the error. The *messagetype* will be one of the following: *error*, *warning*, *fatal*, or

Command line. The *number* is the number of the error and *message* is a self-explanatory description of the error.

The messages for each category are listed by number in Appendix H, "Error Messages."

In addition to error messages, the **MSC** control program returns an exit code that indicates the status of the compilation. Exit codes are useful with the MS-DOS batch command **IF ERRORLEVEL** and with the **MAKE** utility. They allow you to test for the success or failure of the compilation before proceeding with other tasks. Exit codes are discussed in more detail in Appendix E, "Using Exit Codes."

3.10.2 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 below and specify the corresponding *number* after the option. The **/w** option provides a shorter way to say **/W 0**, and has the same effect.

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 automatic data conversions, missing returns in function definitions.
3	Currently equivalent to warning level 2. Reserved for future releases.

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.

Examples

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

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

The first example directs the compiler to perform the highest level of checking, and produce 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 in the second example has the same effect as the following command:

```
MSC /W 0 MAIN.C;
```

3.11 Preparing for Debugging

Options

/Zd

/Zi

/Od

The /Zd option produces an object file containing line-number records corresponding to the line numbers of the source file. The /Zd option is useful when you want to pass an object file to the **SYMDEB** symbolic debugger, available with other Microsoft products. The debugger can use the line numbers to refer to program locations; however, only global symbol-table information is available with this product.

The /Zi option produces an object file containing full symbolic-debugging information for use with the CodeView symbolic debugger. This object file includes full symbol-table information and line numbers.

The **/Od** option tells the compiler not to perform complex optimizations involving code rearrangement; peephole optimizations and other simple optimizations are still performed. (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, since optimization can involve rearrangement of instructions that make it difficult for you to recognize and correct your code when debugging. However, turning off optimizations when your program is close to the size limits may increase the size of the code generated to the point where it might not be possible to link your program.

Other optimization options are discussed in Section 3.12, "Optimizing."

Example

```
MSC TEST.C, /Zi /Od, TEST;
```

This command produces an object file named TEST.OBJ that contains line numbers corresponding to the line numbers of TEST.C. A source-listing file, TEST.LST, is also created. Limited optimization is performed.

3.12 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 9.10 of Chapter 9, "Advanced Topics."

The *string* after the **/O** option lets you choose 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 (the default)
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 9.10.2 of Chapter 9, “Advanced Topics.”

When you do not give an **/O** option to the **MSC** command, it automatically uses **/Ot**, meaning that program execution speed 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 more efficient code. To cause the compiler to favor code size instead, use the **/Os** option.

The **/Od** option turns off optimizations that involve code rearrangement. This option is useful in the early stages of program development to avoid optimizing code that will later 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 letter can be used with either the **s** or the **t** option letter 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 letter 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 0 to `count`.

When you use the `a` option letter, 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 letter with the example above. 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 produce incorrect results.

Example

```
MSC FILE.C /Osa;
```

This command tells the compiler to relax alias checking and to optimize for smaller code size when it compiles `FILE.C`.

3.13 Compiling Large Programs

If you are compiling a program or file with more than 64K of data or with more than 64K of code, you may want to use one of the memory models described in Chapter 8, “Working with Memory Models.” 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 such as the following if an *individual* file exceeds these limits:

```
filename : error 27: DGROUP data allocation exceeds 64K
```

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). Using a compact memory model allows you to create programs with more than 64K of data (the 64K restriction on code still applies). In large- and huge-model programs, code and data can both exceed 64K (although in large-model programs no single data item can be larger than 64K).

If your program exceeds the 64K limit on data or code, you may also want to use the **far** (for data or code) or **huge** (for data only) keyword to selectively move items to a new segment. See Section 8.3 of Chapter 8, “Working with Memory Models,” for a discussion of these options.

No matter which memory model you use, 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.

()

()

()

Chapter 4

Linking

4.1	Introduction	97
4.2	Running the Linker	97
4.2.1	File-Name Conventions	98
4.2.2	“Object Modules” Prompt	98
4.2.3	“Run File” Prompt	99
4.2.4	“List File” Prompt	99
4.2.5	“Libraries” Prompt	100
4.2.6	Separating Entries	101
4.2.7	Selecting Default Responses	101
4.2.8	Terminating the Link Session	102
4.2.9	Using a Command Line	102
4.2.10	Using a Response File	103
4.2.11	The Temporary File	104
4.3	Linking C Program Files	105
4.3.1	The “main” Function	105
4.3.2	Default Libraries and the Library Search Path	106
4.3.3	Changing the Default Libraries	107
4.3.4	LINK Options to Avoid	107
4.4	Listing-File Format	107
4.5	Using Overlays	109
4.5.1	Restrictions	110
4.5.2	Overlay Manager Prompts	110
4.6	Using Options to Control the Linker	111
4.6.1	Viewing the Options List	112

4.6.2	Pausing During Linking	112
4.6.3	Packing Executable Files	113
4.6.4	Listing Public Symbols	114
4.6.5	Including Line Numbers in the List File	114
4.6.6	Preparing for Debugging	115
4.6.7	Preserving Case Sensitivity	116
4.6.8	Ignoring Default Libraries	116
4.6.9	Controlling Stack Size	117
4.6.10	Setting the Maximum Allocation Space	118
4.6.11	Controlling Segments	119
4.6.12	Setting the Overlay Interrupt	120
4.6.13	Ordering Segments	120
4.6.14	Controlling Data Loading	121
4.6.15	Controlling Run-File Loading	122
4.6.16	Preserving Compatibility	122
4.7	How the Linker Works	123
4.7.1	Alignment of Segments	123
4.7.2	Frame Number	124
4.7.3	Order of Segments	124
4.7.4	Combined Segments	125
4.7.5	Groups	125
4.7.6	Fix-ups	126

4.1 Introduction

The Microsoft Overlay Linker (**LINK**) is used to combine object files into a single executable run file. It can be used with object files compiled or assembled on 8086/8088 machines. The format of input to the linker is a subset of the Intel® object module format standard.

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

4.2 Running the Linker

LINK requires two types of input:

1. A command to start **LINK**
2. Responses to command prompts

Start **LINK** by typing the following command at the MS-DOS command level:

LINK

LINK prompts you for the input it needs by displaying the following four lines, one at a time:

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

LINK waits for you to respond to each prompt before printing the next one. The responses you can make to each prompt are explained in sections 4.2.1 through 4.2.7.

Once you understand the **LINK** prompts and operations, you can use the two alternate methods of running **LINK**: command line and response file. The command-line method (discussed in Section 4.2.9) lets you type all commands, options, and file names on the line used to start **LINK**. With

the response-file method (discussed in Section 4.2.10), you create a file that contains all the necessary commands, then tell **LINK** where to find that file.

You can also invoke **LINK** through the **CL** command. See Section C.3 of Appendix C, "The CL Command."

4.2.1 File-Name Conventions

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

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

LINK uses the default file extensions **.OBJ**, **.EXE**, **.MAP**, and **.LIB** when you do not supply extensions with your file names. You can override the default extension for a particular prompt by specifying a different extension. To enter a file name that has no extension, type the name followed by a period. For example, consider the following two responses to prompts:

```
ABC.  
ABC
```

If you typed the first line in response to a prompt, **LINK** would assume that the given file has *no* extension; if you typed the second line, **LINK** would use the default extension for that prompt.

4.2.2 "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.

LINK automatically supplies the **.OBJ** extension when you give a file name 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.

Path names are allowed with the object-file names. This means that you can give **LINK** the path name of an object file in another directory or on another disk. If **LINK** cannot find a given object file, it displays a message and waits for you to change disks.

Each object-file name must be separated from the next by one or more blank spaces or by 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 include more object files.

4.2.3 “Run File” Prompt

The “Run File” prompt lets you supply a name for the executable program file. You can give any file name you like; however, if you are specifying an extension, you should always use **.EXE**, since MS-DOS expects executable files to have this extension (or the **.COM** extension). (If you do not supply an extension, **.EXE** is supplied.)

You are allowed to skip this prompt by typing a carriage return without giving a name. By default **LINK** gives the executable file the base name of the first **.OBJ** file listed at the previous prompt. The **.EXE** extension then replaces the **.OBJ** extension of the object file.

4.2.4 “List File” Prompt

At the “List File” prompt you can tell **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 4.6.4) you can also include in the listing all public symbols and their addresses.

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

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

4.2.5 "Libraries" Prompt

Following the "Libraries" prompt you can give zero or more entries; each entry is separated from the others either by one or more blank spaces or by a plus sign (+). If the plus sign is the last character typed, the "Libraries" prompt reappears on the next line, allowing you to type additional entries. Each entry can be either a path specification or a library name. A path specification can be one of two things: a drive specification, in which case it ends with a colon (:); or a directory specification, in which case it ends with a backslash (\). A directory specification must end with a backslash (\) so that **LINK** can distinguish the directory names from the library names. When you give a path specification or specifications, **LINK** uses the specifications to search for the default libraries, as well as any other libraries given in response to the "Libraries" prompt without paths. You can specify up to 16 different paths; more than that are ignored. However, **LINK** will not return any error messages if you do have more than 16 path specifications.

To locate the default libraries, **LINK** searches in the following order:

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

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

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

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

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 are 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 one of the following options: **/FPc**, **/FPc87**, or **/FPa**). See Section 3.8 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 4.6.8.

4.2.6 Separating Entries

Use the plus sign (+) or one or more space characters to separate file-name entries in a list of object files or libraries. 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 file-name entry; the plus sign can be used only after complete file names.

Example

LINK

```
Object Modules [.OBJ]: FUN TEXT TABLE CARE+
Object Modules [.OBJ]: YOYO+FLIPFLOP+JUNQUE+
Object Modules [.OBJ]: CORSAIR
Run File [FUN.EXE]: ;
```

4.2.7 Selecting Default Responses

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

To select default responses to the current prompt and all remaining prompts, use a 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 “Object Modules” prompt, because there is no default response for that prompt.

Defaults for the other linker prompts are shown below:

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 file name **NUL.MAP**, which tells **LINK** *not* to create a listing file.
3. The default for the "Libraries" prompt is no libraries; in this case, the default libraries are those encoded in the object module. (See Section 4.3.2, "Default Libraries and the Library Search Path.")

4.2.8 Terminating the Link Session

To terminate the link session, press CONTROL-C while entering responses or while **LINK** is working. If you realize that you entered an incorrect response at a previous prompt, you should press CONTROL-C to exit **LINK** and begin again. You can use the normal MS-DOS editing keys to correct entries at the current prompt.

4.2.9 Using a Command Line

To invoke the linker with a command line, give your responses to the command prompts on a single line following the **LINK** command. The responses to the prompts must be separated by commas, as shown below:

```
LINK objectfiles [,,[executablefile] [,,[mapfile] [,,[libraryfiles]]]][[options]] [s]
```

Here *objectfiles* are object-module names, separated by plus signs or spaces. The *executablefile* is the name of the file to receive the executable output. The *mapfile* is the name of the file containing a symbol map listing. The *libraryfiles* are libraries and directories to be searched, separated by plus signs or spaces.

You do not have to give any *options* when you run the linker. If you specify options, you can put them anywhere on the command line. The options available with **LINK** are described in Section 4.6.

You can select the default response for any prompt by omitting the file name or names before the comma. The only exception to this is the default for the listing file: if you use a comma as a placeholder for the listing file on

the command line, **LINK** *will* create a listing file. This file has as its base the base of the run file. For example, the command line

LINK FUN,,;

produces the listing file FUN.MAP; in contrast, the command lines

LINK FUN,;
LINK FUN;

do not produce a listing file.

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

Example

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

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 map file called FUNLIST.MAP is also produced.

4.2.10 Using a Response File

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

LINK @filename

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

A response file contains responses to the **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 **LINK** prompts discussed above. Each new response begins with a new line or a comma; however, you can extend long responses across more than one line by typing a plus sign (+) as the last character of each incomplete line.

You can also enter the name of a response file after any of the linker prompts, or at any position in a command line. The input from the response file will be treated exactly as if it had been entered after prompts or in command lines, with a carriage-return-line-feed combination in the response file treated the same as a RETURN key in response to a prompt, or a comma in a command 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, **LINK** uses the default responses for the executable file and for the remaining prompts.

When you give the **LINK** command with a response file, each **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 file names, the semicolon command character, or carriage returns), **LINK** displays the missing prompts and waits for you to enter responses. When you type an acceptable response, **LINK** continues the link session.

Example

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

This response file tells **LINK** to load the four object modules FUN, TEXT, TABLE, and CARE. The executable file, FUN.EXE, and the map file, FUNLIST.MAP, are produced. The /PAUSE option causes **LINK** to pause before producing the executable file. This permits you to swap disks if necessary. The /MAP option tells **LINK** to include public symbols and addresses in the map file. **LINK** also links any needed routines from the library file, GRAF.LIB. See the discussion of the /PAUSE and /MAP options in Section 4.6, "Using Options to Control the Linker," for more on these options.

4.2.11 The Temporary File

LINK uses available memory for the link session. If the files to be linked create an output file that exceeds available memory, **LINK** creates a temporary disk file to serve as memory. If the linker is running on DOS Version 3.0 or later, it uses a DOS system call to create a temporary file with a

unique name in the current working directory. If the linker is running on a version of DOS prior to 3.0, it creates a temporary file named **VM.TMP**. When this happens, you will see the message

Temporary file *tempfilename* has been created.
Do not change diskette in drive, *letter*

where *tempfilename* is ".\\" followed by either a DOS-generated name or **VM.TMP**, and *letter* is the current drive. After this message appears, you must not remove the disk from the drive specified by *letter* until the link session ends. If the disk is removed, the operation of **LINK** is unpredictable. You may see the following message:

Unexpected end-of-file on scratch file

When this happens, you must rerun the link session. The temporary file created by **LINK** is a working file only. **LINK** deletes it at the end of the link session.

Note

Do not give any of your own files the name **VM.TMP**. The **LINK** utility will display an error message if it encounters an existing file with this name.

4.3 Linking C Program Files

Several special considerations that should be kept in mind when using **LINK** with C files are discussed in sections 4.3.1 through 4.3.4.

4.3.1 The “main” Function

When linking C programs, one (and only one) of the object files you submit to **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, **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 **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 give a path specification showing where the library files reside. (A path specification is a directory name, a drive letter, or a drive letter followed by a directory name.) You can do this by giving path specifications following the **LINK** "Libraries" prompt, by setting the **LIB** environment variable, or by combining the two methods.

You can give zero or more path specifications following the **LINK** "Libraries" prompt. Each path specification must end with a backslash (\) so **LINK** can recognize the specification as a directory name (rather than a library name) unless the path specification is just a drive letter, in which case it would end with a colon (:).

The **LIB** variable can contain one or more path specifications. See Section 2.7 of Chapter 2, "Getting Started," for a detailed discussion of environment variables.

To locate library files, **LINK** goes through the following procedure:

1. The current working directory is searched.
2. If the library files have not been found, **LINK** searches any paths specified following the **LINK** "Libraries" prompt. The directories are searched in order of their appearance on the line.
3. If the library files have not been found, **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 **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. **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 path name, **LINK** searches just that path name for the library; no other directory specifications apply.

4.3.3 Changing the Default Libraries

If you use the **/FPa**, **/FPc87**, or **/FPc** 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 9.9.1 of Chapter 9, “Advanced Topics,” for details.

If you do not want to use the standard C library (**xLIBC.LIB**), you must give the **/NOD** (for “no default library”) option when you link. This option tells **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 4.6.8 for details.

4.3.4 LINK Options to Avoid

Some of the options available with **LINK** 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 with the other **LINK** options because you may need them if you use **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.

4.4 Listing-File Format

You can tell **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. An example is shown below:

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.

The starting address and name of each group is listed at the end of the file. A sample group listing is shown below:

```
Origin Group  
0173:0 DGROUP
```

In this example, **DGROUP** is the name of the data group. **DGROUP** is the only group used by programs compiled with the Microsoft C Compiler, Version 4.0. Version 3.0 uses **IGROUP** for code segments.

If you use the **/MAP** option (see Section 4.6.4), **LINK** 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. An example is shown below:

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 addresses of the external symbols are in the “*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 10.2.7 of Chapter 10, "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 will have two leading underscores, and might conflict with the names reserved by the compiler.

4.5 Using Overlays

You can direct Microsoft **LINK** to create an overlaid version of your program; parts of your program will only be loaded if and when they are needed, and will share the same space in memory. Programs that use overlays are usually smaller and require less memory, but they run more slowly because of the time needed to read and reread the code from disk into memory.

You specify overlays by enclosing them in parentheses in the list of modules that you submit to the linker. Each parenthetical list represents one overlay. For example, you could give the following response to the "Object Modules" prompt:

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

In this example, the modules *(b+c)*, *(e+f)*, and *(i)* are overlays. The remaining modules, and any drawn from the run-time libraries, constitute the resident part (or root) of your program. 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 63 (3F hexadecimal).

4.5.1 Restrictions

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-, large-, and huge-model programs. See Chapter 8, "Working with Memory Models," for details on the standard memory models.

You cannot use long jumps (using 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.

4.5.2 Overlay Manager Prompts

The overlay manager is part of the C run-time library. If you specify overlays during linking, the code for the overlay manager is automatically linked with the other modules of your program. When the executable file is run, the overlay manager searches for that file whenever another overlay needs to be loaded. The overlay manager first searches for the file in the current directory; then, if it does not find the file, the manager searches the directories listed in the **PATH** environment variable. When it finds the file, the overlay manager extracts the overlay modules specified by the root program. If the overlay manager cannot find an overlay file when needed, it prompts the user to enter the file name.

Note

Even with overlays, the linker produces only *one .EXE* file. This file is opened again and again, as long as the overlay manager needs to extract new overlay modules.

For example, assume an executable program called **PAYROLL.EXE**, which does not exist in either the current directory or the directories specified by **PATH**, uses overlays. If the user runs it by entering a complete path specification, the overlay manager will display the following message when it attempts to load overlay files:

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

The user can then enter the drive and/or directory where PAYROLL.EXE is located. For example, if the file is located in directory \EMPLOYEE\DATA\ on Drive B, the user could enter B:\EMPLOYEE\DATA\ or simply \EMPLOYEE\DATA\ if the current drive is B.

If the user later removes the disk in Drive B and the overlay manager needs to access the overlay again, it will not find PAYROLL.EXE, and will display the following message:

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

After the overlay file has been read from the disk, the overlay manager will display the following message:

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

4.6 Using Options to Control 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 **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 **LINK**, options can appear at the end of the line or after individual responses on the line, but must be before the comma separating each response from the next item. In a response file, options can occur alone or following individual responses 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 space and effort. Be sure that your abbreviation is unique so that the linker can determine which option you want. Since several options begin with the letters NO, abbreviations for those options must be longer than NO to be unique. For example, NO is an illegal abbreviation for the /NOIGNORECASE option, since **LINK** would not be able to tell which

of the options beginning with NO you intended. The shortest legal abbreviation for this option is NOI.

Abbreviations must be sequential from the first letter of the option through the last letter typed. No gaps or transpositions are allowed. Some linker options take numeric arguments. A numerical argument can be any of the following:

- A decimal number from 0 to 65535.
- 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.

4.6.1 Viewing the Options List

Option

/HELP

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

4.6.2 Pausing During Linking

Option

/PAUSE

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

If the **/PAUSE** option is given, **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. **LINK** resumes processing when you press the ENTER key.

Note

Do not remove the disk that will receive the list file or the disk used for the temporary file, if one has been created (see Section 4.2.11). If the temporary-disk-file message appears when you have specified the **/PAUSE** option, you should press CONTROL-C to terminate the **LINK** session. Rearrange your files so that the temporary file and the executable file can be written to the same disk. Then try again.

4.6.3 Packing Executable Files

Option

/EXEPACK

The **/EXEPACK** option directs **LINK** to remove sequences of repeated bytes (typically nulls) and optimize the load-time relocation table before creating the executable file. Executable files linked with this option may be smaller, and thus load faster, than files linked without this option. However, you cannot use the **SYMDEB** or **CODEVIEW** symbolic debuggers with packed files; **EXEPACK** strips symbolic information from the input file, and notifies you of this with the following message:

`exepack: (warning) omitting debug data from output file`

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

4.6.4 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 causes **LINK** to create a listing file (also known as a “map file”). The following list describes the effects of this option when used with the prompt and command-line methods:

- Command-line method:

/MAP causes **LINK** to create a listing file, even if no file is specified in the command line. By default, this listing file is given the same base name as the executable file, plus the extension **.MAP**. You can override this default name by giving a name on the command line.

- Prompt method:

If **/MAP** appears before the “List File” prompt, it creates a listing file, even if you do not type a file name at the “List File” prompt. By default, the file is given the same base name as the executable file, plus the extension **.MAP**. You can override the default name by responding to the “List File” prompt.

If the **/MAP** option appears *after* the “List File” prompt, the option takes effect only if you have already explicitly created a listing file by giving a name at the “List File” prompt.

You must specify the **/MAP** option if you intend to debug your program using **SYMDEB**, the symbolic debugger provided with some versions of Microsoft languages.

4.6.5 Including Line Numbers in the List File

Option

/LINENUMBERS

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

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

You must specify the **/LINENUMBERS** option if you intend to do source-level debugging of your program using **SYMDEB**, the symbolic debugger provided with some versions of Microsoft languages.

The **/LINENUMBERS** option forces **LINK** to create a map file, even if no map file is specified in the **LINK** command line or at the "List File" prompt. By default, the file is given the same base name as the executable file, plus the extension **.MAP**. You can override the default name by responding to the "List File" prompt.

4.6.6 Preparing for Debugging

Option

/CO

The **/CO** option is used to prepare for debugging with the CodeView debugger, the symbolic debugger provided with Version 4.0 of the Microsoft C Compiler. This option tells the linker to prepare a special executable file containing symbolic data and line-number information.

You can run this executable file outside the CodeView debugger; the extra data in the file will be ignored. However, to keep file size to a minimum, you will probably want to use the special-format executable file for debugging only, and link a separate version without the **/CO** option after the program is debugged.

The **/CO** option can write this information to the executable file only if you used the **/Zi** option when compiling. **/Zi** also disables a number of optimizations; you can remove all optimizing with the **/Od** option. For example, to debug the C program **TEST.C**, you could use the following command lines:

```
MSC /Zi TEST..;
LINK /CO TEST..;
CODEVIEW TEST.EXE
```

4.6.7 Preserving Case Sensitivity

Option

/NOIGNORECASE

By default, **LINK** treats uppercase letters and lowercase letters as equivalent. Thus ABC, abc, and Abc are considered the same name. When you use the **/NOIGNORECASE** option (usually abbreviated **/NOI**), the linker distinguishes between uppercase letters and lowercase letters, and considers ABC, abc, and Abc as 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 always link with the **/NOI** option. The **CL** control program uses the **/NOI** option by default, but you must give it specifically if you use **MSC** and **LINK**.

Remember 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.

4.6.8 Ignoring Default Libraries

Option

/NODEFAULTLIBRARYSEARCH

The **/NODEFAULTLIBRARYSEARCH** option (usually abbreviated to **/NOD**) tells **LINK** *not* to search the default library, if there is one, to resolve external references. With C files this has the effect of telling **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 the **README.DOC** file included in your software for a list of the files comprising the start-up routines.)

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. The code-helper library, **LIBH.LIB**

4.6.9 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 65536 (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. 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. In contrast, if your program uses very little local data, you may achieve some space savings by decreasing the stack size.

If **LINK** cannot find the stack information it needs, it displays the following error message:

WARNING: NO STACK SEGMENT

Since the start-up file provides stack information, this message usually means that the start-up file is not being linked with your program.

Note

The **EXEMOD** utility (described in Appendix D, “Using EXEPACK, EXEMOD, and SETENV”) can also be used to change the default stack size for C program files. The format of the executable file header that is changed by this option is discussed in the *Microsoft MS-DOS Programmer’s Reference Manual* and in some other reference books on MS-DOS.

4.6.10 Setting the Maximum Allocation Space

Option

/CPARMAXALLOC:*number*

The **/CPARMAXALLOC** option (usually abbreviated to **/CP**) sets the maximum number of 16-byte paragraphs needed by the program when it is loaded into memory. This number is used by the operating system when allocating space for the program prior to loading it. The option is useful when you want to execute another program from within your program and you need to reserve space for the executed program.

LINK normally sets the maximum number of paragraphs to 65535. Since this represents all available memory, the operating system always denies the request and allocates the largest contiguous block of memory it can find. If the **/CP** option is used, the operating system will allocate no more space than given by this option. This means any additional space in memory is free for other programs.

The *number* can be any integer value in the range 1 to 65535. If *number* is less than the minimum number of paragraphs needed by the program, **LINK** ignores your request and sets the maximum value equal to the minimum. The minimum number of paragraphs needed by a program is never less than the number of paragraphs of code and data in the program. To free more memory for programs compiled in the compact, medium, and large memory models, link with **/CP:1**; this leaves no space for the “near” heap.

Note

You can also change the maximum allocation after linking with the **EXEMOD** utility. See Section D.3 of Appendix D, “Using EXEPACK, EXEMOD, and SETENV.” The format of the executable file header that is changed by this option is discussed in the *Microsoft MS-DOS Programmer’s Reference Manual* and in some other reference books on MS-DOS.

4.6.11 Controlling Segments

Option

/SEGMENTS:*number*

The **/SEGMENTS** option (usually abbreviated to **SE**) controls the number of segments the linker allows a program to have. The default is 128, but *number* can be set to any value (decimal, octal, or hexadecimal) in the range 1 to 1024 (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 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.

If the number of segments allocated is too many for the amount of memory **LINK** has available to it, you will see the following error message:

Segment limit too high

Set a lower limit and relink.

4.6.12 Setting the Overlay Interrupt

Option

/OVERLAYINTERRUPT:*number*

By default, the interrupt number used for passing control to overlays is 63 (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.

In general, you should not use **/OVERLAYINTERRUPT** with C routines. The exception to this guideline would be a C program using overlays that spawns another C program using overlays; in this case, each program should use a separate overlay interrupt number, meaning at least one of the programs should be compiled with **/OVERLAYINTERRUPT**.

4.6.13 Ordering Segments

Option

/DOSSEG

The **/DOSSEG** option forces segments to be ordered as follows:

1. All segments with a class name ending in **CODE**.
2. All other segments outside **DGROUP**.
3. **DGROUP** segments, in the following order:
 - Any segments of class **BEGDATA** (this class name is reserved for Microsoft use)
 - Any segments not of class **BEGDATA**, **BSS**, or **STACK**
 - Segments of class **BSS**
 - Segments of class **STACK**

C programs always use this order by default, so you never need to use this option. See Section 9.14, “Naming Modules and Segments,” for a discussion of the segment names used by the C compiler.

4.6.14 Controlling Data Loading

Option

/DSALLOCATE

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

Use the **/DSALLOCATE** option to tell **LINK** to load all data starting at the high end of the data segment, instead. In this case the **DS** register 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** register.

Warning

Do not use the **/DSALLOCATE** option with C programs. It should only be used with assembly-language programs.

4.6.15 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 **LINK** to place the run file as high as possible in memory. Without the **/HIGH** option, **LINK** places the run file as low as possible.

Note

Do not use the **/HIGH** option with C programs. It should only be used with assembly-language programs.

4.6.16 Preserving Compatibility

Option

/NOGROUPASSOCIATION

The **/NOGROUPASSOCIATION** option causes the linker to ignore group associations when assigning addresses to data and code items. It is provided primarily for compatibility with previous versions of the linker (versions 2.02 and earlier) and other Microsoft language compilers.

Note

Do not use the **/NOGROUPASSOCIATION** option with C programs. This option exists strictly for compatibility with older versions of FORTRAN and Pascal (Microsoft versions 3.13 and earlier, or IBM versions prior to 2.0). The **/NOGROUPASSOCIATION** option should never be used except to link with object files produced by those compilers, or with the run-time libraries that accompany the old compilers.

4.7 How the Linker Works

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)

The “executable image” contains the code and data that constitute 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.

You can control the way LINK combines 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 10.2.1 of Chapter 10, “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. You do not need to understand this information to use the linker, but it may be helpful for advanced users who want to link C routines with assembly routines.

4.7.1 Alignment of Segments

LINK uses a segment’s alignment type to set the starting address for the segment. The alignment types are **BYTE**, **WORD**, **PARA**, and **PAGE**. These correspond to starting addresses at byte, word, paragraph, and page boundaries, representing addresses that are multiples of 1, 2, 16, and 256, respectively. The default alignment is **PARA**.

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 null bytes.

The C compiler automatically assigns alignment types to segments. Table 10.1 of Chapter 10, "Interfaces with Other Languages," shows the align types of the segments used by each of the standard memory models.

4.7.2 Frame Number

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

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

4.7.3 Order of Segments

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

The C compiler automatically assigns class types to segments. Table 10.1 in Chapter 10, "Interfaces with Other Languages," shows the class types of the segments used by each of the standard memory models.

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 4.6.13, "Ordering Segments."

4.7.4 Combined Segments

LINK uses combine types to determine whether or not two or more segments sharing the same segment name should be combined into one large segment. The valid combine types are **PUBLIC**, **STACK**, **COMMON**, **MEMORY**, and **PRIVATE**.

If a segment has combine type **PUBLIC**, **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, **LINK** displays an error message.

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

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

The C compiler automatically assigns combine types to segments. Table 10.1 in Chapter 10, "Interfaces with Other Languages," shows the combine types of the segments used by each of the standard memory models.

4.7.5 Groups

Groups let segments that are not contiguous and do not belong to the same class be addressable relative to the same frame address. When **LINK** encounters a group, it adjusts all memory references to items in the group so that they are relative to the same frame address.

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.

The C compiler uses a group called **DGROUP** for data segments. For more information on how the Microsoft C Compiler uses groups, see Section 10.2.1.2 in Chapter 10, "Interfaces with Other Languages."

4.7.6 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 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 in either direction.

A near self-relative reference occurs in instructions that 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 0, or if the beginning of the canonical frame of the target is not addressable.

(

)

)

Chapter 5

Running C Programs on MS-DOS

5.1	Introduction	131
5.2	Passing Command-Line Data to a Program	131
5.2.1	Expanding Wild-Card Arguments	134
5.2.2	Suppressing Command-Line Processing	135
5.3	Returning an Exit Code	136
5.4	Suppressing Null-Pointer Checks	137

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

90

91

92

93

94

95

96

97

98

99

100

101

102

103

104

105

106

107

108

109

110

111

112

113

114

115

116

117

118

119

120

121

122

123

124

125

126

127

128

129

130

131

132

133

134

135

136

137

138

139

140

141

142

143

144

145

146

147

148

149

150

151

152

153

154

155

156

157

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

174

175

176

177

178

179

180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

205

206

207

208

209

210

211

212

213

214

215

216

217

218

219

220

221

222

223

224

225

226

227

228

229

230

231

232

233

234

235

236

237

238

239

240

241

242

243

244

245

246

247

248

249

250

251

252

253

254

255

256

257

258

259

260

261

262

263

264

265

266

267

268

269

270

271

272

273

274

275

276

277

278

279

280

281

282

283

284

285

286

287

288

289

290

291

292

293

294

295

296

297

298

299

300

301

302

303

304

305

306

307

308

309

310

311

312

313

314

315

316

317

318

319

320

321

322

323

324

325

326

327

328

329

330

5.1 Introduction

After compiling a program with the Microsoft C Compiler and linking with the linker, you will have an executable file with the extension **.EXE** that can be run from the MS-DOS prompt.

MS-DOS uses the **PATH** environment variable to find executable files. You can execute your 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 set in the **PATH** environment variable.

Your program can also be executed by other programs, or you can write it so that it will be capable of executing other programs or MS-DOS internal commands. The **spawn**, **exec**, and **system** routines provided in the Microsoft C Run-Time Library allow your program to execute other programs and MS-DOS commands. See the *Microsoft C Compiler Run-Time Library Reference* for a description of these routines.

MS-DOS has several other unique capabilities that your program can use if you write the program to take advantage of them. Among these capabilities are the following:

- Receiving arguments from MS-DOS
- Reading data that were previously passed to the MS-DOS environment table
- Sending a message to MS-DOS by returning an exit code

These features are not a part of the C language, but rather a part of Microsoft's MS-DOS implementation of C. They either don't exist in other operating systems or are handled differently. This chapter explains how to write programs to take advantage of special MS-DOS features, and how to use those features once your program is completed.

5.2 Passing Command-Line Data to a Program

Your C program can access data from a command line or from the environment table. You can use the MS-DOS **SET** or **PATH** command to place data in the environment table. See Section 2.7, "Setting Up the Environment," in Chapter 2, "Getting Started," for a discussion of environment variables. Command-line data are arguments that appear on the same line as the program name when you execute the program.

To pass data to your program on the command line, give one or more arguments after the program name when you execute the program. 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, if your C program is called TEST.EXE, you might give it the following command line:

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

In this case, the program will be executed and three arguments will be passed: 42, de f, and 16.

MS-DOS stores the command-line arguments in the MS-DOS program header. The C run-time library (which becomes part of your program during linking) in turn stores each argument from the program header as a null-terminated string in an array of strings. MS-DOS limits the combined length of all arguments on the command line (including the program name) to 128 bytes. If you provide a longer command line, additional characters will be ignored.

In order for a C program to read and use the data from the command line or from the environment table, the program should declare three variables as arguments to the **main** function. These three variables and their contents are listed in Table 5.1.

Table 5.1
Argument Variables

Variable	Contents
argc	Number of arguments passed
argv	Array of strings containing arguments
envp	Pointer to environment table

By declaring these variables as arguments to **main**, you make them available as local variables in the **main** function. The example below illustrates how to declare these arguments:

```
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. Therefore, if you want to use the **envp** arguments, you must declare **argc** and **argv**, even if you do not use them.

The number of arguments appearing on the command line is passed as the integer variable **argc**, and the command line is passed to the program as the array of strings pointed to by **argv**.

The first argument of any command line is the name of the program to be executed. Therefore, the program name is the first string stored in **argv**, at **argv[0]**. Since a program name must be given in order to run the program, the integer value of **argc** is always at least 1. Therefore, if you pass two arguments to your program, **argc** will have a value of 3 (two arguments and the program name).

The first argument following the program name is stored at **argv[1]**, the second is stored at **argv[2]**, and so on, to the last argument.

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 string C. Under MS-DOS versions 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. Using the **putenv** routine may change the location of the environment table in memory, depending on memory requirements. Therefore, the value given to **envp** at the beginning of the program execution may not be valid throughout the program's execution. In contrast, the **putenv** and **getenv** routines access the environment table properly, even when its location changes. These routines use the global variable **environ** (described in the *Microsoft C Compiler 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.

5.2.1 Expanding Wild-Card Arguments

You can use the MS-DOS wild-card characters, the question mark (?) and the asterisk (*), to specify file-name and path-name arguments on the command line. To prepare for using wild cards, you must link your object file with one of the **xSETARGV.OBJ** object files (where the value of *x* depends on the memory model you have selected).

These object files are included with your compiler software. If you don't link with one of these, your program will not expand wild-card characters on the command line, interpreting them instead as literal question marks and asterisks.

The **xSETARGV.OBJ** files expand 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-quotation 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 your programs frequently expand wild-card characters, you may want to put the wild-card routines (**xSETARGV.OBJ**) in the appropriate standard C libraries (**xLIBC.LIB**) so they will be linked with your program automatically. To do this, use the Microsoft Library Manager (**LIB**) to extract the module named **stdargs** from the library (the module name is

the same in all four libraries) and insert `xSETARGV`. When you replace `stdargv` with the appropriate routine, wild-card expansions will always be performed on command-line arguments. **LIB** is described in Chapter 6, "Managing Libraries."

Example

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

In this example, `SSETARGV.OBJ`, which is in the directory `\LIB`, is linked with `BETA.OBJ`, producing the executable file `BETA.EXE`. When `BETA.EXE` is executed, the wild-card character `*` is expanded, causing all file names 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 quotation marks to be represented literally, so the argument `"HELLO"` (including the quotation marks) is passed.

5.2.2 Suppressing Command-Line Processing

If your program does not take command-line arguments, you can achieve a small space saving 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.

If your program makes calls to the `spawn` or `exec` routines in the C run-time library, you should not suppress the environment-processing routine, since this routine is used to pass an environment from the parent process to the child process.

Example

```
_setargv()  
{  
}  
  
_setenvp()  
{  
}
```

The example above 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.

5.3 Returning an Exit Code

Your program can return an exit code (sometimes called a return code) as a means of leaving a message for MS-DOS. The exit code can then be used by MS-DOS batch files or other programs that test exit codes (`MAKE`, for example). Exit codes and their uses are discussed in more detail in Appendix E, "Using Exit Codes."

Exit codes are returned through the `main` function. This function, like any other C function, can return a value. The value is of `int` type, and is passed to MS-DOS as the exit code of the executed program. This exit code can be checked with the `IF ERRORLEVEL` command in MS-DOS batch files. (See your MS-DOS user's guide for more information on using batch files.)

To cause the `main` function to return a specific value to MS-DOS, you should use a `return` statement or `exit` function to specify the value to be returned. For example, if the `main` function in a program terminates with either the statement `return (6);` or `exit (6);` the value 6 will be returned to MS-DOS. If neither of these methods is used, the return code is undefined.

Example

```
#define TRUE    1
#define FALSE   0

int error = FALSE;

main()
{
    .
    .
    .
    if (error) return (1);
    else return (0);
}
```

In the example above, the value 1 would be returned if the variable `error` were set to `TRUE` somewhere within the body of the program. Otherwise, 0 would be returned to MS-DOS. The example program follows the convention of returning 0 if the program is successful, and some larger number if an error is encountered.

5.4 Suppressing Null-Pointer Checks

When you execute your C program, a special error-checking routine is automatically invoked after your program has terminated to determine whether the contents of the **NULL** segment have changed, and 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. Note 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.

Note

The message

Null pointer assignment

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 **chks** in all four standard libraries (**xLIBC.LIB**). Do not remove the routine entirely or there will be an unresolved reference in your program. Instead, use **LIB** (described in Chapter 6, "Managing Libraries") to replace the **chks** 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	141
6.2	Overview of LIB Operation	142
6.3	Running LIB	143
6.3.1	“Library name” Prompt	144
6.3.2	“Operations” Prompt	144
6.3.3	“List file” Prompt	146
6.3.4	“Output library” Prompt	146
6.3.5	Using the Command Line	147
6.3.6	Using a Response File	148
6.3.7	Extending Lines	149
6.3.8	Terminating the Library Session	150
6.3.9	Selecting Default Responses to Prompts	150
6.4	Library Tasks	150
6.4.1	Creating a Library File	150
6.4.2	Modifying a Library File	151
6.4.3	Adding Library Modules	151
6.4.4	Deleting Library Modules	152
6.4.5	Replacing Library Modules	152
6.4.6	Copying Library Modules	152
6.4.7	Moving Library Modules	153
6.4.8	Combining Libraries	153
6.4.9	Creating a Cross-Reference Listing	153
6.4.10	Performing Consistency Checks	154
6.4.11	Setting the Library-Page Size	154

6.1 Introduction

The Microsoft Library Manager (**LIB**) 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, **LIB** can read the contents of 286 XENIX archives and Intel-style libraries and combine their contents with MS-DOS libraries. You can add the contents of a 286 XENIX archive or an Intel-style library to an MS-DOS library by using the add 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 path name, including a drive designation, directory path name, and file-name extension (usually **.OBJ**). Object modules have only a name. For example, B:\RUN\SORT.OBJ is an object-file name, 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 alternative methods of invoking **LIB**, described in sections 6.3.5 and 6.3.6. 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 **LIB**, including the following tasks:

- 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** 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. Therefore, 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**.

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 file names 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** to use the responses in that file. You may find it easier to use the prompt method until you become comfortable with the **LIB** commands and operations.

6.3.1 “Library name” Prompt

At the “Library name” prompt, give the name of the library file you want. You can also specify a page size at this prompt using the **/PAGESIZE** option.

Usually library files are named with the **.LIB** extension. You can omit the **.LIB** extension when you give the library-file name since **LIB** assumes that the file-name extension is **.LIB**. If your library file does not have the **.LIB** extension, be sure to include the extension when you give the library-file name. Otherwise, **LIB** cannot find the file.

Path names are allowed with the library-file name. You can give **LIB** the path name of a library file in another directory or on another disk.

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

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

Library file does not exist. Create?

Type **y** to create the library file, or **n** to terminate **LIB**. If you type a library-file name 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.

If you wish to set the library-page size, you must enter the **/PAGESIZE** option at the “Library name” prompt. It must follow the library name. See Section 6.4.11, “Setting the Library-Page Size,” for details.

6.3.2 “Operations” Prompt

At the “Operations” prompt, you can type one of the command symbols for manipulating modules (+, -, -+, *, or -*), followed immediately by a module name or an object-file name. You can specify more than one operation at this prompt, in any order. The default for the “Operations” prompt is no change.

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, then press the RETURN key. The ampersand must follow a file name; 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 the command symbols and their meanings and uses:

Symbols	Meaning
+	<p>The plus sign makes an object file the last module in the library file. Immediately following the plus sign, give the name of the object file. You can use path names for the object file. LIB automatically supplies the .OBJ extension, so you can omit the extension from the object-file name.</p> <p>You can also use the plus sign to combine two libraries. When you give a library name following 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-file name. Otherwise, LIB uses the default .OBJ extension when it looks for the file.</p>
-	<p>The minus sign deletes a module from the library file. Immediately following the minus sign, give the name of the module to be deleted. A module name has no path name and no extension.</p> <p>Type a minus sign followed by a plus sign to replace a module in the library. Following the replacement symbol, give the name of the module to be replaced. Module names have no path names and no extensions.</p>
- +	<p>To replace a module, LIB 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.</p>

*

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 path name of the current working directory to the module name to form a complete object-file name. You cannot override the **.OBJ** extension, drive designation, or path name 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

At the "List file" prompt, you can give a file name for a cross-reference listing file. You can specify a full path name 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 public symbols in the library. Each symbol name is followed by the name of the module in which it is referenced.

The second list consists 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 file name **NUL**, which tells **LIB** *not* to create a listing file.

6.3.4 "Output library" Prompt

At the "Output library" prompt you can give the name of a new library file that will have the specified modifications. This prompt appears only if you specify modifications to the library at the "Operations" prompt. The default is the current library-file name. If you do not specify a new library-file

name, the original, unmodified library is saved in a library file with the same name but with a **.BAK** extension replacing the **.LIB** extension.

6.3.5 Using the Command Line

The command-line method of starting **LIB** has the following form:

```
LIB oldlibrary [[/PAGESIZE:number] [commands] [,[[listfile]] [,[[newlibrary]]]] [;]
```

The entries following **LIB** correspond to responses to the **LIB** command prompts. The *newlibrary* entry and the optional *number* for the **/PAGESIZE** option correspond to the “Library name” prompt. If you want **LIB** to perform a consistency check on the library, follow the *newlibrary* entry with a semicolon (;).

The *commands* entries are any of the commands allowed at the “Operations” prompt. The *listfile* 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 *commands* entry by a comma. If you give a file name in the new library field, the library name must be separated from the listing-file name or the last *commands* entry by a comma.

To tell **LIB** to use the default responses for the remaining entries, use a semicolon after any entry except the first. The semicolon should be the last character on the command line.

Examples

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

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

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

```
LIB C;
```

```
LIB LANG,LCROSS.PUB
```

```
LIB FIRST -*STUFF *MORE, ,SECOND
```

The first three examples have the same effect. The first example uses the replace command (-+) to instruct **LIB** to replace the HEAP module in the library LANG.LIB. **LIB** deletes the HEAP module from the library, then appends the object file HEAP.OBJ as a new module in the library. The semicolon 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 to the original library file instead of creating a new library file.

The next two examples do the same thing, but in two separate operations, using the add (+) and delete (-) commands. The effect is the same for the second and third examples because delete operations are always carried out before added operations, regardless of the order of the operations in the command line. This order of execution prevents confusion when a new version of a module replaces an old version in the library file.

The fourth example 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 fifth example tells **LIB** to perform a consistency check of the library file LANG.LIB, then create a cross-reference listing file named LCROSS.PUB.

The last example instructs **LIB** to move the module STUFF from the library FIRST.LIB to an object file called STUFF.OBJ. The module STUFF is removed from the library in the process. The module MORE is copied from the library to an object file called MORE.OBJ. It remains in the library. The revised library is called SECOND.LIB. It contains all the modules in FIRST.LIB except STUFF, which was removed by the move (-*) command. The original library, FIRST.LIB, remains unchanged.

6.3.6 Using a Response File

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

LIB @*filename*

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

You can also enter the name of a response file after any of the linker prompts, or at any position in a command line. The input from the response file will be treated exactly as if it had been entered after prompts or in command lines, with a carriage-return-line-feed combination in the response file treated the same as a RETURN key in response to a prompt, or a comma in a command line.

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 use responses typed on the keyboard.

When you run **LIB** with a response file, the prompts are displayed 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 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-file name; 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 more operations. With the response-file method, you can use the ampersand at the end of a line and continue typing operations on the next line.

6.3.8 Terminating the Library Session

You can press CONTROL-C at any time during a library session to terminate the session and return to MS-DOS. If you notice that you have entered an incorrect response at a previous prompt, you should press CONTROL-C to exit **LIB** and begin again. You can use the normal MS-DOS editing keys to correct errors at the current prompt.

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 file name **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 at 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, give the name of the library file you want to create at the "Library name" prompt. **LIB** supplies the **.LIB** extension.

The name of the new library must not be the name of an existing file. If it is, **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 **y** to create the file, or **n** 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 6.4.11, “Setting the Library-Page Size,” 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 (+) at the “Operations” prompt. You can also add the contents of another library, if you wish. These options are discussed in Section 6.4.3, “Adding Library Modules,” and Section 6.4.8, “Combining Libraries.”

6.4.2 Modifying a Library File

You can modify an existing library file by giving the name of the library file at the “Library name” prompt. All operations you specify at 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 at the “Output library” prompt. The modified library file is stored under the new library-file name, while the original library file remains unchanged.

If you don’t give a file name at 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 with 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 (+) at 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 following the plus sign.

LIB strips the drive designation and the extension from the object-file specification, leaving only the base name. This becomes the name of the object module in the library. For example, if the object file `B:\CURSOR` 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 (-) at the “Operations” prompt to delete an object module from a library. Following the minus sign, give the name of the module to be deleted. A module name has no path name and no extension; it is simply a name, such as `CURSOR`.

6.4.5 Replacing Library Modules

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

To replace a module, **LIB** 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 Copying 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 path name of the current working directory to the module name to form a complete object-file name. You cannot override the `.OBJ` extension, drive designation, or path name given to the object file, but you can later rename the file or copy it to whatever location you like.

6.4.7 Moving Library Modules

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, then deleting the module from the library.

6.4.8 Combining Libraries

You can add the contents of a library to another library by using the plus sign (+) with a library-file name instead of an object-file name. At 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-file name. 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. Therefore, 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. Note 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 at 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. The original library is saved with the extension **.BAK**.

6.4.9 Creating a Cross-Reference Listing

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

You can give the listing file any name and any extension. To cause the listing file to be created outside your current working directory, you can specify a full path name, including drive designation. **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.

6.4.10 Performing Consistency Checks

When you give only a library name followed by a semicolon at 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 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, then creates the cross-reference listing.

6.4.11 Setting the Library-Page Size

You can set the library-page size by adding a page-size option after the library-file name in the **LIB** command line or after the new library-file name at the "Library name" prompt. The option has the following form:

/PAGESIZE:number

The *number* specifies the new page size. It must be an integer value representing a power of 2 between the values 16 and 32768. The option name can be abbreviated to **/P:number**.

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

Note

Because of the indexing technique used by **LIB**, a library with a large page size can hold more modules than a library with a smaller page size. However, for each module in the library, an average of *number*/2 bytes of storage space is wasted (where *number* is the page size). In most cases, a small page size is advantageous; you should use a small page size unless you need to put a very large number of modules in a library.

Another consequence of this indexing technique is that the page size determines the maximum possible size of the **.LIB** file. Specifically, this limit is *number* * 65536. For example, /P:16 means that the **.LIB** file has to be smaller than 1 megabyte (16 * 65536 bytes).

(

(

(

Chapter 7

Maintaining Programs with MAKE

7.1	Introduction	159
7.2	Using MAKE	159
7.2.1	Creating a MAKE Description File	159
7.2.2	Starting MAKE	161
7.2.3	Using MAKE Options	162
7.2.4	Using Macro Definitions	163
7.2.5	Nesting Macro Definitions	164
7.2.6	Using Special Macros	165
7.2.7	Inference Rules	165
7.3	Maintaining a Program: an Example	167

(

)

)

7.1 Introduction

The Microsoft Program Maintenance Utility (**MAKE**) automates the process of maintaining high-level-language programs. **MAKE** automatically carries out all tasks needed to update a program after one or more of its source files has been changed.

Unlike many batch-processing programs, **MAKE** compares the last modification date of the file or files that may need updating with the modification dates of files on which these target files depend. **MAKE** then carries out the given task only if a target file is out of date. **MAKE** does not compile and link all files just because one file has been updated. This can save time when creating programs that have many source files or take several steps to complete.

The rest of this chapter explains how to use **MAKE** and illustrates how to maintain a sample C program.

7.2 Using **MAKE**

To use **MAKE**, you must create a **MAKE** description file that defines the tasks you wish to accomplish and specifies the files on which these tasks depend. Once the description file exists, invoke **MAKE** and supply the file name as a parameter. **MAKE** then reads the contents of the file and carries out the requested tasks.

The following sections explain how to create a **MAKE** description file and start **MAKE**.

7.2.1 Creating a **MAKE** Description File

You can create a **MAKE** description file with a text editor. A **MAKE** description file consists of one or more target/dependent descriptions. Each description has the following general form:

```
targetfile : dependentfiles [[# comment]]  
[[# comment]]  
    command [[# comment]]  
    [[command]] [[# comment]]  
.  
.
```

In this format, *targetfile* is the name of a file that may need updating, *dependentfiles* are the names of any files on which the target file depends, and *command* is the name of an executable file or MS-DOS internal command.

The *targetfile* and *dependentfile* must be valid file names. A path name must be provided for any file that is not on the same drive and directory as the description file.

Any number of dependent files can be given, but only one target name is allowed. Dependent-file names must be separated by at least one space. If you have more dependent files than can fit on one line, you can continue the names on the next line by typing a backslash (\) followed by a new line.

The *command* can be any valid MS-DOS command line consisting of an executable-file name or an MS-DOS internal command. Any number of commands can be given, but each must begin on a new line and must be preceded by a TAB, or by at least one space. The commands are carried out only if one or more of the dependent files has been modified since the target file was created.

One way to remember the **MAKE** description file format is to think of it as an "if-then" statement in the following format: If a *dependentfile* is older than the *targetfile*, or a *dependentfile* does not exist, then do *commands*.

You can give any number of target/dependent descriptions in a description file. You must make sure, however, that the last line in one description is separated from the first line of the next description by at least one blank line.

The number sign (#) is a comment character. All characters on the same line that follow the comment character are ignored. When comments appear in a *command lines* section, the comment character (#) must be the first character on the line (no leading white space). On any other lines, the comment character can appear anywhere.

Note

The order in which you place the target/dependent descriptions is important. **MAKE** examines each description in turn and makes its decision to carry out a given task based on the file's current modification date. If a command in a later description modifies a file, **MAKE** has no way to return to the description in which that file is a target.

Example

```
STARTUP.OBJ: STARTUP.C
              MSC STARTUP,,STARTUP;

PRINT.OBJ:    PRINT.C #Comment allowed after dependent
#Comment before command must start in first column
              MSC PRINT,,PRINT; #Comment allowed after command

PRINT.EXE:    STARTUP.OBJ PRINT.OBJ
              LINK STARTUP+PRINT,PRINT,PRINT;
```

This example defines the actions to be carried out to create three target files. Each file has at least one dependent file and one command. The target descriptions are given in the order in which the target files will be created. Thus **STARTUP.OBJ** and **PRINT.OBJ** are examined and created, if necessary, before **PRINT.EXE**.

Note that a comment can appear on the same line as the target/dependent description line and the command line. However, when the comment appears on a separate line, the comment character (#) must be the first character on the line.

7.2.2 Starting **MAKE**

MAKE must be started with a command line. You cannot use prompts. The **MAKE** command line has the following form:

MAKE *[options]* *[macrodefinitions]* *filename*

The *options* are one or more of the options described in Section 7.2.3. The *macrodefinitions* are one or more macro definitions, as described in Section 7.2.4. The *filename* is the name of a **MAKE** description file. By convention,

a **MAKE** description file has the same file name (but with no extension) as the program it describes; however, *filename* can be any valid file name you choose.

Once you start **MAKE**, it examines each target description in turn. If a given target file is out of date with respect to its dependent file, or if the file does not exist, **MAKE** executes the given command or commands. Otherwise, it skips to the next target/dependent description.

When **MAKE** finds an out-of-date dependent file, it displays the command or commands from the target/dependent description, then executes the commands. If **MAKE** cannot find a specified file, it displays a message informing you that the file was not found. If the missing file is a target file, **MAKE** continues execution, since the missing file will, in many cases, be created by subsequent commands.

If the missing file is a dependent file or command file, **MAKE** stops execution of the description file. **MAKE** also stops execution and displays the exit code if the command returns an error.

When **MAKE** executes a command, it uses the same environment used to invoke **MAKE**. Thus environment variables such as **PATH** are available for these commands.

7.2.3 Using **MAKE** Options

The options available with the **MAKE** command modify its behavior as described below:

Option	Action
/D	Displays the last modification date of each file as the file is scanned
/I	Ignores exit codes (also called return or "errorlevel" codes) returned by programs called from the MAKE description file; MAKE continues execution of the subsequent lines of the description file despite the errors
/N	Displays commands that would be executed by a description file, but does not execute the commands
/S	Executes in "silent" mode; that is, lines are not displayed as they are executed

7.2.4 Using Macro Definitions

Macro definitions let you associate a symbolic name with a particular value. By using macro definitions, you can change values used in the description file without having to edit every line that uses a particular value.

The form of a macro definition is:

name=*value*

The form for using a previously defined macro definition is:

`$(name)`

Occurrences of the pattern `$(name)` in the description file are replaced with the specified *value*. The *name* is converted to uppercase; flags and FLAGS are equivalent. If you define a macro name but leave the *value* blank, the *value* will be a null string.

Macro definitions can be placed in the **MAKE** description file or given on the **MAKE** command line. A *name* is also considered defined if it has a definition in the current environment. For example, if the environment variable **PATH** is defined in the current environment, occurrences of `$ (PATH)` in the description file will be replaced with the **PATH** value.

In the **MAKE** description file, each macro definition must appear on a separate line. Any white space (tab and space characters) between *name* and the equal sign (=) or between the equal sign and *value* is ignored. Any other white space is considered part of *value*. To include white space in a macro definition on the command line, enclose the entire definition in double quotation marks ("").

If the same name is defined in more than one place, the following order of precedence applies:

1. Command line definition
2. Description file definition
3. Environment definition

Example

```
base=ABC
warn="/W 2"

$(base) .OBJ:      $(base) .C
    MSC $(base) $(warn),$(base),$(base);

$(base) .exe:      $(base) .obj \lib\math.lib
    LINK $(base),$(base),$(base);
```

The sample **MAKE** description file above shows macro definitions for the names base and warn. **MAKE** replaces each occurrence of \$(base) with ABC. If the description file is called COMPILE, you can give the following command:

```
MAKE base=DEF compile
```

This command line enables you to override the definition of base in the description file, causing DEF to be compiled and linked instead of ABC.

If you want to override the warning level 2 specified by the macro warn in the **MAKE** description file and use the **MSC** default (warning level 1) instead, you could start **MAKE** with the following command line:

```
MAKE warn= COMPILE
```

Since the value for warn is blank, it will be treated as a null string. Since the null string was given from the command line, which has higher precedence than the definition in the description file, warn will be expanded to a null string and no option will be passed in the **MSC** command line.

7.2.5 Nesting Macro Definitions

Macro definitions can be nested. In other words, a macro definition can include another macro definition. For example, you could have the following macro definition in the **MAKE** description file PICTURE:

```
LIBS=$(DLIB)\MATH.LIB $(DLIB)\GRAPHICS.LIB
```

You could then start **MAKE** with the following command line:

```
MAKE DLIB=D:\LIB PICTURE
```

In this case, every occurrence of the macro LIBS in the description file would be expanded to the following:

D:\LIB\MATH.LIB D:\LIB\GRAPHICS.LIB

Be careful to avoid infinitely recursive macros such as the following:

```
A = $(B)
B = $(C)
C = $(A)
```

7.2.6 Using Special Macros

MAKE recognizes three special macro names and will automatically substitute a value for each. The special names and their values are as follows:

Name	Value Substituted
\$*	Base name portion of the target (without the extension)
\$@	Complete target name
\$**	Complete list of dependencies

These macro names can be used in description files, as shown in the following example.

Example

```
TEST.EXE: MOD1.OBJ MOD2.OBJ MOD3.OBJ
          LINK $**, $@;
          $*
```

The example above is equivalent to the following:

```
TEST:EXE: MOD1.OBJ MOD2.OBJ MOD3.OBJ
          LINK MOD1.OBJ MOD2.OBJ MOD3.OBJ, TEST.EXE;
          TEST
```

7.2.7 Inference Rules

MAKE allows you to create inference rules that specify commands for target/dependent descriptions even when there is no explicit command in the **MAKE** description file. An inference rule is a way of telling **MAKE** how to produce a file with one type of extension from a file with the same base name and another type of extension.

For example, if you define a rule for producing **.OBJ** files from **.C** files, then the actual commands do not have to be repeated in the description file for each target/dependent description.

Inference rules take the following form:

```
.dependentextension.targetextension :  
    command  
    [[command]]  
    .  
    .  
    .
```

For lines that do not have explicit commands, **MAKE** looks for a rule that matches both the target's extension and the dependent's extension. If it finds such a rule, **MAKE** performs the commands given by the rule.

MAKE looks first for dependency rules in the current description file, but if it does not find an appropriate rule, it will search for **TOOLS.INI**, the tools-initialization file. **MAKE** looks for **TOOLS.INI** on the current drive and directory, then searches any directories specified with the MS-DOS **PATH** command.

If **MAKE** finds **TOOLS.INI**, it looks through the file for a line beginning with the tag [**make**], which must come at the beginning of the line. Inference rules following this line will be applied if appropriate.

Example

```
[make]  
.C.OBJ:  
    MSC $*.C,,,;  
  
TEST1.OBJ: TEST1.C  
  
TEST2.OBJ: TEST2.C  
    MSC TEST2.C;
```

In the sample description file above, an inference rule is defined in the first line. The file name in the rule is specified with the special macro name **\$*** so that the rule will apply to any base name. When **MAKE** encounters the dependency for files **TEST1.OBJ** and **TEST1.C** it looks first for commands

on the next line. When it does not find any, **MAKE** checks for a rule that may apply and finds the rule defined in the first lines of the description file. **MAKE** applies the rule, replacing the \$* macro with TEST1 when it executes the command, producing the following message:

```
MSC TEST1.C...;
```

When **MAKE** reaches the second dependency for the TEST2 files, it does not search for a dependency rule, since a command is explicitly stated for this target/dependent description.

7.3 Maintaining a Program: an Example

MAKE is especially useful for programs in development, because it offers a quick way to recreate a modified program after small changes.

Consider a test program called WORK.EXE that is made from two source files, WORK1.C and WORK2.C. Both source files use an include file called WORK.H, and both modules must be linked with routines in a library file called MATH.LIB. During development, you will often want to compile and link to create WORK.EXE, but you won't always want to recompile all the files. You only want to recompile the source files that have changed.

The following target/dependent descriptions copied to the **MAKE** description file WORK will carry out the appropriate tasks:

```
WORK.EXE:    WORK.H
             MSC /Zi WORK1...;
             MSC /Zi WORK2...;

WORK1.OBJ:   WORK1.C
             MSC /Zi WORK1...;

WORK2.OBJ:   WORK2.C
             MSC /Zi WORK2...;

WORK.EXE:   WORK1.OBJ WORK2.OBJ \LIB\MATH.LIB
             LINK WORK1+WORK2,WORK,WORK,\LIB\MATH.LIB /CO
```

After each session of debugging and editing source files, start **MAKE** with the following command line:

```
MAKE WORK
```

MAKE carries out the following steps:

1. Checks to see if WORK.H has been changed since the last time WORK.EXE was created by the linker. If the include module has been changed, then both source files must be recompiled. If the include module was not changed, **MAKE** skips to the next dependency.
2. Checks to see if WORK1.C has been changed since the last time WORK1.OBJ was created by the compiler. If so, WORK1.C will be recompiled.
3. Checks WORK2.C in the same way WORK1.C was checked in step 2. Note that if only one of the source files has been changed, only that file will be recompiled. However, if both source files have been recompiled in step 1, then they are not recompiled in this step.
4. Checks to see if either of the object files have been changed since the last time the modules were linked. If one or both of the files were recompiled, the program will be relinked. The program will also be relinked if the library file MATH.LIB has been changed since the last time the program was linked.

When the source files are created, **MAKE** compiles and links both source files, since none of the target files exists. If you invoke **MAKE** again without changing any of the dependent files, all commands will be skipped. If you change one of the source files, that file will be recompiled and the program will be relinked. If you change the library file MATH.LIB, but make no other changes, **MAKE** will skip the commands in the first three dependencies, but will relink the program as specified in the last dependency.

Chapter 8

Working with Memory Models

8.1	Introduction	171
8.2	Using the Standard Memory Models	173
8.2.1	Creating Small-Model Programs	174
8.2.2	Creating Medium-Model Programs	175
8.2.3	Creating Compact-Model Programs	175
8.2.4	Creating Large-Model Programs	176
8.2.5	Creating Huge-Model Programs	176
8.3	Using the near, far, and huge Keywords	177
8.3.1	Library Support for near, far, and huge	179
8.3.2	Declaring Data with near, far, and huge	179
8.3.3	Declaring Functions with near and far	181
8.3.4	Pointer Conversions	183
8.4	Creating Customized Memory Models	185
8.4.1	Code Pointers	187
8.4.2	Data Pointers	187
8.4.3	Setting Up Segments	188
8.4.4	Library Support for Customized Memory Models	189

)

1

)

1

)

1

8.1 Introduction

You can gain greater control over how your program uses memory by specifying the memory model for the program. If you do not specify a memory model, MSC uses the small memory model by default. The small memory model is sufficient for most programs.

You cannot use the small memory model if your program satisfies one or more of the following three conditions:

1. Your program has more than 64K of code.
2. Your program has more than 64K of data.
3. Your program contains individual arrays that need to be larger than 64K.

Advanced programmers may have other reasons for using a model other than the small memory model.

If you decide that the small memory model will not be adequate for your program, you have four options for larger memory models:

1. You can specify one of the other standard memory models (medium, compact, large, or huge) using the **/Aletter** option.
2. You can create a mixed-model program using the **near**, **far**, and **huge** keywords.
3. You can create your own customized memory model using the **/Astring** option.
4. Method 2 can be combined with either method 1 or method 3.

The terms “near,” “far,” and “huge” are crucial to understanding the concept of memory models. These terms indicate how data can be accessed in the segmented architecture of the 8086 family of microprocessors.

The MS-DOS operating system loads the code and data allocated by your program into “segments” in physical memory. Each segment is up to 64K long. Since separate segments are always allocated for the program code and data, the minimum number of segments allocated for a program is two; these two segments, required for every program, are called the default segments. The small memory model uses only the two default segments. The other memory models discussed in this chapter allow more than one code segment and/or data segment per program.

In the 8086/80286 family of microprocessors, all memory addresses consist of two parts:

1. A 16-bit number that represents the base address of a memory segment
2. Another 16-bit number that gives an offset within that segment

The architecture of the 8086 microprocessor is such that code can be accessed within the default code or data segment using just the 16-bit offset value. This is possible because the segment addresses for the default segments are always known. This 16-bit offset value is called a “near” address, and can be accessed with a “near” pointer. Since only 16-bit arithmetic is required to access any near item, near references to code or data are smaller and more efficient.

When data or code lie outside the default segments, the address must use both the segment and offset values. Such addresses are called “far” addresses, and can be accessed by using “far” pointers in a C program. Accessing far data or code items is more expensive in terms of program speed and size, but their use allows your programs to address all memory, rather than just a 64K piece.

There is a third type of address in Microsoft C, the “huge” address. A huge address is similar to a far address in that both consist of a segment value and an offset value but they differ in the way address arithmetic is performed on pointers. Because items (both code and data) referenced by far pointers are still assumed to lie completely within the segment in which they start, pointer arithmetic is done only on the offset portion of the address. This gain in pointer arithmetic efficiency is achieved, however, by limiting the size of any single item to 64K. With data items, huge pointers overcome this size limitation: pointer arithmetic is performed on all 32 bits of the data item’s address, thus allowing data items referenced by huge pointers to span more than one segment, provided they adhere to the limitations outlined in Section 8.2.5, “Creating Huge-Model Programs.”

The rest of this chapter deals with the various methods you can use to control whether your program makes far, near, or huge calls to access code or data.

8.2 Using the Standard Memory Models

The Microsoft C Compiler package includes four standard libraries that support five standard memory models. Using the standard memory models is the simplest way to control how your program accesses code and data in memory.

When you use the standard memory models, the compiler handles library support for you. The library corresponding to the memory model you specify is used automatically. Each memory model has its own library, except for the huge memory model, which uses the large-model library.

The advantage of using standard models for your programs is simplicity. In the standard models, memory management is specified by compiler options; since the standard models do not require the use of extended keywords, they are the best way to write code that can be ported to other systems (particularly systems that do not use segmented architectures).

The disadvantage of using standard memory models exclusively is that they may not produce the most efficient code. For example, if you have an otherwise small-model program containing a large array that pushes the total data size for your program over the 64K limit for small model, it may be to your advantage to declare the one array with the **far** keyword, while keeping the rest of the program small model, as opposed to using the standard compact memory model for the entire program. For maximum flexibility and control over how your program uses memory, you can combine the standard-memory-model method with the **near**, **far**, and **huge** keywords described in Section 8.3.

The **/Aletter** option for **MSC** (or **CL**) is used to specify one of the five standard memory models (small, medium, compact, large, or huge) at compile time. These options are discussed in the next five sections.

Note

In the following sections, which describe in detail the different memory-model addressing conventions, it is important to keep in mind two common features of all five models:

1. No *single* source module can generate 64K or more of code.
 2. No *single* data item can exceed 64K, unless it appears in a huge-model program, or it has been declared with the **huge** keyword.
-

8.2.1 Creating Small-Model Programs

Option

/AS

The small-model option tells the compiler to create a program that occupies the two default segments: one for code and one for data.

Small-model programs are typically C programs that are short or have a limited purpose. Since code and data for these programs are each limited to 64K, the total size of a small-model program can never exceed 128K. Most programs fit easily into this model.

The default in small-model programs is that both code and data items are accessed with near addresses. You can override the default for data by using the **far** or **huge** keywords, and the default for code by using the **far** keyword (**huge** is relevant only for data items—specifically, arrays and pointers to arrays).

The compiler creates small-model programs by default when you do not specify a program model. The **/AS** option is provided for completeness; you need never give it explicitly.

8.2.2 Creating Medium-Model Programs

Option

/AM

The medium-model option provides a single segment for program data, and multiple segments for program code. Each source module is given its own code segment.

Medium-model programs are typically C programs that have a large number of program statements (more than 64K of code), but a relatively small amount of data (less than 64K). Program code can occupy any amount of space and is given as many segments as needed; total program data cannot be greater than 64K. The medium model provides a useful trade-off between speed and space, since most programs refer more frequently to data items than to code.

8.2.3 Creating Compact-Model Programs

Option

/AC

The compact-model option directs the compiler to allow multiple segments for the data of the program. Only one segment is created for the program code.

Compact-model programs are typically C programs that have a large amount of data, but a relatively small number of program statements. Program data can occupy any amount of space and are given as many segments as needed.

The default in compact-model programs is that code items are accessed with near addresses and data items are accessed with far addresses. You can override the default by using the **near** and **huge** keywords for data, and the **far** keyword for code.

8.2.4 Creating Large-Model Programs

Option

/AL

The large-model option allows the compiler to create multiple segments as needed for both code and data.

Large-model programs are typically very large C programs that use a large amount of data storage during normal processing.

The default in large-model programs is that both code and data items are accessed with far addresses. You can override the default by using the **near** and **huge** keywords for data, and the **near** keyword for code.

8.2.5 Creating Huge-Model Programs

Option

/AH

The huge-model option is similar to the large-model option, except that the restriction on the size of individual data items is removed for arrays.

Some size restrictions apply to elements of huge arrays where the array is larger than 64K, however. To provide efficient addressing, array elements are not permitted to cross segment boundaries. This has the following implications:

1. No array element can be larger than 64K.
2. For any array larger than 128K, all elements must have a size in bytes equal to a power of 2 (for example, 2 bytes, 4 bytes, 8 bytes, 16 bytes, and so on). However, if the array is 128K or smaller, its elements may be any size, up to and including 64K.

In huge-model programs, care must be taken when using the **sizeof** operator or when subtracting pointers. The C language defines the value returned by the **sizeof** operator to be an **int** value, but the size in bytes of a huge array is a **long int** value. To solve this discrepancy, the Microsoft C Compiler produces the correct size of a huge array when the following type cast is used:

```
(long) sizeof (huge_item)
```

Similarly, the C language defines the result of subtracting two pointers as an **int** value. When subtracting two huge pointers, however, the result may be a **long int** value. The Microsoft C Compiler gives the correct result when the following type cast is used:

```
(long) (huge_ptr1 - huge_ptr2)
```

8.3 Using the near, far, and huge Keywords

One limitation of the predefined memory-model structure is that, when you change memory models, all data and code address sizes are subject to change. However, the Microsoft C Compiler lets you override the default addressing convention for a given memory model and access items with either a near, far, or huge pointer. This is done with the **near**, **far**, or **huge** keywords. These special type modifiers can be used with a standard memory model to overcome addressing limitations for particular items (either data or code) without changing the addressing conventions for the program as a whole. Table 8.1 explains how the use of these keywords affects the addressing of code or data, or pointers to code or data.

Table 8.1**Addressing of Code and Data Declared with near, far, and huge**

Keyword	Data	Function	Pointer Arithmetic
near	Resides in default data segment; referenced with 16-bit address (pointer to data is 16 bits)	Assumed to be in current code segment; referenced with 16-bit address (pointer to function is 16 bits)	Uses 16 bits
far	May be anywhere in memory, not assumed to reside in current data segment; referenced with 32-bit address (pointer to data is 32 bits)	Not assumed to be in current code segment; referenced with 32-bit address (pointer to function is 32 bits)	Uses 16 bits
huge	May be anywhere in memory, not assumed to reside in current data segment; individual data items (arrays) can exceed 64K in size; referenced with 32-bit address (pointer to data is 32 bits)	Not applicable to code	Uses 32 bits for data

The **near**, **far**, and **huge** keywords are not a standard part of the C language; they are meaningful only for systems that use a segmented architecture similar to that of the 8086 family of microprocessors. Keep this in mind if you want your code to be ported to other systems.

In the Microsoft C Compiler, the **near**, **far**, and **huge** keywords are enabled by default. To treat these keywords as ordinary identifiers, you must give the **/Za** option at compile time. This option is useful if you are concerned with porting C programs from environments in which these are not keywords; for instance, a program might have been written using one of these words as a label.

8.3.1 Library Support for near, far, and huge

When using the **near**, **far**, and **huge** keywords to modify addressing conventions for particular items, you can usually use one of the standard libraries (small, compact, medium, or large) with your program. The large-model libraries are also appropriate for use with huge-model programs. However, you must use care when calling library routines; in general, you cannot pass **far** pointers, or addresses of **far** data items, to a small-model library routine (some exceptions to this statement are the library routines **malloc**, **hfree**, and the **printf** family).

You can, of course, always pass the *value* of a **far** item to a small-model library routine. For example:

```
long far time_val;
time(&time_val);           /* Illegal */
printf("%ld\n", time_val); /* Legal */
```

If you use the **near**, **far**, or **huge** keywords, it is recommended that you use function declarations with argument-type lists to ensure that pointers are passed to functions correctly (see Section 8.3.1, “Pointer Conversions”).

For more information on library routines and memory models, see Section 2.11, “Using Huge Arrays with Library Functions,” in Chapter 2, “Using C Library Routines,” of the *Microsoft C Compiler Run-Time Library Reference*.

8.3.2 Declaring Data with near, far, and huge

The **near**, **far**, and **huge** keywords modify either objects or pointers to objects. When using them to declare data or code (or pointers to data or code), keep the following rules in mind:

- The keyword always modifies the object or pointer immediately to its right. In complex declarators such as `char far* *p;` think of the **far** keyword and the item to its right as being a single unit. In this case, `p` is a pointer to a **far** pointer to **char** (the size of `p` depends on the memory model being used). See the *Microsoft C Compiler Language Reference* for complete rules for using special keywords in complex declarations.
- If the item immediately to the right of the keyword is an identifier, the keyword determines whether the item will be allocated in the

default data segment (**near**) or a separate data segment (**far** or **huge**). For example,

```
char far a;
```

allocates a as an item of type **char** with a far address.

- If the item immediately to the right of the keyword is a pointer, the keyword determines whether the pointer will hold a near address (16 bits), a far address (32 bits), or a huge address (also 32 bits). For example,

```
char far *p;
```

allocates p as a far pointer (32 bits) to an item of type **char**.

The following examples show data declarations using the **near**, **far**, and **huge** keywords:

Examples

```
char a[3000];           /* Example 1: small-model program */
char far b[30000];      /* Example 2: small-model program */

char a[3000];           /* Example 3: large-model program */
char near b[3000];      /* Example 4: large-model program */

char huge a[70000];     /* Example 5: small-model program */
char huge *pa;          /* Example 6: small-model program */

char *pa;                /* Example 7: small-model program */
char far *pb;            /* Example 8: small-model program */

char far * *pa;          /* Example 9: small-model program */
char far * *pa;          /* Example 10: large-model program */

char far * near *pb;     /* Example 11: any model */
char far * far *pb;      /* Example 12: any model */
```

The declaration in the first example allocates the array a in the default segment; in contrast, the array b in the second example may be allocated in any segment. Since these declarations are made in a small-model program, array a would probably represent frequently used data that was deliberately placed in the default segment for fast access, while array b would probably represent seldom used data that might make the data segment exceed 64K and force the programmer to use a larger memory model if it were not declared with the **far** keyword. The second example uses a large array, because it is more likely that a programmer would want to specify the address allocation size for items of substantial size.

In Example 3, the speed of access would probably not be critical for array *a*; even though it may or may not be allocated to the default data segment, it is always referenced with a 32-bit address. In Example 4, array *b* is explicitly allocated **near** to improve speed of access in this memory model (large).

In Example 5, *a* must be declared as **huge** because it is larger than 64K. Using the **huge** keyword instead of the standard huge memory model means that the price for using huge data is only paid for this one large item. Other data can be accessed quickly within the default segment. The pointer *pa* in Example 6 could be used to point to *a*. Any pointer arithmetic done with *pa* (such as *pa*++), would be done using 32-bit arithmetic.

In Example 7, *pa* is declared as a near pointer to **char**. The pointer is near by default since the example is in a small-model program. In contrast, *pb* in Example 8 is allocated as a far pointer to **char**; *pb* could be used to point to, and step through, an array of characters that has been stored in a segment other than the default data segment. For example, *pa* might be used to point to the array *a* in Example 1, while *pb* might be used to point to the array *b* in Example 2.

The pointer declarations in examples 9 and 10 show the interaction between the memory model chosen and the **near** and **far** keywords: although the declarations for *pa* in these two examples are identical, Example 9 declares *pa* as a near pointer to an array of far pointers to type **char**, while Example 10 declares *pa* as a far pointer to an array of far pointers to type **char**.

In Example 11, *pb* is declared as a near pointer to an array of far pointers to type **char**; in Example 12, *pb* is declared as a far pointer to an array of far pointers to type **char**. Note that, in these final two examples, the inclusion of the **far** and **near** keywords overrides the model-specific addressing conventions shown in examples 9 and 10; the declarations for *pb* would have the same effect, regardless of the memory model.

8.3.3 Declaring Functions with **near** and **far**

The rules for using the **near** and **far** keywords for functions are similar to those for using them with data:

- The keyword always modifies the function or pointer immediately to its right. See Section 4.3.3, “Declarators with Special Keywords,” of the *Microsoft C Compiler Language Reference* for more information about rules for evaluating complex declarations.

- If the item immediately to the right of the keyword is a function, then the keyword determines whether the function will be allocated as near or far. For example, `char far fun();` defines `fun` as a function called with a 32-bit address and returning type **char**.
- If the item immediately to the right of the keyword is a pointer to a function, then the keyword determines whether the function will be called using a near (16-bit) or far (32-bit) address. For example,
`char (far * pfun) ();`
defines `pfun` as a far pointer (32 bits) to a function returning type **char**.
- Function declarations must match function definitions.
- The **huge** keyword cannot be applied to functions.

Examples

```

char far fun( );           /* Example 1: small model */
char far fun( )
{
    .
    .
}

static char far * near fun( ); /* Example 2: large model */
static char far * near fun( )
{
    .
    .
}

void far fun( );           /* Example 3: small model */
void (far * pfun) ( ) = fun;

double far * (far fun)( );  /* Example 4: compact model */
double far * (far *pfun)( ) = fun;

```

In the first example, `fun` is declared as a function returning type **char**. The **far** keyword in the declaration means that `fun` must be called with a 32-bit call.

In the second example, `fun` is declared as a near function that returns a far pointer to type `char`. Such a function might be seen in a large-model program as a helper routine that is used frequently, but only by the routines in its own module. Since all routines in a given module share the same code segment, the function could always be accessed with a near call. However, you could not pass a pointer to `fun` as an argument to another function outside the module in which `fun` was declared.

The third example declares `pfun` as a far pointer to a function that has a `void` return type, and then assigns the address of `fun` to `pfun`. In fact, `pfun` could be used to point to any function accessed with a far call. Note that if the function pointed to by `pfun` has not been declared `far`, or if it is not far by default, then calling that function through `pfun` would cause the program to fail.

The fourth example declares `pfun` as a far pointer to a function that returns a far pointer to type `double`, and then assigns the address of `fun` to `pfun`. This might be used in a compact-model program for a function that is not used frequently and thus does not need to be in the default code segment. Both the function and the pointer to the function must be declared as `far`.

8.3.4 Pointer Conversions

Passing pointers as arguments to functions may cause automatic conversions in the size of the pointer argument, since passing a pointer to a function forces the pointer size to the larger of the following two sizes:

- The default pointer size for that type, as defined by the memory model used during compilation.

For example, in medium-model programs, data pointer arguments are near by default and code pointer arguments are far by default.

- The type of the argument.

If the forward declaration of a function includes declared argument types, the compiler performs type checking and enforces the conversion of actual arguments to the declared type of the corresponding formal argument. However, if no declaration is present or the argument-type list is empty, the compiler will convert pointer arguments automatically to the larger of the default type or the type of the argument. To avoid mismatched arguments, you should always specifically give the argument types in a forward declaration.

Example

```

/* This program produces unexpected results in compact-,
** large-, or huge-model programs.
*/
main( )
{
    int near *x;
    char far *y;
    int z = 1;

    test_fun(x, y, z); /* x will be coerced to far
                           ** pointer in compact, large,
                           ** or huge model
    */
}

int test_fun(ptr1, ptr2, a)
{
    int near *ptr1;
    char far *ptr2;
    int a;

    {
        printf("Value of a = %d\n", a);
    }
}

```

If the preceding example is compiled as a small-model program (no memory model options or **/AS** on **MSC** command line) or medium-model program (**/AM** option), the size of pointer argument *x* is 16 bits, the size of pointer argument *y* is 32 bits, and the value printed for *a* is 1. However, if the preceding example is compiled with the **/AC**, **/AL**, or **/AH** option, both *x* and *y* are automatically converted to far pointers when they are passed to *test_fun*. Since *ptr1*, the first parameter of *test_fun*, is defined as a near pointer argument, it takes only 16 bits of the 32 bits passed to it. The next parameter, *ptr2*, takes the remaining 16 bits passed to *ptr1*, plus 16 bits of the 32 bits passed to it. Finally, the third parameter, *a*, takes the left-over 16 bits from *ptr2*, instead of the value of *z* in the *main* function. This shifting process does not generate an error message, since both the function call and the function definition are legal, but in this case the program does not work as intended, since the value assigned to *a* is not the value intended.

To pass *ptr1* as a near pointer, you should include a forward declaration that specifically declares this argument for *test_fun* as a near pointer, as shown below:

```

/* First, declare test_fun so the compiler knows in advance
** about the near pointer argument:
*/
int test_fun(int near*, char far *, int);

main( )

{
    int near *x;
    char far *y;
    int z = 1;

    test_fun(x, y, z); /* now, x will not be coerced
                         ** to a far pointer; it will be
                         ** passed as a near pointer,
                         ** no matter what memory
                         ** model is used
    */
}

int test_fun(ptr1, ptr2, a)
    int near *ptr1;
    char far *ptr2;
    int a;

{
    printf("Value of a = %d\n", a);
}

```

Note that it would not be sufficient to reverse the definition order for `test_fun` and `main` in the first example to avoid pointer coercions; the pointer arguments must be declared in a forward declaration, as in the second example.

8.4 Creating Customized Memory Models

A third method of managing memory models is to combine features of the standard memory models to create your own customized memory model. You should have a thorough understanding of C memory models and the 8086 architecture before creating your own nonstandard memory models, since there is no library support — other than the C start-up routines — for any of the options that follow.

The **/Astring** option lets you change the attributes of the standard memory models to create your own memory models. 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.

The standard-memory-model options (**/AS**, **/AM**, **/AC**, **/AL**, and **/AH**) can be specified in the **/Astring** form. As an example of how to construct memory models, the standard-memory-model options are listed below with their **/Astring** equivalents:

Standard	Custom Equivalent
/AS	/Asnd
/AM	/Alnd
/AC	/Asfd
/AL	/Alfd
/AH	/Alhd

As an example of the use of customized models, you might want to create a huge-compact model. This model would allow huge data items, but only one code segment. The option for specifying this model would be **/Ashd**.

An even more common use of customized models is to set up segments (see Section 8.4.3 for more information).

Note

For the purposes of the descriptions that follow, the letters **l** for (“long”) and **s** for (“short”) are used for code pointers to distinguish them in the memory-model string from the letters for data pointers. The terms “short” and “long” are equivalent to “near” and “far,” respectively.

8.4.1 Code Pointers

Options

`/Asxx`
`/Alxx`

The letter **s** tells the compiler to generate near (16-bit) pointers and addresses for all code items. This is the default for small- and compact-model programs.

The letter **l** means that far (32-bit) pointers and addresses are used to address all code items. Far pointers are the default for medium-, large-, and huge-model programs.

8.4.2 Data Pointers

Options

`/Anxx`
`/Afxx`
`/Ahxx`

Three sizes are available for data pointers: near, far, and huge. The letter **n** tells the compiler to use near (16-bit) pointers and addresses for all data. This is the default for small- and medium-model programs.

The letter **f** specifies that all data pointers and addresses are far (32-bit). This is the default for compact- and large-model programs.

The letter **h** specifies that all data pointers and addresses are far (32-bit). This is the default for huge-model programs.

When far data pointers are used, no single data item may be larger than a segment (64K) because address arithmetic is performed only on 16 bits (the offset portion) of the address. When huge data pointers are used, individual data items can be larger than a segment (64K) because address arithmetic is performed on the entire 32 bits of the address.

8.4.3 Setting Up Segments

Options

/Adxx
/Auzz
/Awxx

The letter **d** tells the compiler that **SS** equals **DS**; that is, the stack segment and the default data segment are combined into a single segment. This is the default for all programs. In small- and medium-model programs, the stack and all data combined must occupy less than 64K; thus, any data item is accessed with only a 16-bit offset from the segment address in the **SS** and **DS** registers.

In compact-, large-, and huge-model programs, initialized global and static data are placed in the default 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 remember when passing pointers as arguments in large-model programs. Although you may have more than 64K of total data in these models, there can be no more than 64K of data in the default segment. The /**Gt** and /**ND** options can be used to control allocation of items in the default data segment if a program exceeds this limit. (See Section 9.13, "Setting the Data Threshold," and Section 9.14, "Naming Modules and Segments," for more information about these options.)

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 letter **u** is specified, the address in the **DS** register is saved upon entry to each function, and the new **DS** value for the module in which the function was defined is loaded into the register. The previous **DS** value is restored on exit from the function. Therefore, only one data segment is accessible at any given time.

A single segment must be allocated for the stack, and its address stored in the stack register. The stack cannot be placed in a data segment since it must be available throughout the entire program.

The letter **w**, like the letter **u**, sets up a separate stack segment, 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 under 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.

Even though **u** and **w** set up a separate segment for the stack, the stack's size is still fixed at the default size unless this is overridden with the **/Fc** compiler option (**CL** only), or the **/STACK** linker option.

8.4.4 Library Support for Customized Memory Models

Most C programs make function calls to the routines in the C run-time library. Library support is provided for the five standard memory models (small, medium, compact, large, and huge) through four separate run-time libraries (huge and large models both use the large library). 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.

Library support is not guaranteed for programs using a customized memory model, and you will probably need to create a customized library to be used with your customized memory model. You should use the **/NOD** (for no default library search) option when linking, and specify the library files and object files you want to use. Be sure to use the correct start-up 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 start-up file from the medium-model library.

In general, library functions do not support customized memory models, since a particular run-time routine may in turn call another library routine that conflicts with your customized model.

()

()

()

Chapter 9

Advanced Topics

9.1	Introduction	193
9.2	Disabling Special Keywords	193
9.3	Packing Structure Members	193
9.4	Restricting Length of External Names	194
9.5	Labeling the Object File	195
9.6	Suppressing Default-Library Selection	195
9.7	Changing the Default char Type	196
9.8	Controlling Stack and Heap Allocation	197
9.9	Controlling Floating-Point Operations	198
9.9.1	Changing Libraries at Link Time	198
9.9.2	Using the NO87 Environment Variable	200
9.10	Advanced Optimizing	201
9.10.1	Removing Stack Probes	201
9.10.2	Maximum Optimization	203
9.11	Controlling the Function-Calling Sequence	203
9.12	Controlling Binary and Text Modes	205
9.13	Setting the Data Threshold	206
9.14	Naming Modules and Segments	207
9.15	Compiling for Windows Applications	209

1

2

3

9.1 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.

9.2 Disabling Special Keywords

Option

/Za

The Microsoft C Compiler has been enhanced to consider the identifiers in the list that follows as keywords when processing a given file:

```
cdecl  
far  
fortran  
huge  
near  
pascal
```

If you are concerned with porting C programs from other systems in which these are not keywords, use the **/Za** option to tell the compiler to treat these words as ordinary identifiers. When this option is given, the compiler automatically defines the identifier **NO_EXT_KEYS**. In the include files provided with the C Run-Time Library, this identifier is used with **#ifndef** to conditionally compile blocks of text containing the keyword **cdecl**. For an example of this conditional compilation, see the file **stdio.h**.

9.3 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 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.

9.4 Restricting Length of External Names

Option

/H*number*

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.

9.5 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 white-space characters.

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

Example

```
MSC MAIN.C, /V"Microsoft C Compiler Version 4.0";
```

The above command places the string

Microsoft C Compiler Version 4.0

in the object file MAIN.OBJ.

9.6 Suppressing Default-Library Selection

Option

/Zl

Ordinarily the compiler places the names of the default libraries (the standard C library, the helper library **LIBH.LIB**, 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.

9.7 Changing the Default char Type

Option

/J

In Microsoft C, the **char** type is signed by default, so if a **char** value is widened to an **int**, the result will be sign extended. You can change this default to unsigned with the /J option, causing the **char** type to be zero extended when widened to an **int**. However, if a **char** value is explicitly declared **signed**, the /J option does not affect it, and the value is sign extended when widened to an **int**.

9.8 Controlling Stack and Heap Allocation

You can change the model used to allocate heap space by linking your program with one of the ***xVARSTCK.OBJ*** object files (where *x* is the first letter of the library you choose). These files are the small-, medium-, compact-, and large-model versions of a routine that allows the memory allocation functions (**malloc**, **calloc**, **_expand**, **_fmalloc**, **_nmalloc**, and **realloc**) to allocate items in unused stack space if they run out of other memory.

Programs compiled and linked under Microsoft C run with a fixed stack size (the default size is 2048 bytes). The stack resides above static data and the heap uses whatever space is left above the stack. However, for some programs a fixed-stack model may not be ideal; a model where the stack and heap compete for space is more appropriate. Linking with the ***xVARSTCK.OBJ*** object files gives you such a model: when the heap runs out of memory, it tries to use available stack space until it runs into the top of the stack. When the allocated space in the stack is freed, it is once again made available to the stack. Note that the stack cannot grow beyond the last allocated heap item in the stack or, if there are no heap items in the stack, beyond the size it was given at link time. Note also that while the heap can use unused stack space, the reverse is not true: the stack cannot use unused heap space.

When you link your program with one of the ***xVARSTCK.OBJ*** files, you should be wary of suppressing stack checking with the **check_stack** pragma, or the **/Gs** or **/Ox** option; this is because stack overflow can occur more easily in programs that use this option, possibly causing errors that would be difficult to detect. (See Section 9.10.1, “Removing Stack Probes,” and Section 9.10.2, “Maximum Optimization,” for more information on suppression of stack checking.)

Example

```
MSC TEST.C;  
LINK TEST+SVARSTCK;
```

These command lines compile **TEST.C** and then link the resulting object module with **SVARSTCK.OBJ**, the variable-stack object file for small-model programs.

9.9 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 3.8 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.

9.9.1 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 the following:

1. **EM.LIB** (the emulator) plus **xLIBFP.LIB**, where *x* depends on the memory model
2. **87.LIB** (the 8087/80287 library) plus **xLIBFP.LIB**
3. **xLIBFA.LIB**

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 (**xLIBFA.LIB**) 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 at link time you must give the name of the standard C library (**xLIBC.LIB**) and the name of the helper library (**LIBH.LIB**) as well as the names of floating-point libraries. Always give the name of the floating-point library or libraries on the command line *before* the name of the standard C library or the helper library.

Examples

```
MSC /AM CALC;  
LINK CALC+ANOTHER+SUM /NOD,,,87+MLIBFP+MLIBC+LIBH;  
  
MSC /FPa CALC;  
LINK CALC+ANOTHER+SUM /NOD,,,EM+SLIBFP+SLIBC+LIBH;  
  
MSC /FPC87 CALC;  
LINK CALC+ANOTHER+SUM /NOD,,,SLIBFA.LIB+SLIBC.LIB+LIBH.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, **/FPI**, is used. **/FPI** generates 8087/80287 instructions and specifies the emulator (**EM.LIB**) plus **MLIBFP.LIB** in the object file. In the **LINK** command line, the **/NOD** option is specified and the names of the 8087, floating-point, code-helper, and standard C libraries are given in the "Libraries" field. This forces the program to use an 8087 coprocessor; it will fail if none is present. Note that the medium-model libraries (**MLIBFP.LIB** and **MLIBC.LIB**) must be used.

In the second example, **CALC.C** is compiled as small model (by default) and with the alternate math option (**/FPa**). The **LINK** command line specifies the **/NOD** option and gives the names **EM.LIB**, **SLIBFP.LIB**, **SLIBC.LIB**, and **LIBH.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 **/FPC87** option, which places the library names **87.LIB** and **SLIBFP.LIB** in the object file. The **LINK** command line overrides this default-library specification by giving the **/NOD** option and the names of the alternate math library (**SLIBFA.LIB**), the standard library (**SLIBC.LIB**), and the code-helper library (**LIBH.LIB**).

9.9.2 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 2.7 of Chapter 2, "Getting Started," or your MS-DOS documentation for a discussion of environment variables.)

If **NO87** is 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 printed only 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.

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 used only 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 programs linked with any of the alternate math libraries (**xLIBF/A.LIB**).

Examples

```
SET NO87=Use of coprocessor suppressed  
SET NO87=space
```

The first example causes the message

Use of coprocessor suppressed

to appear on the screen when a program that uses an 8087 or 80287 is executed, and an 8087 or 80287 is present.

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

9.10 Advanced Optimizing

This section describes additional optimizing procedures that can be used with the optimizing options described in Section 3.12 of Chapter 3, “Compiling,” to create more efficient programs from your code.

9.10.1 Removing Stack Probes

Options

```
/Gs  
# pragma check_stack[]+]  
# pragma check_stack[]-]
```

You can reduce the size of a program and speed up execution slightly by removing stack probes. You can do this either with the **/Gs** option, or with the **check_stack** pragma.

A stack probe is a short routine called on entry to a function to verify that there is enough room in the program stack to allocate local variables required by the function. The stack probe routine is called at every function entry point. Ordinarily, the stack probe routine generates a stack overflow message when it determines that the required stack space is not available. When stack checking is turned off, the stack probe routine is not called, and stack overflow can occur without being diagnosed (i.e., no message is printed).

In general, use the **/Gs** option when you want to turn off stack checking for an entire module. This 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, or that have only modest local variable requirements. In the absence of the **/Gs** option, stack checking is on.

Use the **check_stack** pragma when you want to turn stack checking on or off only for selected routines, leaving the default (as determined by the presence or absence of the **/Gs** option) for the rest. When you want to turn off stack checking, put the following line before the definition of the function you don’t want to check:

```
#pragma check_stack-
```

Note that the preceding line disables stack checking for all routines that follow it, not just the routines on the same line. To reinstate stack checking, insert the following line:

```
#pragma check_stack+
```

If the trailing + or - is left off the **check_stack** pragma, stack checking is disabled if the /Gs option is present, and enabled if it is not. The interaction of the **check_stack** pragma with the /Gs option is explained in greater detail in Table 9.1.

Table 9.1
Using the `check_stack` Pragma

Syntax	Compiled with /Gs Option?	Action
<code># pragma check_stack</code>	yes	Turns off stack checking for routines that follow
<code># pragma check_stack</code>	no	Turns on stack checking for routines that follow
<code># pragma check_stack+</code>	yes	Turns on stack checking for routines that follow
<code># pragma check_stack+</code>	no	Turns on stack checking for routines that follow
<code># pragma check_stack-</code>	yes	Turns off stack checking for routines that follow
<code># pragma check_stack-</code>	no	Turns off stack checking for routines that follow

Although the /Gs option, combined with the /Osa option, described with the /Ostring options in Section 3.12, “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. If you wanted stack checking for only a few functions in `FILE.C`, you could use the `check_stack` pragma surrounding the definitions of functions you wanted to check.

9.10.2 Maximum Optimization

Option

/Ox

The `/Ox` option is a shorthand way to combine optimizing options to produce the fastest possible program. Its effect is the same as using the following options on the same command line:

```
/Oat /Gs
```

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

9.11 Controlling the Function-Calling Sequence

Options

```
/Gc  
fortran  
pascal  
cdecl
```

The `fortran`, `pascal` and `cdecl` keywords, and the `/Gc` option, allow you to control the function call/return sequence and naming convention, so your C programs can call and be called by functions written in FORTRAN and Pascal.

Because C, unlike other languages such as Microsoft Pascal and Microsoft FORTRAN, allows the user to write functions that take a variable number of arguments, it must handle function calls differently. Languages such as Pascal and FORTRAN normally push actual parameters to a function in left-to-right order, with the last argument in the list being the last one

pushed. In contrast, C functions do not know the number of actual parameters, so they must push their arguments from right to left, with the first argument in the list being the last one pushed. Additionally, the calling function must remove the arguments from the stack in C (rather than having the called function do it, as in Pascal and FORTRAN). If the code for removing arguments is in the function definition (as in Pascal and FORTRAN), it appears only once; if it is in the calling function (as in C), it appears every time there is a function call. Since function calls are more numerous than function definitions, the Pascal/FORTRAN method is often slightly smaller and more efficient.

The Microsoft C Compiler has the ability to generate the Pascal/FORTRAN call/return sequence in one of several ways. The first is through the use of the **pascal** and **fortran** keywords. These keywords, when applied to functions or pointers to functions, indicate a corresponding Pascal or FORTRAN function; therefore, the correct call/return sequence must be used. In the following example, `sort` is declared as a function using the alternative call/return sequence:

```
short pascal sort(char *, char *);
```

The **pascal** and **fortran** keywords can be used interchangeably. Use them when you want to use the left-to-right calling sequence for selected functions only.

The second method for generating the Pascal/FORTRAN call/return sequence is to use the **/Gc** option. If you use the **/Gc** option, the entire module will be compiled using the alternative call/return sequence. You might use this method to make it possible to call all the functions in a C module from another language, or to gain the performance and size improvement provided by this call/return sequence. However, if you use the **/Gc** option, you cannot call or define functions that take variable numbers of parameters, nor can you call functions such as the C library functions that use the C calling sequence. Moreover, when you use **/Gc** to compile a module, the compiler assumes that all functions called from that module use the Pascal/FORTRAN call/return sequence, even if the functions are defined outside that module.

To overcome these restrictions, the **cdecl** keyword has been added to Microsoft C. When applied to a function or function pointer, it indicates that the associated function is to be called using the normal C call/return sequence. This allows you to write C programs which take advantage of the more efficient call/return sequence while still having access to the entire C library, other C objects, and even user-defined functions that can take variable-length argument lists.

For convenience, the **cdecl** keyword has already been applied to run-time library function declarations in the include files distributed with this compiler.

Use of the **pascal** and **fortran** keywords, or the **/Gc** option, also affects the naming convention for the associated item (or, in the case of **/Gc**, all items): the name is converted to uppercase (capital letters), and the leading underscore that C normally prefixes is not added. The **pascal** and **fortran** keywords can be applied to data items and pointers, as well as functions; when applied to data items or pointers, these keywords force the naming convention described above for that item or pointer.

The **pascal**, **fortran**, and **cdecl** keywords, like the **near**, **far**, and **huge** keywords, are disabled by use of the **/Za** option. If this option is given, these names will be treated as ordinary identifiers, rather than keywords.

Example

```
int cdecl var_print(char*,...);
```

In the preceding example, `var_print` is allowed to have a variable number of arguments by declaring it as a function using the normal right-to-left C function call/return sequence and naming conventions; the `cdecl` keyword overrides the left-to-right calling sequence set by use of the **/Gc** option when compiling the source file in which this declaration appears; if this file is compiled without the **/Gc** option, `cdecl` will have no effect since it is the same as the default C convention.

For more information on mixed-language programming, see Chapter 10, “Interfaces with Other Languages.”

9.12 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–line-feed combinations (CR–LF) are translated into a single line-feed (LF) character on input. Line-feed characters are translated to carriage-return–line-feed combinations on output.

In some cases you may want to process files without making these translations. In binary mode, carriage-return-line-feed translations are suppressed.

Standard library routines such as **fopen** or **open** give you the option of overriding 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 path name of **BINMODE.OBJ** to the list of object file names when you link your program.

When you link with **BINMODE.OBJ**, all files opened in your program default to binary mode, with the exceptions 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 modes of **stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn**, which are not explicitly opened by users.

9.13 Setting the Data Threshold

Option

/Gt[[*number*]]

By default, the compiler allocates all static and global data items to the default data segment in the small and medium memory models. In compact-, large-, and huge-model programs, only *initialized* static and global data items are assigned to the default data segment. The **/Gt** option causes all data items whose size is greater than or equal to *number* bytes to be allocated to a new data segment. When *number* is specified, it must follow the **/Gt** option immediately, with no intervening spaces. When *number* is omitted, the default threshold value is 256.

You can only use the **/Gt** option with compact-, large-, and huge-model programs, since small- and medium-model programs have only one data segment. The option is particularly useful with programs that have more than 64K of initialized static and global data in small data items.

9.14 Naming Modules and Segments

Options

```
/NM modulename  
/NT textsegmentname  
/ND datasegmentname
```

Note the space between each preceding option and the following *name*; this space is required. “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-file name. You can change this name using the /NM (for “name module”) option. The new *modulename* 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 10.2, “Assembly-Language Interface.”)

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 base name along with the suffix **_TEXT**. The data segment is named **_DATA**, as in the small model.

In compact-model programs, the data from each module is placed in a separate segment with a distinct name, formed by using the module base name along with the suffix **_DATA**. The exception to this is initialized global and static data, which are put in the default data segment, **_DATA**. The code segment is named **_TEXT**, as in the small model.

In large- and huge-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 initialized global and static data placed in the default data segment). The naming conventions for text and data segments are summarized in Table 9.2.

Table 9.2
Segment-Naming Conventions

Model	Text	Data	Module
Small	_TEXT	_DATA	<i>filename</i>
Medium	<i>module</i> _TEXT	_DATA	<i>filename</i>
Compact	_TEXT	_DATA ¹	<i>filename</i>
Large	<i>module</i> _TEXT	_DATA ¹	<i>filename</i>
Huge	<i>module</i> _TEXT	_DATA ¹	<i>filename</i>

¹ Name of default data segment; other data segments have unique private names

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 to a given name the names of the text and data segments in each module being compiled. The *textsegmentname* used with the **/NT** option and *databsegmentname* used with the **/ND** option can be any combination of letters and digits.

If you use the **/ND** option to change the name of the default data segment, your program can no longer assume that the address contained in the stack segment register (**SS**) is the same as the address in the data segment register (**DS**). You must therefore compile your program with the long form of the memory-model option and the **u** flag, as in the following example:

```
MSC PROG1 /Ausn /ND DATA1;
```

Use of the **/A_{xx}** options forces the compiler to generate code to load **DS** with the correct data-segment value on entry to the code. See Section 8.4, “Creating Customized Memory Models,” for more information on the **/A_{string}** options.

9.15 Compiling for Windows Applications

Options

`/Awxx`
`/Gw`

The `/Awxx` option controls the segment setup, and should be used for C programs that interface with the Microsoft Windows operating system. For more information on this option, see Section 8.4.3, “Setting Up Segments.”

You should use the `/Gw` option for developing applications to run on the Windows environment. See your *Microsoft Windows Software Development Kit* for details on how and when to use this option.



Chapter 10

Interfaces with Other Languages

10.1	Introduction	213
10.2	Assembly-Language Interface	213
10.2.1	Segment Model	213
10.2.1.1	Segments	214
10.2.1.2	Groups	216
10.2.1.3	Classes	217
10.2.2	The C Calling Sequence	219
10.2.3	The Pascal/FORTRAN Calling Sequence	221
10.2.4	Entering an Assembly Routine	221
10.2.5	Return Values	222
10.2.6	Exiting a Routine	224
10.2.7	Naming Conventions	224
10.2.8	Register Considerations	225
10.2.9	Program Example	226
10.3	Mixed-Language Programming	229
10.3.1	Memory Models	230
10.3.2	Choosing a Calling Convention	230
10.3.2.1	Passing Parameters by Reference or Value	231
10.3.2.2	Using Varying Numbers of Parameters	236
10.3.3	Naming Conventions	236
10.3.4	Writing Interfaces to Pascal or C from FORTRAN	238
10.3.5	Calling Procedures in Pascal or C from FORTRAN	240

10.3.6	Writing Interfaces to FORTRAN or C from Pascal	241
10.3.7	Calling Procedures in FORTRAN or C from Pascal	241
10.3.8	Writing Interfaces to FORTRAN or Pascal from C	242
10.3.9	Calling Procedures in FORTRAN or Pascal from C	243
10.3.10	Data Types	243
10.3.10.1	Using the Equivalent Data Types Tables	243
10.3.10.2	Integers	244
10.3.10.3	Boolean and Character Types	248
10.3.10.4	Real Numbers	248
10.3.10.5	Passing Strings	250
10.3.10.6	Pointers	254
10.3.10.7	Arrays, Super Arrays and Huge Arrays	256
10.3.10.8	Records and Structs	259
10.3.10.9	Procedural Parameters	262
10.3.11	Return Values	262
10.3.12	Sharing Data	262
10.3.13	Input and Output	264
10.3.14	Compiling and Linking	264
10.3.15	Error Messages	265

10.1 Introduction

The Microsoft C Compiler can be used to prepare modules for use by other languages, and modules prepared with other languages can be used with C programs.

This chapter first tells how to mix assembly-language modules with C modules. This is a powerful technique for preparing assembly-language libraries for C, or for using C routines (including those from the standard library) in assembly-language programs.

The chapter also discusses mixing modules created with Microsoft C, Microsoft FORTRAN, and Microsoft Pascal.

10.2 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 Microsoft C Compiler versions 2.03 or earlier, turn to Section F.3.3 of Appendix F, “Converting from Previous Versions of the Compiler,” for a discussion of differences between the assembly-language interface for this compiler and earlier versions.

10.2.1 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. 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.

10.2.1.1 Segments

Figure 10.1 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 order shown here. Since this is the default order for C programs, you do not need to use **/DOSSEG** with C programs, but you may find it useful when linking assembly-language routines.

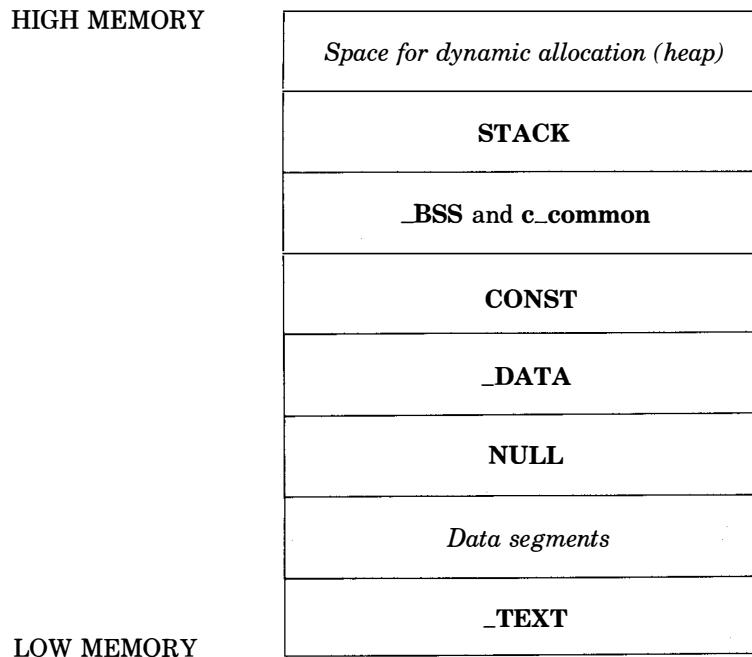


Figure 10.1 Segment Setup in C Programs

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:

Segment	Contents
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 those that are explicitly declared as far or huge items in the source file.
c_common	The c_common segment contains all uninitialized global data items for small- and medium-model programs. In compact- or 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 or huge in the source file, or data items that are forced into their own segment by use of the /Gt option.
_DATA	Writing to string literals is allowed in C. Therefore, strings are stored in the _DATA segment rather than the CONST segment.
NULL	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 huge , 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 at program termination is displayed to notify the user. Although a program may appear to run correctly when this happens, it may not run under other environments.

Data segments Initialized static and global **far/huge** 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 precede **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**. Uninitialized **huge** items are placed in segments with class **HUGE_BSS**. In large- and huge-model programs, *global* uninitialized data are treated as though declared **far** (unless specifically declared **near**) and given class **FAR_BSS**.

_TEXT The **_TEXT** segment is the code segment. In small- and compact-model programs, the code for all modules is combined in this segment. In medium-, large-, and huge-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 *modulename_TEXT* for medium-, large-, and huge-model programs). Data should be placed in the segment appropriate for their use, as described above. Usually this is the default data segment, **_DATA**.

10.2.1.2 Groups

All segments with the same group name must fit into a single physical segment, which is up to 64K 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 **NUL**, **_DATA**, **CONST**, **BSS**, **c_common**, and **STACK** segments are grouped together in this data group. 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** contain the same value unless the **u** or **w** flag of the **/A** option is used.

In compact-, large-, and huge-model programs, or small- and medium-model programs using **far** data declarations, **DS** may be temporarily changed 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. Its purpose is to provide 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.

10.2.1.3 Classes

Table 10.1 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 10.1

Segments, Groups, and Classes for Standard Memory Models

Memory Model	Segment Name	Align Type	Combine Class	Class Name	Group
Small	_TEXT	byte	public	CODE	
	<i>Data segments</i> ¹	para	private	FAR_DATA	
	<i>Data segments</i> ²	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	<i>module_</i> _TEXT	byte	public	CODE	
	.				
	.				
	.				
	<i>Data segments</i> ¹	para	private	FAR_DATA	
	<i>Data segments</i> ²	para	public	FAR_BSS	

Table 10.1 (continued)

Memory Model	Segment Name	Align Type	Combine Class	Class Name	Group
	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
Compact	_TEXT	byte	public	CODE	
	<i>Data segments</i> ³	para	private	FAR_DATA	
	<i>Data segments</i> ⁴	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	<i>module_- TEXT</i>	byte	public	CODE	
	.				
	.				
	.				
	<i>Data segments</i> ³	para	private	FAR_DATA	
	<i>Data segments</i> ⁴	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
Huge	<i>module_- TEXT</i>	byte	public	CODE	
	.				
	.				
	.				
	<i>Data segments</i> ³	para	private	FAR_DATA	
	<i>Data segments</i> ⁴	para	public	FAR_BSS	

Table 10.1 (continued)

Memory Model	Segment Name	Align Type	Combine Class	Class Name	Group
	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

¹ Segment(s) for initialized **far** or **huge** data² Segment(s) for uninitialized **far** or **huge** data³ Segment(s) for initialized global and static data⁴ Segment(s) for uninitialized global and static data

10.2.2 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**, **signed char**, **signed short**, **signed 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 unless the /J option is given when compiling, **char** type arguments are sign extended to **int** type before being pushed onto the stack; if the /J option is given, **char** arguments are zero extended to **unsigned int**. In either case, **unsigned char** type arguments are always zero extended, and **signed char** type arguments are always sign extended.

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. In the memory models (compact, large, and huge) where the default pointer size is **far**, function arguments that are data pointers are automatically coerced to far pointers, unless there is a function declaration preceding the function call that declares the arguments as **near**.

pointers. For example, if the following program is compiled in compact, large, or huge model, it will not print "1" as expected; this is because the value of a is not where test_fun expects it to be. Instead, the value contained in a consists of the extra bytes pushed on the stack for the pointer argument ptr.

```
main( )
{
    int near *x;
    int y = 1;

    /* x will be coerced to far pointer, in spite
     ** of its declaration as near: */
    test_fun(x, y);
}

int test_fun(ptr, a)
{
    int near *ptr;
    int a;

    {
        /* The value printed for a will not be 1: */
        printf("Value of a = %d\n", a);
    }
}
```

The correct way to use this function is as follows:

```
/* First, declare test_fun so the compiler knows in advance
** about the near pointer argument:
*/
int test_fun(int near*, int);

main( )

{
    int near *x;
    int y = 1;

    /* Now, x will be passed as the near pointer: */
    test_fun(x, y);
}
```

```

int test_fun(ptr, a)
    int near *ptr;
    int a;

{
/* Value of a = 1 */
printf("Value of a = %d\n", a);
}

```

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.

10.2.3 The Pascal/FORTRAN Calling Sequence

The Pascal/FORTRAN calling convention, enabled for an entire module by use of the **/Gc** option, or for individual functions within a module with the **fortran** or **pascal** keywords, causes functions to use calls in which function arguments are pushed onto the stack left to right (i.e., the last argument is the last argument pushed). When this alternative calling sequence is enabled, the called function is responsible for removing the arguments from the stack. Also, use of **/Gc** means that it is not possible to have functions with variable-length argument lists, unless they are explicitly declared with the **cdecl** keyword. (See Section 9.11, “Controlling the Function-Calling Sequence,” for more information about the **/Gc** option and the **fortran** and **pascal** keywords.)

10.2.4 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** flag 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    ;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
.
.
```

This is the same sequence used by the C compiler; in fact, you can generate an assembly-language listing such as that above by compiling your C program with the **/Fa** or **/Fc** option (see Section 3.5, "Producing Listing Files," for more information). 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, and [bp+6] for a far function call. Subsequent arguments begin at [bp+6], [bp+8], or [bp+10], depending on 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].

Note 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 a good idea to write macros 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.

10.2.5 Return Values

In order for assembly-language routines to return values to a C-language program or receive return values from C functions, they must follow the C return value conventions. The conventions are shown in Table 10.2.

Table 10.2
C Return Value Conventions

Return Value Type	Register
char	AX
short	AX
int	AX
signed char	AX
signed short	AX
signed 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 constant, or static or global value
float or double	address of value in AX; value must be constant, or static or global value
near pointer	AX
far pointer	segment selector in DX; offset in AX

10.2.6 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:

```
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
```

This sequence does not change the **AX**, **BX**, **CX**, or **DX** register or any of the segment registers. The sequence does not remove arguments from the stack; this is the responsibility of the calling routine.

Note

If the module from which the assembly-language routine is called has been compiled with the **/Gc** option, or if the external declaration of the assembly-language routine contains the **pascal** or **fortran** keyword, then the assembler routine must remove its own arguments from the stack before returning to the calling routine. In this case, the **ret** instruction at the end of the preceding example should be replaced with the **ret num** instruction (return and pop *num* bytes off the stack, where *num* is the size in bytes of all arguments).

Note 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.

10.2.7 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 with the following declaration:

```
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, versions 3.0 and later, offers an option to control case sensitivity.

10.2.8 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** flag of the **/A** option is specified to set up separate stack and data segments.

10.2.9 Program Example

To illustrate the assembly-language interface, the two functions `mul` and `main` from the small-model C program in Example 1 are written as assembly-language routines in Examples 2 and 3.

Example 1

```
int a = 2, b = 7, c = 0;
main( )
{
    c = mul(a,b);
}

int mul(i,j) /* Performs multiplication by repeated
               ** additions
               */
int i,j;
{
    register int k, sum;

    sum = 0;
    for (k = 1; k <= j; k++)
        sum += i;
    return(sum);
}
```

Example 2

This example replaces the `mul` function in Example 1 with an assembly-language routine called by `main` (you can obtain a similar assembly listing by compiling the `mul` function with the `/Fa` or `/Fc` option on the **MSC** command line).

```
TITLE    mulbyadd

_TEXT    SEGMENT  BYTE PUBLIC 'CODE'
_TEXT    ENDS
_DATA   SEGMENT  WORD PUBLIC 'DATA'
_DATA   ENDS
CONST   SEGMENT  WORD PUBLIC 'CONST'
CONST   ENDS
_BSS    SEGMENT  WORD PUBLIC 'BSS'
_BSS    ENDS

DGROUP  GROUP    CONST,    _BSS,      _DATA
ASSUME  CS: _TEXT,  DS: DGROUP,  SS: DGROUP, ES: DGROUP
```

```

_TEXT      SEGMENT
        PUBLIC _mul
_mul     PROC NEAR
;
;      i = 4
;      j = 6
;
        push    bp
        mov     bp,sp
        sub     sp,4
;
;      register di = sum
;      register si = k
        push    di          ; save si,di
        push    si
;
        sub     di,di        ; sum = 0;
        mov     si,1         ; k = 1;
        jmp     SHORT $test
$loop:
        add     di,[bp+4]    ; sum +=i;
        inc     si           ; i++;
;
$test:
        cmp     [bp+6],si    ; j >= k ?
        jge     $loop         ; yes, loop again
        mov     ax,di         ; no, return sum
;
        pop     si           ; restore si,di
        pop     di
        mov     sp,bp
        pop     bp
        ret
;
_mul     ENDP
_TEXT    ENDS
END

```

Note that this routine must save the proper registers, retrieve the arguments from the stack, do its calculations, place the return value in the **AX** register, restore the registers, and return control to the calling function. Also, if the assembly routine were written to work with a medium- or large-model C program, the `_mul` procedure would be declared **FAR** instead of **NEAR**.

Example 3

This example replaces the main function in Example 1 with an assembly-language routine that calls mul (you can obtain a similar assembly listing by compiling the main function with the /Fa or /Fc option).

```
DGROUP GROUP _DATA
ASSUME CS:_TEXT, DS:DGROUP, SS:DGROUP

EXTRN _mul:NEAR

_DATA SEGMENT WORD PUBLIC 'DATA'
PUBLIC _a
_a DW 02H
PUBLIC _b
_b DW 07H
PUBLIC _c
_c DW OOH
_DATA ENDS

_TEXT SEGMENT BYTE PUBLIC 'CODE'

PUBLIC _main
_main PROC NEAR

    push bp
    mov bp, sp

    push _b
    push _a
    call _mul           ; mul(a,b);
    add sp, 4            ; pop arguments off stack
    mov _c, ax           ; store result in c

    mov sp, bp
    pop bp
    ret

_main ENDP
_TEXT ENDS
END
```

Note that this 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.

10.3 Mixed-Language Programming

Microsoft FORTRAN and Pascal (versions 3.3 and later) and Microsoft C (versions 3.0 and later) provide support for programmers who use more than one of these languages.

Note

Microsoft C for XENIX does not include the **fortran** and **pascal** keywords described in this document. Therefore, if you want your MS-DOS C programs to be compatible with XENIX, you cannot call FORTRAN and Pascal from the XENIX version of C unless you use the C calling conventions.

FORTRAN and C programmers, please note: throughout this section, the term "procedure" is used instead of "subroutine" or "function," and the term "parameter" is used instead of "argument." This is the terminology used in Pascal.

Mixed-language programming offers several advantages:

1. You can use libraries of procedures written in different languages.

For example, you can access the Microsoft C library from programs written in FORTRAN or Pascal. There are also many proprietary libraries available for use with Microsoft FORTRAN which you can access from Microsoft Pascal and C.

To use a library written for a particular language, you must have the library supplied with that language's compiler. To use a proprietary FORTRAN library from C, for example, you need the library supplied with the FORTRAN compiler, as well as the proprietary library itself. This is because programs written in Microsoft Pascal, C, or FORTRAN contain calls to their respective run-time libraries.

2. You can use features not available in your language.

It is hard to write bit-manipulation procedures in FORTRAN, for example, but it is easy in C or Pascal. Also, some interfaces, such as those that use C or Pascal structures, are not compatible with FORTRAN.

3. If you write your own libraries, you can now produce one library that is compatible with all three languages.

Of course, to ensure compatibility, you must pay close attention to the guidelines given in this section.

10.3.1 Memory Models

The current versions of Pascal and FORTRAN do not offer a choice of memory models; they are compatible only with large-model C. Some components of the C library are referenced from the other languages' libraries. If you use the library for the wrong memory model, these interfaces will be incorrect. Therefore, if you use C procedures that call, or are called by, Pascal or FORTRAN routines, you must compile your C code with the large model. Use the /AL option to specify large model.

10.3.2 Choosing a Calling Convention

FORTRAN, Pascal, and C each have conventions for passing parameters.

The languages differ in the order in which parameters are pushed onto the stack. Microsoft Pascal and Microsoft FORTRAN push parameters onto the stack in the order in which they appear in the procedure declaration. C pushes its parameters in the reverse order.

The languages also differ in whether code telling how to restore the stack when a procedure returns is in the calling procedure or in the called procedure. In the FORTRAN/Pascal convention, this code is in the called procedure; in the C language, this code follows the procedure call.

The FORTRAN/Pascal convention is slightly faster and produces less code. The C convention allows you to use a variable number of parameters (because the first parameter is always the last one pushed, it is always on top of the stack, and always has the same address relative to the start of the frame). These conventions are incompatible.

Finally, the languages differ in which parameters they pass by reference and by value. Section 10.3.2.1, "Passing Parameters by Reference or Value," discusses these differences.

If you control both the calling and the called code, you can choose which calling convention to use. If you intend to pass variable numbers of parameters, you must use the C calling convention. For more information, see Section 10.3.2.2, "Using Varying Numbers of Parameters." Otherwise, you

may want to use the convention of the language that you use most often, so that you can usually use the default calling convention.

To make calls from one language to another, you must tell the compiler which convention to use. Microsoft C, Pascal, and FORTRAN all provide ways of specifying which convention you are using, both when you call an external procedure and when you define a public procedure. Table 10.3 indicates how to specify calling conventions from each language.

Table 10.3
Specifying Calling Conventions

Calling Convention	Language Calling From	Attributes/Keywords to Use
C	Pascal	C attribute on procedure declaration
	FORTRAN	C attribute on INTERFACE statement
	C	Default or cdecl keyword with /Gc option
FORTRAN	Pascal	FORTRAN attribute on procedure declaration
	FORTRAN	Default
	C	fortran keyword on procedure declaration, or /Gc option
Pascal	Pascal	Default
	FORTRAN	PASCAL attribute on INTERFACE statement
	C	pascal keyword on procedure declaration, or /Gc option

10.3.2.1 Passing Parameters by Reference or Value

When a parameter is passed by reference, the address of the parameter is passed. Procedures access the parameter's value through the address; any changes to the parameter affect the stored value. When a parameter is passed by value, a copy of the parameter is placed on the stack when the procedure is called. The procedure can change the value of the parameter without affecting the original value from which the copy was taken.

For each parameter, you must decide whether to pass by value or by reference. If you pass by reference, you also have to choose whether to pass a long address (segment and offset) or a short address (offset only).

If the called procedure needs to change the actual value in the variable as a way of returning a result, you have to pass by reference. Passing by value protects against accidental updating and, for variables smaller than about 4 bytes, can be more efficient.

The following list describes the defaults for each language:

- FORTRAN passes all parameters by reference (including constants and expressions), but passing by value can be specified. If a procedure is given the C or Pascal attribute, the default is changed: all parameters for that procedure are passed by value unless otherwise specified.
- C always passes arrays by reference, and passes all other parameters by value. In C, you can pass pointers as parameters; the procedure can use the pointers to modify stored values, producing the same effect as passing by reference.
- Pascal passes by value, but passing by reference can be specified.

If you do not choose the default case, you have to specify certain keywords, attributes, or pointer types. These will vary, according to the calling conventions you are using. See tables 10.4 through 10.6.

If you are passing parameters when using C calling conventions, use the constructs described in Table 10.4 when declaring parameters.

Table 10.4
Passing Parameters With C Calling Conventions

Parameter	C	Pascal	FORTRAN
Long address	Pointer to <i>type</i>	VARS keyword	REFERENCE attribute
Short address	near pointer to <i>type</i>	VAR keyword	REFERENCE, NEAR attributes
Value	Default	Default	Default

For example, assume that you are using the C calling conventions. Table 10.3 shows what attributes and keywords are necessary to use the C calling conventions. When calling from Pascal, specify the **C** attribute on the procedure declaration. When calling from FORTRAN, specify the **C** attribute on the **INTERFACE** statement. When calling from C, the C calling conventions are the default, unless your program has been compiled with the **/Gc** option, or the function your program is calling has been declared with the **fortran** or **pascal** keyword (see Section 9.11, “Controlling the Function Calling Sequence”).

Assume that you want to pass an integer parameter, *x*, using a long address. Compatibility of data types is discussed in Section 10.3.10; for now, assume that the C **int** type, the Pascal **integer** type, and the FORTRAN **INTEGER** type are equivalent. Table 10.4 shows that when declaring the parameter *x* in your C procedure, you should use a pointer (a **far** pointer, the default) of the appropriate type (in this case, **int**). The following is the C declaration:

```
int *x;
```

When declaring the parameter *x* in your Pascal procedure, use the **VARS** keyword:

```
VARS x:INTEGER;
```

For the FORTRAN procedure, specify this reference attribute:

```
INTEGER X[REFERENCE]
```

If you want to pass using a short address instead, use these declarations:

```
int near *x;
```

```
VAR x:INTEGER;
```

```
INTEGER X[REFERENCE,NEAR]
```

You follow the same steps when declaring parameters even if you are using other calling conventions. If you are passing parameters using Pascal or FORTRAN calling conventions, use the constructs described in Tables 10.5 and 10.6 when declaring parameters.

Table 10.5
Passing Parameters With Pascal Calling Conventions

Parameter	C	Pascal	FORTRAN
Long address	Pointer to <i>type</i>	VARS keyword	REFERENCE attribute
Short address	near pointer to <i>type</i>	VAR keyword	REFERENCE , NEAR attributes
Value	Default	Default	Default

Table 10.6
Passing Parameters With FORTRAN Calling Conventions

Parameter	C	Pascal	FORTRAN
Long address	Pointer to <i>type</i>	VARS keyword	Default
Short address	near pointer to <i>type</i>	VAR keyword	NEAR attribute
Value	Default	Default	VALUE attribute

If you are not writing both the called procedure *and* the calling procedure, you must pass the parameter as declared in the existing procedure's definition. If you are not experienced with the language you are accessing, it is not always easy to determine if a parameter is being passed by value or by reference. The following lists indicate how to tell the difference.

The following kinds of parameters are passed by value:

- In Pascal, any parameter declared except **VAR**, **CONST**, **VARS**, and **CONSTS** parameters
- In C, any parameter declared except arrays
- In FORTRAN, a parameter declared with the **VALUE** attribute

- In FORTRAN, a parameter in a procedure when that procedure is declared with the **C** or **PASCAL** attribute (unless the **REFERENCE** attribute is specified)

The following kinds of parameters are passed by reference with a short (2-byte, offset only) address:

- In Pascal, a formal parameter declared as **VAR** or **CONST**.
- In Pascal, a variable passed by passing a pointer to that variable. The pointer itself is passed by value. (It is not recommended that you use pointers in this way; the correspondence between pointers and machine addresses is implementation dependent.)
- In Pascal, a variable passed by passing **ADR** *variable*. The address itself (as with pointers) is passed by value.
- In C, a parameter passed by passing a **near** pointer to the parameter. (The pointer is passed by value.)
- In C, an array declared with the keyword **near**.
- In FORTRAN, in procedures without the **C** or **PASCAL** attribute, a parameter with the **NEAR** attribute.
- In FORTRAN, in procedures with the **C** or **PASCAL** attribute, a parameter with the **NEAR** and **REFERENCE** attributes.
- In FORTRAN, a variable passed by short address by taking **LOCNEAR**(*variable*), then passing the result as an **INTEGER*2**, by value.

The following kinds of parameters are passed by reference with a long (4-byte, segmented) address:

- In Pascal, **ADS** *variable*. (The address is passed by value.)
- In Pascal, parameters declared with the **VARS** or **CONSTS** keywords.
- In C, a parameter passed by passing a **far** pointer to the parameter. (The pointer is passed by value.) Note that in large-model C programs, **far** pointers are the default pointer type.
- In C, arrays not declared with the keyword **near**.
- In FORTRAN, any parameter of a FORTRAN-protocol routine, except those declared with the **NEAR** or **VALUE** attribute.

- In FORTRAN, a variable passed by long address by taking **LOC(variable)** or **LOCFAR(variable)**, then passing the result as an **INTEGER*4**, by value.

10.3.2.2 Using Varying Numbers of Parameters

If you are going to use varying numbers of parameters, remember the following factors:

- The number of actual parameters must be less than or equal to the number of formal parameters (if the called procedure is written in FORTRAN or Pascal).

In Pascal and FORTRAN there is no easy way to access parameters that have not been formally defined. However, you can use the **VARYING** attribute to pass fewer arguments than are defined.

- You must use the **C** and **VARYING** attributes on your FORTRAN INTERFACE statement or Pascal procedure declaration.

The **VARYING** attribute tells the FORTRAN or Pascal compiler not to check if there are more or fewer actual parameters than formal parameters. However, actual parameters for which a formal parameter is specified *will* be checked for type compatibility, according to the usual rules of the calling procedure's language.

10.3.3 Naming Conventions

If you follow these two rules, the Microsoft Pascal, FORTRAN, and C compilers handle all the necessary adjustments in names:

- If you are using any FORTRAN routines, all identifiers (names) should be six characters or less in length.
- Avoid using uppercase characters in C identifiers. If you must use uppercase characters, do *not* use the /NOIGNORECASE option, and do not use other identifiers that have the same spelling as the uppercase or mixed-case C identifier. (For example, if one C identifier is `AnExample`, don't use `anexample`, `ANEXAMPLE`, or `AnExAmP1E` as an identifier.)

If you cannot follow those two rules, you must make certain adjustments yourself. The remainder of this section explains the default naming conventions of each language, and how certain attributes and keywords affect

those naming conventions. This information should allow you to solve any special problems in naming.

In all three languages, names appear differently in the object and source files. There are differences in the following three elements of the naming conventions used by the three languages:

Element	Differences
Case	In FORTRAN and Pascal, any lowercase letters in a public identifier are changed to uppercase before the name is inserted in the object file. By default, no such transformation is done on C names, but at link time you can specify that case distinctions are to be ignored.
Length	In FORTRAN, by default, names are truncated to six significant characters.
Underscores	In C, public names are always prefixed with an underscore character (_) before they are inserted in the object file.

These differences in naming conventions mean that default FORTRAN and Pascal public names will not correspond to default C public names. Certain attributes and keywords can help make names correspond.

If you want FORTRAN or Pascal identifiers to follow the C conventions, specify the **C** attribute on the following specifiers:

- Names of public or external procedures, or data objects in Pascal
- Names of procedures, interfaces, or named common blocks in FORTRAN

The name is changed to lowercase and a leading underscore is added. FORTRAN identifiers will still be truncated to six characters. To specify a longer name, or to specify external C routines that have uppercase letters in their identifiers, you can use the **ALIAS** in FORTRAN. There is no **ALIAS** feature in Pascal; to refer to a C object with uppercase letters in its identifier, you *must not* link with the **/NOIGNORECASE** option, and all your C identifiers must have unique spellings.

If you use the **pascal** or **fortran** keyword in C, the name is changed to uppercase and the leading underscore is not added to the name. All such names must have unique spellings.

Note that in FORTRAN, if an **INTERFACE** and the subprogram referred to in that **INTERFACE** are in the same unit of compilation, the same names must be used for parameters in each. An error message is generated if you violate this rule.

10.3.4 Writing Interfaces to Pascal or C from FORTRAN

To declare external procedures in C or Pascal from FORTRAN, FORTRAN provides the **INTERFACE** statement.

Suppose, for example, that you want to access the procedure **time** in the C library. There are three basic steps:

1. Find the declaration of the C procedure.
2. Build an **INTERFACE** program unit to determine the following:
 - The attributes and type for the procedure
 - The attributes and types for the parameters
3. Add the **INTERFACE** to the program.

The final step, calling the C procedure, is described in Section 10.3.5.

For this example, the declaration of the C procedure **time** looks like this:

```
long time (tloc)
long *tloc;
```

The first step in building the **INTERFACE** is to determine what attributes and type to use for the procedure. First, determine what FORTRAN type is equivalent to the type of the procedure **time**. The first word in the C procedure declaration, **long time (tloc)**; shows that **time** has type **long**. Referring to Table 10.11, "Signed 4-Byte Integers," in Section 10.3.10.2, "Integers," you can see that the FORTRAN **INTEGER*4** type is equivalent to the C **long** type. This gives enough information to write the following statement:

```
INTERFACE TO INTEGER*4 FUNCTION TIME
```

Second, decide which calling convention to use. Since you have no control over the C procedure, you must use the calling conventions that it uses. To specify the C calling conventions, use the C attribute as follows:

```
INTERFACE TO INTEGER*4 FUNCTION TIME [C]
```

Now determine what attributes and data types to use for the parameters. In this case, there is just one parameter, tloc. Therefore, you can write the following:

```
INTERFACE TO INTEGER*4 FUNCTION TIME [C]
+ (TLOC)
```

Note, however, that in the second line of the C procedure declaration, tloc is preceded by an asterisk, indicating that a pointer is being passed. You can pass a pointer from FORTRAN using the **LOCFAR** or **LOC** procedure, or you can pass the argument itself by reference. For now, assume that you want to pass by reference. FORTRAN normally defaults to passing by reference, but the procedure time is qualified by the C attribute, so tloc will default to being passed by value. To specify passing by reference, add the **REFERENCE** attribute, as shown below:

```
INTERFACE TO INTEGER*4 FUNCTION TIME [C]
+ (TLOC [REFERENCE])
```

The type of the parameter tloc is indicated by the first word in the second line of the C procedure declaration, long*tloc. Since the FORTRAN **INTEGER*4** type is equivalent to the C **long** type, you can finish the INTERFACE unit as follows:

```
INTERFACE TO INTEGER*4 FUNCTION TIME [C]
+ (TLOC [REFERENCE])
INTEGER*4 TLOC
END
```

If you decide to pass a pointer to tloc instead of passing it by reference, you proceed, in the same manner, to this point:

```
INTERFACE TO INTEGER*4 FUNCTION TIME [C]
+ (TLOC)
```

Pointers are passed by value, so do not specify the **REFERENCE** attribute. Since pointers are normally 4-byte segmented addresses, the result of **LOC** is a 4-byte integer, and therefore you must declare the parameter tloc to be a 4-byte integer:

```
INTERFACE TO INTEGER*4 FUNCTION TIME [C]
```

```
+ (TLOC)
INTEGER*4 TLOC
END
```

Step number three, adding the **INTERFACE** unit to your program, is identical for both cases. The only rule is that the **INTERFACE** must occur before any references to the procedure are made. It is usually easiest to put all **INTERFACE** statements at the beginning of the compiland.

The final step, calling the procedure, is different for the **REFERENCE** and pointer cases, as described in Section 10.3.5.

10.3.5 Calling Procedures in Pascal or C from FORTRAN

Once you have declared a procedure, you can call it in your program just as if it were in the same language as your calling procedure. Note that when calling from FORTRAN, you must always declare the procedure in the program units that use it.

For the example discussed in Section 10.3.4, start writing the calling routine like this:

```
SUBROUTINE CLOCK
INTEGER*4 TIME
INTEGER*4 TLOC
```

Don't forget to declare the procedure, as in the following line:

```
INTEGER*4 TIME
```

Now, if you passed `tloc` by reference, you can complete the call as follows:

```
SUBROUTINE CLOCK
INTEGER*4 TIME
INTEGER*4 TLOC
WRITE (*, *) TIME (TLOC)
END
```

If you passed a pointer, your procedure call looks like this:

```
SUBROUTINE CLOCK
INTEGER*4 TIME
INTEGER*4 TLOC
WRITE (*, *) TIME (LOC (TLOC) )
END
```

You could substitute the **LOCFAR** procedure for the **LOC** procedure. In this implementation they are identical.

Note that if `time` were a subroutine instead of a function, you could call that subroutine with the FORTRAN **CALL** statement.

10.3.6 Writing Interfaces to FORTRAN or C from Pascal

From Pascal, attach the **fortran** or **c** attribute to an **EXTERN** procedure declaration to interface with procedures written in FORTRAN or C.

10.3.7 Calling Procedures in FORTRAN or C from Pascal

Once you have declared a procedure, you can call it in your program just as if it were in the same language as your main program.

For example, the following Pascal program fragment calls `time`, passing `tloc` by reference:

```
FUNCTION time (VARS tloc:INTEGER4) : INTEGER4[C];
          EXTERN;
PROCEDURE clock;
  VAR tloc: INTEGER4;
BEGIN
  WRITELN (time(tloc))
END;
```

If you pass a pointer by value, the program fragment looks like this:

```
FUNCTION time (tloc:ADSMEM) : INTEGER4 [C]; EXTERN;
PROCEDURE clock;
  VAR tloc:INTEGER4;
BEGIN
  WRITELN(time(ADS tloc))
END;
```

10.3.8 Writing Interfaces to FORTRAN or Pascal from C

From C, the **fortran** and **pascal** keywords can be used to declare selected procedures written in, or compatible with, FORTRAN and Pascal. These keywords, which are enabled by default by the Microsoft C Compiler, imply changes in external naming, calling conventions, and return variable conventions.

If you want all procedures in your C program to be compatible with FORTRAN or Pascal, use the /Gc option when compiling.

FORTRAN and Pascal procedures are declared in the same manner as C procedures: you specify the procedure identifier, the return type, and the type and number of parameters to the procedure. (See the *Microsoft C Compiler Language Reference* for a complete discussion of the syntax of procedure declarations.)

The following additional rules apply when you use the **fortran** and **pascal** keywords:

1. Whenever a **fortran** or **pascal** keyword is used in a declaration, the types of parameters must be declared with a parameter-type list.
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 procedure declarations. The examples below illustrate the syntax of **pascal** and **fortran** declarations.

Examples

```
short pascal thing(short, short); /* Example 1 */  
long (pascal *thing)(void); /* Example 2 */  
short near pascal thing(short); /* Example 3 */  
short pascal near thing(short); /* Example 4 */
```

Example 1 declares `thing` to be a Pascal procedure taking two **short** parameters and returning a **short** value.

In Example 2, `thing` is declared as a pointer to a Pascal procedure that takes no parameters and returns a **long** value. Note that **void** is used to indicate that there is no return value.

Examples 3 and 4 are equivalent. Both declare `thing` to be a **near** Pascal procedure. The procedure takes one **short** parameter and returns a **short** value.

10.3.9 Calling Procedures in FORTRAN or Pascal from C

To call a Pascal or FORTRAN procedure from C, you must declare that procedure external, as in the following declaration:

```
extern void fortran m(long);
```

Note that **void** is used to indicate that there are no parameters.

Once you have declared a procedure, you can call it in your program just as if the procedure were in C.

10.3.10 Data Types

FORTRAN, Pascal, and C each have a variety of data types. Some are completely compatible; others require manipulation to work between languages. Sections 10.3.10.1 through 10.3.10.8 explain how specific data types differ in each language. Tables 10.7 through 10.27 show equivalent data types for each language.

10.3.10.1 Using the Equivalent Data Types Tables

To use tables 10.7–10.27 to pass parameters, you also have to refer to tables 10.4–10.6.

For example, suppose that you want to pass an **INTEGER*2** variable from FORTRAN to C. First, you have to choose a calling convention, as explained in Section 10.3.2, “Choosing a Calling Convention.” Assume that you want to use the C calling conventions. Refer to Table 10.4, “Passing Parameters With C Calling Conventions.”

Second, decide whether to pass the parameter by reference or by value. Assume that you want to pass the parameter by reference, using a short address. Table 10.4 shows that you use the **REFERENCE** and **NEAR** attributes in FORTRAN, and a **near** pointer of the appropriate type in C.

Third, determine what data type in C is equivalent to the **INTEGER*2** type in FORTRAN. Find the table that lists signed, 2-byte integers: Table 10.9. Note that **INTEGER*2** is listed as an appropriate FORTRAN data type. Check the "Notes" column to see if there is anything to be careful of when using **INTEGER*2**.

Now, look at the "C" row. You can choose between **short** and **int**, but the "Notes" column shows that **int** is machine dependent. For maximum portability, choose the C **short** type. Finally, apply the appropriate attributes and keywords to the data types, as follows:

```
INTEGER*2 X [REFERENCE, NEAR]
```

This statement in a FORTRAN **INTERFACE** declared with the **C** attribute is equivalent to a C parameter declared as

```
short near * x
```

Note that using a **REFERENCE** parameter in FORTRAN corresponds to using a pointer type in C.

10.3.10.2 Integers

In C, any integral parameters shorter than an **int** (such as **char**) are converted to **int** type before being passed by value. Unsigned integral types shorter than an **unsigned int** (such as **unsigned char**) are converted to **unsigned int** type.

To ensure that your FORTRAN or Pascal routine handles C parameters correctly, you have two options:

1. You can allow for the C conversions when you declare parameters to the FORTRAN or Pascal procedure. This means, for example, that all integer parameters must be declared to have the size corresponding to a C **int**, or **long int**, for integer parameters larger than an **int**.
2. You can pass pointers to the parameters instead of the values themselves (passing by reference). In the FORTRAN or Pascal routine, declare the passed parameters as a pointer to or reference

parameter of the appropriate type, then use the pointer to access the value indirectly.

Also, note that the C **int** type is machine specific. For the 8086 family of microprocessors, the C **int** type is equivalent to the following types:

- **INTEGER2** in Pascal
- **INTEGER*2** in FORTRAN
- **INTEGERC** in Pascal
- **INTEGER[C]** in FORTRAN

For any given processor and operating system, variables defined with the last two types are equivalent to variables of the C **int** type as defined by the Microsoft C Compiler for the same system. The last two types are therefore more portable than the first two.

Tables 10.7 through 10.11 show integer data types and their equivalents in Pascal, C, and FORTRAN.

Table 10.7
Signed 1-Byte Integers

Language	Data Type	Notes
Pascal	x:sint x:a..b	For $a \geq -127$ and $b \leq 127$
C	char x struct { char x;} } x	When passed by reference When passed by value
FORTRAN	None	

Table 10.8
Unsigned 1-Byte Integers

Language	Data Type	Notes
Pascal	x:byte x:wrd(a)..wrd(b) x:(a,b,...n)	For $0 \leq a \leq b$ For $b \leq 255$ For $\text{ord}(n) < 256$
C	unsigned char x struct { unsigned char x; } x	When passed by reference When passed by value
FORTRAN	CHARACTER*1 X	FORTRAN has no unsigned types, so you must use CHARACTER*1; use the ICHAR and CHAR functions to transfer values. Do not pass negative values.

Table 10.9
Signed 2-Byte Integers

Language	Data Type	Notes
Pascal	x:integer2 x:integerc x:integer	If \$INTEGER:2 (the default) is in effect
C	short x int x	Machine dependent
FORTRAN	INTEGER*2 X INTEGER[C] X INTEGER X	If \$STORAGE:2 is in effect

Table 10.10
Unsigned 2-Byte Integers

Language	Data Type	Notes
Pascal	x:word x:wrd(a)..wrd(b) x:(a,b,..n)	For $b > 255$ For ord (n) > 255
C	unsigned short x unsigned int x	Machine dependent
FORTRAN	INTEGER*2 X	FORTRAN has no unsigned types, so you must use INTEGER*2 . Do not pass negative values or values greater than 32767. Note that many unsigned operations can be performed safely on INTEGER*2 values.

Table 10.11
Signed 4-Byte Integers

Language	Data Type	Notes
Pascal	x:integer4 x:integer	If \$INTEGER:4 is in effect
C	long x	
FORTRAN	INTEGER*4 X INTEGER X	If \$STORAGE:4 (the default) is in effect

C also has unsigned 4-byte integers. FORTRAN and Pascal do not. However, many unsigned arithmetic operations can be performed on signed variables, and will yield correct results. This level of type equivalence may be sufficient for some applications.

10.3.10.3 Boolean and Character Types

For Pascal Boolean values, the integer one (1) means true. Zero (0) means false.

Tables 10.12 and 10.13 show how Boolean and character types, respectively, are represented in Pascal, C, and FORTRAN.

Table 10.12
Boolean Types

Language	Data Type	Notes
Pascal	x:boolean	
C	unsigned char x	
FORTRAN	CHARACTER*1 X	Use as for unsigned 1-byte integers; 1=false and 0=true. FORTRAN LOGICAL types are <i>not</i> equivalent. See tables 10.24 and 10.25 for FORTRAN LOGICAL types.

Table 10.13
Character Types

Language	Data Type
Pascal	x:char
C	unsigned char x
FORTRAN	CHARACTER X

10.3.10.4 Real Numbers

C passes all real parameters by value and as double-precision values. To ensure that your FORTRAN or Pascal routine handles C parameters correctly, you have the following three options:

1. You can allow for the C conversions when you declare parameters to the FORTRAN or Pascal procedure. This means that you must declare all floating-point parameters as double-precision parameters (**REAL*8** in FORTRAN, **real8** in Pascal), and specify the **VALUE** attribute in FORTRAN.
2. You can pass pointers to the parameters instead of the parameters themselves. In the FORTRAN or Pascal routine, declare the passed parameters as a pointer to the appropriate type, then dereference the pointer to access the value.
3. To avoid expansion of a **float** value to a **double**, you can pass the value as a structure. The members of structures do not undergo type conversion when the structure is passed as a parameter. For example, the following declaration defines a structure variable, **arg**, with a single **float** member:

```
struct fptype {float a;} arg;
```

The structure variable **arg** can then be passed as a parameter. Passing such a struct as a parameter in C is equivalent to pushing a **REAL*4** in FORTRAN (except that FORTRAN normally passes by reference) or a **real4** value in Pascal.

Floating-point values returned to C from Pascal or FORTRAN are handled as structured values.

Tables 10.14 and 10.15 show equivalent real types in Pascal, C, and FORTRAN.

Table 10.14
Single-Precision Real Numbers

Language	Data Type	Notes
Pascal	x:real4 x:real	If \$real:4 (the default) is in effect
C	float x struct { float x; } x	When passed by value
FORTRAN	REAL X REAL*4 X	

Table 10.15
Double-Precision Real Numbers

Language	Data Type	Notes
Pascal	x:real8 x:real	If \$real:8 is in effect
C	double x	
FORTRAN	REAL*8 X or DOUBLE PRECISION X	

10.3.10.5 Passing Strings

Pascal, FORTRAN, and C each store character strings in memory in a different way. In order to pass strings from one language to another, you must give the computer the appropriate information about how the string is set up.

C strings are considered arrays of characters. The null (zero-value) character marks the end of the string and is the last character of the array. For example, the string

String of text

is indicated in C as

```
unsigned char str[ ]="String of text"
```

This is stored in memory as a 15-byte array: 14 bytes of significant text (i.e., the string itself) and 1 null character that marks the end of the string:

S	t	r	i	n	g		o	f		t	e	x	t	\0
---	---	---	---	---	---	--	---	---	--	---	---	---	---	----

FORTTRAN strings do not have delimiters in memory. The length of the string is determined in advance. The above string is written in FORTTRAN as

```
STR='String of text'
```

It is stored in memory as 14 bytes of text:

S	t	r	i	n	g		o	f		t	e	x	t
---	---	---	---	---	---	--	---	---	--	---	---	---	---

Pascal has two forms of string: a fixed-length string type, **STRING**, which is the same as the FORTTRAN **string** type; and a variable-length string type, **LSTRING**. Using **LSTRING**, the string above is designated as follows:

```
VAR STR: LSTRING(14);
STR := 'String of text';
```

It is stored in memory as 15 bytes. The first byte indicates the number of bytes allocated in memory for the string; the remaining 14 bytes are the string itself:

14	S	t	r	i	n	g		o	f		t	e	x	t
----	---	---	---	---	---	---	--	---	---	--	---	---	---	---

Table 10.16 summarizes how each language handles string and array types. The placeholder *a* in the table is a constant, and each type occupies *a* bytes.

Table 10.16
String and Array Types

Language	Type
Pascal	c:STRING(<i>a</i>) c:ARRAY[1.. <i>a</i>] OF CHAR; c:LSTRING(<i>a</i> -1);
FORTRAN	CHARACTER* <i>a</i> C CHARACTER*1 C(<i>a</i>)
C	unsigned char c[<i>a</i>] struct cstr { unsigned char c[<i>a</i>];} c

Table 10.17 shows equivalent string types in each language.

Table 10.17
Strings

Language	Data Type	Notes
Pascal	x:array[1.. <i>n</i>] of char	
C	char x[<i>n</i>];	
FORTRAN	CHARACTER* <i>n</i> x INTEGER x ((<i>n</i> + 1)/2)	Not equivalent in future releases of FORTRAN. Not recommended. Can be equivalenced to a CHARACTER variable to allow access to individual bytes. This option will be equivalent in future releases, as well as in the present release.

Sections 10.3.10.5.1 through 10.3.10.5.3 explain how to pass strings from one language to another.

10.3.10.5.1 Passing FORTRAN Strings to C and Pascal

FORTRAN strings have the same format in memory as Pascal STRINGS, so you may pass them directly.

To pass FORTRAN strings to C, use the new C string feature. When a standard FORTRAN string constant is followed by the character C, that string is then interpreted as a C string constant. A null character is automatically appended to the end of the string, and backslashes (\) are treated as escapes. See the *Microsoft FORTRAN Compiler User's Guide* for information on the new C string feature.

Note

In subsequent releases of Microsoft FORTRAN, strings will be passed differently. In versions 3.3 and all earlier versions, the length of a string is not passed with the string. In later versions, the length of a string will be passed with the string as a super-array type. These two methods are incompatible.

If you are calling FORTRAN from C or Pascal and you are using strings, your calling code may have to be changed for a later version of Microsoft FORTRAN.

10.3.10.5.2 Passing Pascal Strings to C and FORTRAN

Since Pascal strings and FORTRAN strings have the same format in memory, you can pass them directly.

To pass Pascal **STRING** types to C, use concatenation to add an extra null byte to the end of the string. For example, if `strg` is a variable of type **STRING**, the null byte can be added as follows:

```
strg:"String of text"*&CHR(0);
```

Then `strg` can be passed to any C function that expects a string argument.

To pass variables of type **LSTRING** to C and FORTRAN, you must convert them to type **STRING** and handle the length byte yourself.

10.3.10.5.3 Passing C Strings to Pascal and FORTRAN

To FORTRAN and Pascal, C strings are just arrays. When passing C strings to Pascal and FORTRAN, allow room for the null byte at the end of the string.

10.3.10.6 Pointers

Tables 10.18 through 10.20 show equivalent pointer types for each language.

Table 10.18

Near Pointers

Language	Data Type	Notes
Pascal	<code>x:^t</code> <code>ADR t</code>	Machine dependent
C	<code>t near * x</code>	
FORTRAN	<code>T OBJECT</code> <code>INTEGER*2 X</code> <code>X=LOCNEAR(OBJECT)</code>	

Table 10.19

Far Pointers

Language	Data Type
Pascal	<code>ADS t</code>
C	<code>t * x</code> <code>t far * x</code>
FORTRAN	<code>T OBJECT</code> <code>INTEGER*4 X</code> <code>X=LOC(OBJECT)</code>
	<code>T OBJECT</code> <code>INTEGER*4 X</code> <code>X=LOCFAR(OBJECT)</code>

Table 10.20
Procedure Pointers

Language	Data Type	Notes
Pascal	x:adsproc x:adsfunc	You must declare the procedure public so that the ADS operator can get a far address. The compiler gives near addresses for local routines.
C	t (*x) ()	
FORTRAN	T PROC EXTERNAL PROC INTEGER*4 X X=LOC(PROC) T PROC EXTERNAL PROC INTEGER*4 X X=LOCFAR(PROC)	EXTERNAL must be used if the procedure name is used other than to invoke the function (in this example, the address of the procedure is taken). Otherwise, FORTRAN will create a new variable (with the same name) and take the address of that variable, rather than that of the procedure.

When using procedure pointers and calling a FORTRAN or Pascal routine from C with the C calling convention, use the following syntax to declare the procedure pointers in the argument declaration section of your C procedure:

returntype (x)(typeslist)*

The *returntype* is the C type of the return value. The *typeslist* is given with the same syntax used to declare the argument list of a **pascal** or **fortran** routine from C. When using the Pascal calling convention, use the following syntax:

*returntype (pascal * x)(typeslist)*

And when using the FORTRAN calling convention, use the following syntax:

*returntype (fortran * x)(typeslist)*

For example, you could pass a Pascal **ADSPROC** to the following C routine:

```
f(x)
short (pascal * x) (short);
```

In this example, *x* is a pointer to a **pascal** routine that takes a **short** and returns a **short**.

10.3.10.7 Arrays, Super Arrays and Huge Arrays

FORTRAN arrays are allocated in column order. *A(2,1)*, for example, is followed by *A(3,1)*. C and Pascal arrays are allocated in row order. *A(2,1)*, for example, is followed by *A(2,2)*.

The lower bound of indices to a C array is always 0. For FORTRAN, it is always 1. For Pascal it can be any value.

For example, if you define a C array *x[6][3]*, an equivalent array in FORTRAN would be *X(3,6)*. An equivalent Pascal array would be *x:array[0..5,0..2]*. If you specify element *x[5,0]* in Pascal, or element *x[5][0]* in C, the equivalent FORTRAN element is *X(1,6)*.

Or, if you define a Pascal array as

```
x:array[2..6,2..3] of integer2
```

the equivalent FORTRAN array is

```
INTEGER*2 X(2,5)
```

The equivalent C array is

```
short x[5][2]
```

FORTRAN large arrays (arrays specified with the **HUGE** attribute or the **\$LARGE** metacommand) cannot be used from Pascal or C.

In C, arrays are always passed by reference. If you use the **C** attribute from FORTRAN, arrays are passed by value, like C structs. That is, the entire array is laid out on the stack. To pass an array as an array (from FORTRAN to C), you must use the **REFERENCE** attribute, or pass the result of **LOC**, **LOCNEAR**, or **LOCFAR**.

Following are the two methods for using C arrays of two or more dimensions in FORTRAN or Pascal procedures:

1. Use the **typedef** statement to define a synonym, *name*, for the array *type* [*m*][*n*]..., as follows:

```
typedef type name[m][n] ...;
```

Declare the FORTRAN or Pascal procedure as

```
extern void fortran f(name);
```

or use

```
extern void pascal f(name);
```

In your main program, declare a variable of the type you have defined (*name*), then use that variable as the argument of the FORTRAN or Pascal procedure, as follows:

```
name x;
f(x);
```

2. Declare the FORTRAN or Pascal procedure as

```
extern void fortran f(type[m][n] ...);
```

or use

```
extern void pascal f(type[m][n] ...);
```

In your main program, declare a variable as follows:

```
type x[m][n];
```

Then use that variable as the argument of the FORTRAN or Pascal procedure, as follows:

```
f(x);
```

For example, using Method 1 above to pass a two-dimensional array, first define the synonym *shortarraytype* as follows:

```
typedef short shortarraytype[2][2];
```

The type *shortarraytype* is now equivalent to *short*[2][2]. Declare the Pascal procedure *p* by entering

```
extern void pascal p(shortarraytype);
```

In your main program, use the following to declare a variable *x* of type *shortarraytype*, then use *x* as the argument to the procedure *p*:

```
main()
{
    shortarraytype x;
    p(x);
}
```

Tables 10.21 and 10.22 show equivalent array types for Pascal, C, and FORTRAN.

Table 10.21
Arrays (Lower Bound of Pascal Array Is 0)

Language	Data Type	Notes
Pascal	x:array [0..j,0..m] of <i>type</i>	
C	<i>type</i> x[j+1][m+1] struct { <i>type</i> x[j+1][m+1];} x	When passed by reference When passed by value
FORTRAN	<i>type</i> x(m+1,j+1)	

Table 10.22
Arrays (Lower Bound of Pascal Array Is Nonzero)

Language	Data Type	Notes
Pascal	x:array [i..j,k..m] of <i>type</i>	
C	<i>type</i> x[j-i+1][m-k+1] struct { <i>type</i> x[j-i+1][m- k+1];} x	When passed by reference When passed by value
FORTRAN	<i>type</i> x(m-k+1,j-i+1)	

A super array pointer is a near pointer to the start of an array, followed by the upper bounds (in the same order as they are declared). Table 10.23 indicates how to specify a super array pointer from each language.

Table 10.23
Super Array Pointers

Language	Data Type	Notes
Pascal	<pre>type v=super array [0..*,0..*] of type x:^v</pre>	
C	<pre>struct { type near *ptr; short a; short b;} x:</pre>	Set <i>a</i> equal to (1st-dimension-of-target - 1). Set <i>b</i> equal to (2nd-dimension-of-target - 1).
FORTRAN	None	

10.3.10.8 Records and Structs

Pascal record types and C structs correspond fairly well. Variant records are more difficult, but can be used if you declare the tag field as a structure member, then build a union of all the variant parts.

In FORTRAN you can simulate a single instance of a record by using **EQUIVALENCE**, but there is no way to replicate the instance or apply such a structure to a parameter. If the record or struct contains only fields of the same size, you can use an array. Otherwise, you need to define an equivalence “group” with variables equivalenced so that they map to the appropriate elements of the struct. If the whole structure is less than 127 bytes, you can use a character variable to represent the whole structure. This means that you can assign a parameter with a single statement. This approach results in inefficient code and programs that are difficult to follow. It is recommended that you use Pascal and C to write interface procedures where possible. These could, for example, translate the structure into separate variables and scalars, which are easier to use with FORTRAN.

Note that you cannot pass a Pascal **set** type to FORTRAN.

Use Pascal records and C structs to correspond to FORTRAN **COMPLEX** data types, as shown in Tables 10.24 and 10.25.

Table 10.24
Single-Precision Complex Numbers

Language	Data Type
Pascal	x:record re, im:real; end;
C	struct complex8 { float re,im;} x
FORTRAN	COMPLEX X

Table 10.25
Double-Precision Complex Numbers

Language	Data Type
Pascal	x:record re, im:real8; end;
C	struct complex16 { double re,im;} x
FORTRAN	COMPLEX*16 X

Pascal records and C structs can also be used to pass FORTRAN logical values. For FORTRAN logical values, the integer one (1) means true. Zero (0) means false. Tables 10.26 and 10.27 give examples of passing FORTRAN logical values.

Table 10.26
Two-Byte LOGICAL Values

Language	Data Type	Notes
Pascal	x:record logical:boolean; pad: array[0..0] of byte; end;	
C	struct { char logical; char pad[1]; } x;	
FORTRAN	LOGICAL*2 X LOGICAL	If \$STORAGE:2 is in effect

Table 10.27
Four-Byte LOGICAL Values

Language	Data Type
Pascal	x:record logical:boolean; pad: array[0..2] of byte; end;
C	struct { char logical; char pad[3]; } x;
FORTRAN	LOGICAL*4 X

10.3.10.9 Procedural Parameters

Formal procedural arguments in Pascal and FORTRAN are compatible. They are not compatible with procedure pointers in C.

However, Pascal and FORTRAN procedure arguments can be represented by a C struct that mimics the Pascal/FORTRAN sequence.

If you are calling C from Pascal or FORTRAN, it is recommended that you use C procedure pointers. If you want to pass a procedure to a Pascal or FORTRAN procedure, you must use Pascal arguments, since neither Pascal nor FORTRAN can call through procedure pointers. See Table 10.20 for equivalent procedure-pointer types.

10.3.11 Return Values

FORTRAN and Pascal routines can return values to a C program. For the C program to handle the return values correctly, the programmer must understand the correspondence between data types in the different languages.

The C compiler performs conversions on return values before they are actually returned to the calling procedure. These conversions are the same as those given for parameters. Integral values shorter than an **int** are expanded to **int** size, and **float** values are converted to **double**. These types are discussed in sections 10.3.10.2, "Integers," and 10.3.10.4, "Real Numbers."

The C compiler detects structured return values that are 4 bytes or less in length and returns them as integers of the appropriate size.

10.3.12 Sharing Data

Pascal and C can refer to each other's public data items, as long as you specify appropriate attributes to use the correct naming conventions and keywords to ensure correct storage allocation. (All Pascal static variables should be declared with the **near** keyword in C.) FORTRAN **COMMON** blocks are public data areas and can be referenced as external data objects in C and Pascal. You can use the **COMMON** block names as the names of struct variables in C or record variables in Pascal, for example. To access a common block from Pascal, however, the common block must have the **NEAR** attribute. Blank common has the public name **COMMQQ**. FORTRAN *cannot* access C data objects.

Alternatively, you can use the **LOC** procedures in FORTRAN to give the address of a common block, pass the address to a C or Pascal procedure, then use that address from C or Pascal. The following example shows how this could be done:

```

INTERFACE TO SUBROUTINE CFUNC[C] (EXTP)
INTEGER*4 EXTP
END
COMMON/EXT/I,J
CALL CFUNC (LOC(I))
.
.
.
END

void cfunc (ext)
struct {long i,j;}*ext
{
    ext->i = ext ->j;
}

```

You can use the following method when you have several common blocks to set up:

```

SUBROUTINE SETADS (ADSEXT, ADSPAR, ADSBL)
INTEGER*4 ADSEXT,ADSPAR,ADSBL
COMMON/EXT/I1
COMMON/PAR/I2
COMMON I3
ADSEXT=LOCFAR (I1)
ADSPAR=LOCFAR (I2)
ADSBL=LOCFAR (I3)
END

long *ext, *par, *blank;
void fortran setads( long **, long **, long **);

main( )
{
    long dummy;

    setads( &ext, &par, &blank);

```

```
ext[0] = 100000;      /* Set FORTRAN common variable
                      ** I1 to 100000
                      */
.
.
.
}
```

10.3.13 Input and Output

A given file can be opened by only one language at a time, except for the standard output channel when that channel refers to the terminal. In this case, each FORTRAN **WRITE** statement that refers to the terminal should be followed by

```
WRITE(*,*)
```

if there is a possibility that a C or Pascal routine might write to the terminal immediately thereafter. This will clear the carriage-control character.

10.3.14 Compiling and Linking

The order in which modules are linked is important. You must make sure that you link them as follows:

1. If you are linking with the C floating-point library, it must be specified first.
2. If you are using Pascal or FORTRAN, their math libraries must be specified second. The math libraries for Pascal and FORTRAN are identical, and need be specified only once when you are mixing Pascal and FORTRAN.
3. If you are using Pascal or FORTRAN, their language libraries must be specified third.
4. If you are using the C-language library, it must be specified last, along with the code-helper library, **LIBH.LIB**.

10.3.15 Error Messages

Errors that occur during compile time are generated by the compiler for the language in which the error occurs. Most run-time errors come from the language in which the part of the program causing the error was written. On the other hand, floating-point errors may come from any of the languages used in the program. For Pascal and FORTRAN, these errors are identical. However, for C, floating-point error messages are slightly different, and there is no message number.

(

)

(

)

(

)

Appendices

- A ASCII Character Codes 269
- B Command Summary 271
- C The CL Command 289
- D Using EXEPACK, EXEMOD,
and SETENV 301
- E Using Exit Codes 309
- F Converting from Previous
Versions of the Compiler 317
- G Writing Portable Programs 345
- H Error Messages 363

()

()

()

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=Delete

Appendix A (*continued*)

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=Delete

Appendix B

Command Summary

B.1	Introduction	273
B.2	Compiler Summary	273
B.2.1	MSC Options	274
B.2.2	Pragmas	278
B.2.3	Standard Memory Models	278
B.2.4	Pointer and Integer Sizes	278
B.2.5	Segment Names	279
B.3	Linker Summary	280
B.3.1	Linker Command Characters	280
B.3.2	Linker Options	280
B.4	LIB Summary	283
B.5	MAKE Summary	284
B.5.1	MAKE Description Files	284
B.5.2	MAKE Options	285
B.5.3	Macro Definitions with MAKE	285
B.5.4	MAKE Inference Rules	286
B.6	EXEPACK Summary	286
B.7	EXEMOD Summary	287
B.8	SETENV Summary	288

1

2

3

B.1 Introduction

This appendix summarizes the commands and options available with **MSC** and the following Microsoft utilities: **LINK**, **LIB**, **MAKE**, **EXEPACK**, **EXEMOD**, and **SETENV**.

B.2 Compiler Summary

Command Line

MSC *sourcefile* [[,][*objectfile*]] [[,][*sourcelistfile*]] [[,][*objectlistfile*]]]]] [*options*] [;]

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.

With the command-line method, *sourcefile* is the name of the C source file and *objectfile* is the name of the object file produced by the compiler. The optional *sourcelistfile* is a listing file showing numbered source lines, error messages if encountered, and symbol-table information. The *objectlistfile* is a listing file showing the object code. The *options* appear anywhere a space can appear in the command line.

MSC uses three environment variables to locate the files it needs. Before invoking **MSC**, use the MS-DOS **PATH** and **SET** commands to assign a path name or names to the following variables:

Variable	Types of Files:
PATH	Executable compiler files
INCLUDE	Include files
TMP	Temporary files
LIB	Library files

B.2.1 MSC Options

The **MSC** options are listed in alphabetical order in this section. The dash (-) 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 <i>letter</i>	Sets the program configuration. The <i>letter</i> may be S , M , C , L , or H , standing for "small," "medium," "compact," "large," and "huge" models, respectively. The default is /AS .												
/A <i>string</i>	Sets the program configuration. The <i>string</i> consists of three characters in any order, one from each of the following groups:												
	<table border="0"> <thead> <tr> <th style="text-align: center;">Group</th> <th style="text-align: center;">Code</th> <th style="text-align: center;">Description</th> </tr> </thead> <tbody> <tr> <td style="vertical-align: top;">Code pointer size</td> <td style="text-align: center;">s l</td> <td>Small Large</td> </tr> <tr> <td style="vertical-align: top;">Data pointer size</td> <td style="text-align: center;">n f h</td> <td>Near Far Huge</td> </tr> <tr> <td style="vertical-align: top;">Segment setup</td> <td style="text-align: center;">d u w</td> <td>SS equal to DS SS not equal to DS, DS loaded for each module SS not equal to DS, DS fixed</td> </tr> </tbody> </table>	Group	Code	Description	Code pointer size	s l	Small Large	Data pointer size	n f h	Near Far Huge	Segment setup	d u w	SS equal to DS SS not equal to DS , DS loaded for each module SS not equal to DS , DS fixed
Group	Code	Description											
Code pointer size	s l	Small Large											
Data pointer size	n f h	Near Far Huge											
Segment setup	d u w	SS equal to DS SS not equal to DS , DS loaded for each module SS not equal to DS , DS fixed											
/C	Preserves comments when preprocessing a file (use only with /E , /P , or /EP).												
/D <i>identifier</i> [[= <i>string</i>]]	Defines <i>identifier</i> to the preprocessor. The value is <i>string</i> 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.
/Fs [[filename]]	Produces the source-listing 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.
/Gc	Generates the alternative (FORTRAN/Pascal style) call/return sequence and naming convention for an entire module.
/Gs	Removes calls to stack-probe routine.
/Gt [[number]]	Places data items greater than <i>number</i> bytes in new segment (256 bytes is the default); relevant only in compact-, large-, and huge-model programs.

/Gw

Generates Windows applications information (see your *Microsoft Windows Software Development Kit* for more information).

/H*number*

Restricts significant characters of external names to *number* characters.

/HELP

Lists the most common **MSC** options to the standard output. Any combination of uppercase and lowercase letters will work with this option; for example, **/help** or **/HelP** would work equally well.

/I*directory*

Adds *directory* to the top of the list of directories to be searched for include files.

/J

Changes the default for **char** type from signed to unsigned.

/ND *datasegmentname*

Sets the data segment name.

/NM *modulename*

Sets the module name.

/NT *textsegmentname*

Sets the text segment name.

/O*string*

Controls optimization. The *string* consists of one or more of the following characters:

Code	Description
d	Disables optimization
a	Relaxes alias checking
s	Favors code size
t	Favors execution time (default)
x	Maximizes optimization (equivalent to /Oas /Gs)

The default is **/Ot**.

/P

Preprocesses the source file and sends output to file with the base name of the source file and the extension **.I**.

/u	Removes definitions of all four predefined identifiers (MS_DOS , M_186 , M_186xM , and NO_EXT_KEYS).
/U <i>Identifier</i>	Removes definition of the given predefined identifier.
/V <i>string</i>	Copies <i>string</i> to the object file.
/w	Suppresses compiler warning messages.
/W <i>number</i>	Sets the output level (<i>number</i> = 0, 1, 2, or 3) for compiler warning messages.
/X	Ignores the list of “standard places” in the search for include files.
/Za	Disables language extensions. These include cdecl , far , fortran , huge , near , pascal , and other capabilities.
/Zd	Includes line-number and limited symbolic information in object file.
/Ze	Enables language extensions. These include cdecl , far , fortran , huge , near , pascal , and other extensions. This option is the default.
/Zg	Generates function declarations from function definitions and writes declarations to standard output.
/Zi	Enables full symbolic information for use with the Microsoft CodeView symbolic debugger.
/Zl	Removes default library information from object file.
/Zp	Packs structure members.
/Zs	Performs syntax check only.

B.2.2 Pragmas

The Microsoft C Compiler supports the `check_stack` pragma; this pragma instructs the compiler to turn stack checking on or off for selected routines, as explained in Section 9.10.1, “Removing Stack Probes.”

B.2.3 Standard Memory Models

Table B.1 defines the number of text and data segments for small, medium, compact, large, and huge memory models.

Table B.1
Text and Data Segments in Standard Memory Models

Model	Text Segments	Data Segments
Small	One	One
Medium	One per module	One
Compact	One	One default data segment*
Large	One per module	One default data segment*
Huge	One per module	One default data segment*

* The number of additional data segments depends on the program requirements.

B.2.4 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
Compact	32	16	16
Large	32	32	16
Huge	32	32	16

B.2.5 Segment Names

Table B.3 lists the default text and data segment names in the standard memory models. The default *modulename* is the file name.

Table B.3
Segment Names in Standard Memory Models

Model	Text	Data
Small	_ TEXT	_ DATA
Medium	<i>modulename</i> _ TEXT	_ DATA
Compact	_ TEXT	_ DATA*
Large	<i>modulename</i> _ TEXT	_ DATA*
Huge	<i>modulename</i> _ TEXT	_ DATA*

* Name of default data segment; other data segments have unique, private names

B.3 Linker Summary

Command Line

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

The Microsoft Overlay Linker, **LINK**, recognizes the command characters and options listed in this section. The files to be linked can be given either in response to prompts, in a command line, or in a response file.

If you use a command line, *objectfiles* are the names of object files, and *executablefile* is the executable program file produced by **LINK**. The optional *mapfile* is a listing of the names and addresses of segments, and optionally, public symbols. The optional *libraryfiles* are library files containing modules that must be linked with the object files. The *options* can appear anywhere in the command line.

LINK uses the environment variable **LIB** to locate library files. Before invoking **LINK**, use the MS-DOS **SET** command to assign a path name or names 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 interrupt the link session and return to MS-DOS.

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 **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 through the last letter 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 65535.
- 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 notation.
- 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 notation.

The linker options are listed in alphabetical order below. The options are abbreviated to the minimum length that distinguishes them from other **LINK** options:

Option	Task
/CO	Links a special-format executable file containing the symbolic information needed by the Microsoft Code-View debugger.
/CP:<i>number</i>	Sets the maximum memory allocation of program to <i>number</i> .
/DO	Enforces the following loading order: <ol style="list-style-type: none"> 1. All segments with a class name ending with CODE. 2. All other segments outside of DGROUP. 3. GROUP 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.

/DS	Tells LINK to load all data at the high end of the data segment. Do not use this option with C programs.
/E	Packs the executable file during linking.
/HE	Lists all LINK options to standard output.
/HI	Causes the run file to be placed as high as possible in memory. Do not use this option with C programs.
/L	Includes in the list file the line numbers and addresses of the source statements in the input modules.
/M	Creates a listing file containing all public (global) symbols defined in the input modules.
/NOD	Causes default libraries to be ignored.
/NOG	Provides compatibility with previous versions of LINK . Do not use this option with C programs.
/NOI	Causes the linker to distinguish between uppercase and lowercase letters.
/O: <i>number</i>	Sets the overlay interrupt number to <i>number</i> . In general, you should not use this option with C programs, with the exceptions outlined in Section 4.6.12, "Setting the Overlay Interrupt."
/P	Causes LINK to pause in the link session so you can change disks.
/SE: <i>number</i>	Sets the number of segments the linker allows a program to have. The default is 128.
/ST: <i>number</i>	Sets the stack size to <i>number</i> , which may be any positive value up to 65536 bytes. The default for C programs is 2K (2048 bytes).

B.4 LIB Summary

Command Line

```
LIB oldlibrary [/PAGESIZE:number] [commands] [,,[listfile] [,,[newlibrary]]] [;]
```

In the **LIB** command line, *oldlibrary* is the name of the library file to be processed, and *commands* are the commands indicating operations to be performed on the library file. The **/PAGESIZE** option can be used to change the page size (16 bytes by default). The *listfile* is a listing of modules and symbols within the library, and *newlibrary* is the name for a new library if you want to create one.

The following commands are recognized by the Microsoft Library Manager, **LIB**:

Command	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 MAKE Summary

Command Line

MAKE [[*options*]] [[*macrodefinitions*]] *makefilename*

The **MAKE** utility automates the process of updating program files. In the command line, *options* are one or more of the options described in Section B.5.2. The *macrodefinitions* are one or more macro definitions, as described in Section B.5.3. The *makefilename* is the name of a **MAKE** description file. A **MAKE** description file, by convention, has the same file name (but with no extension) as the program it describes. Although any file name can be used, this convention is preferred.

B.5.1 MAKE Description Files

A **MAKE** description file consists of one or more target/dependent descriptions. Each description has the following general form:

```
targetfile : dependentfiles [[# comment]]
[[# comment]]
    command [[# comment]]
    [command] [[# comment]]
    .
    .
    .
```

The *targetfile* is the name of a file that may need updating, *dependentfile* is the name of a file on which the target file depends, and *command* is the name of an executable file or an MS-DOS internal command. If a comment is on a separate line, the comment specifier (#) must be the first character on the line.

One way to remember the **MAKE** description format is to think of it as an "if-then" statement in the following format: if a *dependentfile* is older than the *targetfile*, or a *dependentfile* does not exist, then do *commands*.

B.5.2 MAKE Options

The options available with the **MAKE** command modify its behavior as follows:

Option	Effect
/D	Displays the last modification date of each file as the file is scanned
/I	Ignores exit codes returned by programs called from the MAKE description file; MAKE continues execution of the next lines of the description file despite the errors
/N	Displays commands that would be executed by a description file, but does not execute the commands
/S	Executes in “silent” mode; lines are not displayed as they are executed

B.5.3 Macro Definitions with MAKE

Macro definitions let you associate a symbolic name with a particular value. The form of a macro definition is as follows:

name=*value*

The form for using a previously defined macro definition is as follows:

`$(name)`

Occurrences of the pattern `$(name)` in the description file are replaced with the specified *value*. The *name* is converted to uppercase letters. If you define a macro name but leave *value* blank, *value* will be a null string. In the **MAKE** description file, each macro definition must appear on a separate line. Any white space (tab and space characters) between *name* and the equal sign (`=`), or between the equal sign and *value*, is ignored. Any other white space is considered part of *value*. To include white space in a macro definition on the command line, enclose the entire definition in double quotation marks ("").

If the same name is defined in more than one place, the following order of precedence applies:

1. Command-line definition
2. Description-file definition
3. Environment definition

MAKE recognizes the following special macro names and will automatically substitute a value for each:

Name	Value Substituted
\$*	Base-name portion of the target (without the extension)
\$@	Complete target name
\$**	Complete list of dependencies

B.5.4 MAKE Inference Rules

Inference rules take the following form:

```
.dependentextension.targetextension :  
    command  
    [command]  
    .  
    .  
    .
```

For lines that do not have explicit commands, **MAKE** looks for a rule that matches both the *targetextension* and the *dependentextension*. If it finds such a rule, **MAKE** performs the commands given by the rule.

B.6 EXEPACK Summary

Command Line

EXEPACK *executablefile* *outputfile*

The **EXEPACK** utility compresses sequences of identical characters from the given *executablefile* and optimizes the relocation table. The compressed file is written to the *outputfile*, and the original file is unmodified.

B.7 EXEMOD Summary

Command Line

EXEMOD *executablefile* [[/H]] [[/STACK *num*]] [[/MIN *num*]] [[/MAX *num*]]

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 *executablefile* without any options.

Option	Task
/STACK <i>num</i>	Allows you to set the size of the stack for your program by setting the initial SP (stack pointer) value to <i>num</i> , where <i>num</i> is a hexadecimal value in bytes. The minimum allocation value is adjusted upward if necessary. This has the same effect as the /STACK option for the linker, except that with EXEMOD the file is already linked.
/MIN <i>num</i>	Sets the minimum allocation value to <i>num</i> , where <i>num</i> 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 <i>num</i>	Sets the maximum allocation to <i>num</i> , where <i>num</i> is a hexadecimal value in paragraphs. The maximum allocation value must be greater than or equal to the minimum allocation value.
/H	Displays the current status of the MS-DOS program header. This has the same effect as entering EXEMOD without any options. The /H option should not be used with other options.

B.8 SETENV Summary

Command Line

SETENV *filename* [[*environmentsize*]]

The **SETENV** utility is used to modify **COMMAND.COM** in order to increase the size of the environment table. Normally, *filename* specifies **COMMAND.COM**. It must be a valid, unmodified copy of **COMMAND.COM**, though it could have a different name if you renamed it. The optional *environmentsize* is a decimal number specifying the size in bytes of the new allocation. **COMMAND.COM** normally allocates 10 paragraphs (160 bytes). The specified *environmentsize* will be rounded up to the nearest paragraph multiple.

If *environmentsize* is not given, **SETENV** will report the value that the **COMMAND.COM** file is currently allocating for the environment table.

Appendix C

The CL Command

C.1	Introduction	291
C.2	Command Syntax and Options	291
C.3	Linking with the CL Command	294
C.4	Additional Options	296
C.5	XENIX-Compatible Options	297

()

()

()

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.

CL uses four environment variables to locate the files it needs. Before invoking **CL**, use the MS-DOS **PATH** and **SET** commands to assign a path name or names to the following variables:

Variable	Types of Files
PATH	Executable compiler files
INCLUDE	Include files
TMP	Temporary files
LIB	Library files

C.2 Command Syntax and Options

Command Line

CL [[*options*] *filenames* [-link *libraryfield*]]

Each *option* in the command line is a command option, and each *filename* specifies a file to be processed. You can give more than one option or file name, 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 kinds of data you can pass in the *libraryfield*.

If you give the **CL** command with no arguments, **CL** displays a summary of the **CL** command-line syntax. If you provide arguments, each *filename* must be the name of a C-language source file or an object file. If the file name is for a source file, the file name must include the extension **.c** or **.C**. When **CL** processes the file, it looks at the file-name 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 file names 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 details.

An option consists of a dash (-) followed by a combination of one or more letters that have special meaning to **CL**. You can use a forward slash (/) instead of the dash as an option character if you prefer. The dash is used in this chapter for XENIX compatibility. All options available with the **MSC** command are also available with **CL**.

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**, **-Fs**, 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 File Name*	Default Extension
-Fs	Produces source listing	Base name of source file plus .LST	.LST
-Fa	Produces assembly listing	Base name of source file plus .ASM	.ASM
-Fc	Produces combined source-assembly listing	Base name of source file plus .COD	.COD
-Fe	Names the executable file	Base name of first source or object file on command line plus .EXE	.EXE
-Fl	Produces object listing	Base name of source file plus .COD	.COD
-Fm	Creates map file	Base name of first source or object file on the command line plus .MAP	.MAP

Table C.1 (*continued*)

Option	Task	Default File Name*	Default Extension
-Fo	Names object file	Base name of source file plus .OBJ	.OBJ

* The default file name for the **-Fs**, **-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 file name for the **-Fe** and **-Fo** options is used when the option is not given, or when a directory name is given as the argument to the option.

Table C.2
Arguments to -F Options

Options	File-Name Argument	Path-Name Argument	No Argument
-Fa , -Fc , -Fl , -Fs	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 file name 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 file name 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

Table C.2 (*continued*)

Options	File-Name Argument	Path-Name Argument	No Argument
-Fo	Uses given file name as the object-file name 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

Important

No spaces are allowed between the option and the argument (if any) for any of the **-Fx** 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 objectfiles [/CO]  

basename /NOI  

NUL;
```

Note 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 *objectfiles* are all object files produced in the compiling stage of the **CL** command, plus any object files specified on the **CL** command line. The **/CO** option (for the Microsoft CodeView symbolic debugger) is added to the first line of the response file if the **-Zi** option is given on the **CL** command line. The **/NOI** option tells the linker *not* to ignore case; uppercase and lowercase letters are considered different. By default, *basename* is the name supplied for the executable file; it corresponds to the base name 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. Note, however, that the **-Fm** option can be used in the **CL** command to override the default and produce a map file. A map file is also produced when the **-Zd** option is given on the **CL** command line; with **-Zd**, **CL** builds the following response file:

```
LINK objectfiles [/LI]  

basename /NOI  

basename;
```

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 *libraryfield* are searched before the default libraries.

The *libraryfield* can contain one or more of the following:

- A path name

The linker searches the given path name for the default libraries *before* searching directories given by the **LIB** variable.

- Additional or alternate library names

If a path name is included with the library name, only that path name is searched. Otherwise, the linker uses the standard library search path.

- 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.

- 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 4.6).

C.4 Additional Options

In addition to the **MSC** options summarized in Section B.2.1, "Command Summary," the **CL** command recognizes the options listed below. The options are shown with the dash (-) character for XENIX compatibility, but can be given with the forward slash (/) character if you prefer.

Option	Task
-c	Creates an object file for each source file on the command line; suppresses linking
-F <i>hexnumber</i>	Forces stack size to be set to <i>hexnumber</i> bytes; space required between -F and <i>hexnumber</i>
-Fe<i>programname</i>	Names the executable program file as <i>programname</i>
-Fm[<i>mapname</i>]	Creates a map file
-link <i>libraryfield</i>	Passes the specified <i>libraryfield</i> to the linker

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. The other XENIX options not specifically listed in Table C.3 are not supported by either **CL** or **MSC**.

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; disables the link step	Same; CL only
-C	Preserves comments when preprocessing a file (only when -P or -E).	Same
-dos	Performs cross-compilation to create DOS-executable file	Same (meaningful only on XENIX)
-D <i>name</i>[[=<i>string</i>]]	Defines <i>name</i> to the preprocessor. The value is <i>string</i> or 1.	Same
-E	Preprocesses each source file, copying the result to the standard output.	Same
-EP	Same as for -E , except does not put # line directives in output.	Same
-F <i>number</i>	Sets the size of the program stack. The given size must be a hexadecimal number.	Same; CL only
-I <i>pathname</i>	Adds <i>pathname</i> to the list of directories to be searched for # include files.	Same
-K	Removes stack probes from a program	-Gs

Table C.3 (continued)

XENIX Option	Task	MSC/CL Option
-L	Creates an object-listing file containing assembled object code and suppresses linking	-Fl -c; CL only
-M<i>string</i>	Sets the program configuration. The <i>string</i> may be any combination of s (small model), m (medium model), c (compact model), l (large model), h (huge model), e (enables far , near , huge , fortran , pascal , and cdecl keywords), 2 (enables 286 code generation), b (reverses word order for items of type long), t [<i>num</i>] (sets data threshold for largest item in a segment), and d (compiles program so that stack segment not equal to data segment). The s , m , c , l , and h options are mutually exclusive.	-Me is equivalent to -Ze . -M2 is equivalent to -G2 . -Mt [<i>num</i>] is equivalent to -Gt [<i>num</i>]. -Mb has no equivalent. -Ms is equivalent to -AS . -Mm is equivalent to -AM . -Mc is equivalent to -AC . -Md is equivalent to -Axxu . -Ml is equivalent to -AL . -Mh is equivalent to -AH .
-m <i>name</i>	Creates a map file	No equivalent in MSC; equivalent to -Fm<i>name</i> in CL
-nl<i>num</i>	Sets the maximum length of external symbols	-H<i>num</i>
-ND <i>name</i>	Sets the data-segment name	Same
-NM <i>name</i>	Sets the module name	Same
-NT <i>name</i>	Sets the text-segment name	Same
-o <i>filename</i>	Makes <i>filename</i> the name of the final executable program	-Fo<i>filename</i>
-O	Invokes the object-code optimizer	-Ot (default)
-P	Preprocesses source files and sends output to files with the extension .i	Same

Table C.3 (*continued*)

XENIX Option	Task	MSC/CL Option
-S	Creates an assembly source listing and suppresses linking	-Fa -c; CL only
-V <i>string</i>	Copies <i>string</i> to the object file	Same
-w	Suppresses compiler warning messages	Same
-W <i>number</i>	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

Using EXEPACK, EXEMOD, and SETENV

D.1	Introduction	303
D.2	The EXEPACK Utility	303
D.3	The EXEMOD Utility	304
D.4	The SETENV Utility	307

~

~

~

D.1 Introduction

The Microsoft EXE File Compression Utility, **EXEPACK**, and the Microsoft EXE File Header Utility, **EXEMOD**, supplied with the Microsoft C Compiler, allow you to modify executable program files. The Microsoft Environment Utility, **SETENV**, enlarges the MS-DOS environment table.

EXEPACK compresses executable files by removing sequences of repeated characters from the file and by optimizing the relocation table. **EXEMOD** allows you to examine and modify file-header information. **SETENV** allows you to use more and larger environment variables. The following sections explain how to use the **EXEPACK**, **EXEMOD**, and **SETENV** programs.

D.2 The EXEPACK Utility

EXEPACK compresses sequences of identical characters from a specified executable file and optimizes the relocation table. Using **EXEPACK**, you can reduce the size of some files and decrease the time required to load them.

EXEPACK will not always give a significant savings in disk space, and may sometimes actually increase file size because of an enhanced .EXE loader. Programs that have a large number of load-time relocations (about 500 or more) and long streams of repeated characters will usually be shorter if packed.

The **EXEPACK** program has exactly the same function as the **LINK /EXEPACK** option, except that **EXEPACK** works on files that have already been linked. One use for this utility is to pack the executable files provided with the Microsoft C Compiler. Some of the programs are already packed on your distribution disk. If you have floppy disks, you may want to pack all programs in order to make more room on your disks.

The **EXEPACK** command-line format is as follows:

EXEPACK *executablefile packedfile*

The *executablefile* is the file to be packed and *packedfile* is the name for the packed file. The *packedfile* should have a different name or be on a different drive or directory, since **EXEPACK** will not pack a file onto itself.

Do not try to get around the rule against packing a file onto itself by specifying the same file in a different way. You may be able to fool **EXEPACK**, but the result will be a damaged file. If you want the packed file to replace the original, you should use a separate name for the packed file, then delete the original and rename the packed copy.

When using **EXEPACK** to pack an executable overlay file or a file that calls overlays, the packed file should always be renamed with the original name to avoid the overlay manager prompt (see Section 4.5.2, "Overlay Manager Prompts").

Note

Using **EXEPACK** on a file containing symbolic debug information will remove that information from the file.

Example

```
EXEPACK WORK.EXE WORK.TMP  
DEL WORK.EXE  
RENAME WORK.TMP WORK.EXE
```

In this example, the executable file **WORK.EXE** is packed to a temporary file. The original is then deleted and the new packed version is renamed with the original name.

D.3 The EXEMOD Utility

EXEMOD modifies fields in the MS-DOS file header. In order to use this utility, you need to understand the MS-DOS conventions for file headers. They are explained in the *Microsoft MS-DOS Programmer's Reference Manual* and in some other reference books on MS-DOS.

Some of the options available with **EXEMOD** are the same as **LINK** options, except that they work on files that have already been linked. Unlike the **LINK** options, the **EXEMOD** options require that values be specified in hexadecimal.

To display the current status of the header fields, type the following:

EXEMOD executablefile

To modify one or more of the fields in the file header, type the following:

EXEMOD executablefile [/H] [/STACK num] [/MIN num] [/MAX num]

EXEMOD expects the *executablefile* to be the name of an existing file with the .EXE extension. If the file name is given without an extension, **EXEMOD** appends .EXE and searches for that file. If you supply a file with an extension other than .EXE, **EXEMOD** displays an error message.

The options in examples are shown with the forward slash (/) option designator, but a dash (-) may also be used. Options can be given in either upper- or lowercase, but they cannot be abbreviated. The options and their effects are described in the following list:

Option	Effect
/STACK num	Allows you to set the size of the stack for your program by setting the initial SP (stack pointer) value to <i>num</i> , where <i>num</i> is a hexadecimal value setting the number of bytes. The minimum allocation value is adjusted upward, if necessary. This option has the same effect as the LINK /STACK option, except that it works on files that are already linked.
/MIN num	Sets the minimum allocation value to <i>num</i> , where <i>num</i> is a hexadecimal value setting the number of paragraphs. The actual value set may be different from the requested value if adjustments are necessary to accommodate the stack.
/MAX num	Sets the maximum allocation to <i>num</i> , where <i>num</i> is a hexadecimal value setting the number of paragraphs. The maximum allocation value must be greater than or equal to the minimum allocation value. This option has the same effect as the LINK /CPARMAXALLOC option.
/H	Displays the current status of the MS-DOS program header. Its effect is the same as entering EXEMOD with an <i>executablefile</i> but no options. The /H option should not be used with other options.

Note

The **/STACK** option can be used on programs assembled with **MASM** or programs compiled with the Microsoft C Compiler versions 3.0 and later, the Microsoft Pascal Compiler versions 3.3 and later, or the Microsoft FORTRAN Compiler versions 3.3 and later. Use of the **/STACK** option on programs developed with other compilers may cause the programs to fail, or **EXEMOD** may return an error message.

EXEMOD works on packed files. When it recognizes a packed file, it will print the following message:

`exemod: (warning) packed file`

It will then continue to modify the file header.

When packed files are loaded, they are expanded to their unpacked state in memory. If the **EXEMOD /STACK** option is used on a packed file, the value changed is the value that **SP** will have after expansion. If either the **/MIN** or **/STACK** option is used, the value will be corrected as necessary to accommodate unpacking of the modified stack. The **/MAX** option operates as it would for unpacked files.

If the header of a packed file is displayed, the **CS:IP** and **SS:SP** values are displayed as they will be after expansion, which is not the same as the actual values in the header of the packed file.

Examples

EXEMOD TEST.EXE

TEST.EXE

	(hex)	(dec)
Minimum load size (bytes)	419D	16797
Overlay number	0	0
Initial CS:IP	0403:0000	
Initial SS:SP	0000:0000	0
Minimum allocation (para)	0	0
Maximum allocation (para)	FFFF	65535
Header size (para)	20	32
Relocation table offset	1E	30
Relocation entries	1	1

The first example shows the file header for file TEST.EXE. The following command line shows how to modify the header:

```
EXEMOD TEST.EXE /STACK FF /MIN FF /MAX FFF
```

The following example shows a display of values after the modification:

TEST.EXE	(hex)	(dec)
Minimum load size (bytes)	528D	20877
Overlay number	0	0
Initial CS:IP	0403:0000	
Initial SS:SP	0000:0OFF	256
Minimum allocation (para)	FF	256
Maximum allocation (para)	FFF	4095
Header size (para)	20	32
Relocation table offset	1E	30
Relocation entries	1	1

D.4 The SETENV Utility

The **SETENV** utility allows you to allocate more environment space to MS-DOS by modifying a copy of **COMMAND.COM**.

Normally, MS-DOS versions 2.0 and later allocate 160 bytes (10 paragraphs) for the environment table. This may not be enough if you want to set numerous environment variables using the **SET** or **PATH** command. For example, if you have a hard disk with several levels of subdirectories, a single environment variable might take 40 or 50 characters. Since each character uses 1 byte, you could easily require more than 160 bytes, if you wanted to set several environment variables.

Note

SETENV is guaranteed to work only with PC-DOS versions 2.0, 2.1, 3.0, and 3.1. **SETENV** may or may not work with other versions of MS-DOS. Moreover, you should not use **SETENV** with versions of MS-DOS later than Version 3.1. Consult your MS-DOS manual for information on how to increase environment size in these later versions.

To enlarge the environment table, you must use **SETENV** to modify a copy of **COMMAND.COM**. Make sure you work on a copy and retain an unmodified version of **COMMAND.COM** for backup.

The command line for modifying the environment table is as follows:

SETENV *filename* [[*environmentsize*]]

Normally *filename* specifies **COMMAND.COM**. It must be a valid, unmodified copy of **COMMAND.COM**, though it could have a different name if you renamed it. The optional *environmentsize* is a decimal number specifying the size in bytes of the new allocation; *environmentsize* must be a number greater than or equal to 160, and less than or equal to 65520. The specified *environmentsize* will be rounded up to the nearest multiple of 16 (the size of a paragraph).

If *environmentsize* is not given, **SETENV** will report the value that the **COMMAND.COM** file is currently allocating.

After modifying **COMMAND.COM**, you must reboot so that the environment table will be set to the new size.

Examples

SETENV **COMMAND.COM**

Microsoft (R) Environment Expansion Utility Version 2.00
Copyright (C) Microsoft Corp 1985. All rights reserved.

command.com: Environment allocation = 160

In the first example, no environment size is specified, so **SETENV** reports the current size of the environment table.

SETENV **COMMAND.COM** 605

In the second example, an environment size of 605 bytes is requested. Since 605 bytes is not on a paragraph boundary (a multiple of 16), **SETENV** rounds the request up to 608 bytes. **COMMAND.COM** is modified so that it will automatically set an environment table of 608 bytes (38 paragraphs). You must reboot to set the new environment-table size.

Appendix E

Using Exit Codes

E.1	Introduction	311
E.2	Exit Codes with MAKE	311
E.3	Exit Codes with MS-DOS Batch Files	311
E.4	Exit Codes for Programs in the C Compiler Package	312
E.4.1	Compiler Exit Codes	312
E.4.2	LINK Exit Codes	313
E.4.3	CodeView Exit Codes	313
E.4.4	LIB Exit Codes	314
E.4.5	MAKE Exit Codes	314
E.4.6	EXEPACK Exit Codes	314
E.4.7	EXEMOD Exit Codes	314
E.4.8	SETENV Exit Codes	315

()

()

()

E.1 Introduction

All the programs in the Microsoft C Compiler package return an exit code (sometimes called an “errorlevel” code) that can be used by MS-DOS batch files or other programs such as **MAKE**. If the program finishes without errors, it returns a code of 0. The code returned varies if the program encounters an error. This appendix discusses several uses for exit codes, and lists the exit code numbers that can be returned by each program in the Microsoft C Compiler package.

E.2 Exit Codes with **MAKE**

MAKE automatically stops execution if a program executed by one of the commands in the **MAKE** description file encounters an error. The exit code is displayed as part of the error message.

For example, assume the **MAKE** description file TEST contains the following lines:

```
TEST.OBJ :      TEST.C  
              MSC TEST;
```

If the source code in TEST.C contains a program error (but not if it contains a warning error), you would see the following message the first time you use **MAKE** with the **MAKE** description file TEST:

```
make: MSC TEST; - error 2
```

This error message indicates that the command `MSC TEST;` in the **MAKE** description file returned code 2.

E.3 Exit Codes with MS-DOS Batch Files

If you prefer to use MS-DOS batch files instead of **MAKE** description files, you can test the code returned with the **IF ERRORLEVEL** command. The sample batch file following, called `COMPILE.BAT`, illustrates how:

```
MSC %1;  
IF NOT ERRORLEVEL 1 LINK %1;  
IF NOT ERRORLEVEL 1 %1
```

You can execute this sample batch file with the following command:

```
COMPILE TEST
```

MS-DOS then executes the first line of the batch file, substituting TEST for the parameter %1, as in the following command line:

```
MSC TEST;
```

It returns a code of 0 if the compilation is successful, or a higher code if the compiler encounters an error. In the second line, MS-DOS tests to see if the code returned by the previous line is 1 or higher. If it is not (that is, if the code is 0), MS-DOS executes the following command:

```
LINK TEST;
```

LINK also returns a code, which will be tested by the third line.

E.4 Exit Codes for Programs in the C Compiler Package

An exit code of 0 always indicates execution of the program with no fatal errors. Warning errors also return exit code 0. Some programs can return various codes indicating different kinds of errors, while other programs return only 1 to indicate that an error occurred. The exit codes for each program are listed in sections E.4.1 through E.4.8.

E.4.1 Compiler Exit Codes

Code	Meaning
0	No fatal error
2	Program error
4	System level error (such as out of disk space or compiler internal error)

E.4.2 LINK Exit Codes

Code	Meaning
0	No error
1	All LINK fatal errors not listed below
16	Data record too large
32	No object modules specified
33	Cannot open list file
66	Common area longer than 65536 bytes
96	Too many libraries
144	Invalid object module
145	Too many TYPDEFs
146	Too many group, segment, and/or class names in one module
147	Too many segments, or too many segments in one module
148	Too many overlays
149	Segment size exceeds 64K
150	Too many groups or too many GRPDEFs in one module
151	Too many external symbols in one module
177	Group larger than 64K

E.4.3 CodeView Exit Codes

The Microsoft CodeView debugger does not return exit codes. However, it does display return codes returned by programs run within the debugger. For example, if you run an executable file called TEST.EXE from within the CodeView debugger and the program encounters an error that returns 1, you will see the following line:

```
Program terminated normally (1)
```

E.4.4 LIB Exit Codes

Code	Meaning
0	No error
1	All LIB fatal errors not listed below
4	Internal error
13	Too many symbols
16	Page size too small

E.4.5 MAKE Exit Codes

Code	Meaning
0	No error
1	Any MAKE fatal error

If a program called by a command in the **MAKE** description file produces an error, the exit code will be displayed in the **MAKE** error message.

E.4.6 EXEPACK Exit Codes

Code	Meaning
0	No error
1	Any EXEPACK fatal error

E.4.7 EXEMOD Exit Codes

Code	Meaning
0	No error
1	Any EXEMOD fatal error

E.4.8 SETENV Exit Codes

Code	Meaning
0	No error
1	Any SETENV fatal error

)

)

)

Appendix F

Converting from Previous Versions of the Compiler

F.1	Introduction	319
F.2	Differences between Versions 3.0 and 4.0	319
F.2.1	Enhancements and Additions to the Compiler Software	320
F.2.2	Changes in the Language Syntax	320
F.2.3	New Features for the MS-DOS Implementation of C	322
F.2.4	New Library Routines and Include Files	323
F.2.5	Changes in Library-Routine Syntax	324
F.3	Differences Between Version 4.0 and Versions Prior to 3.0	324
F.3.1	Language-Definition Differences	325
F.3.2	Run-Time-Library Differences	330
F.3.2.1	abs	331
F.3.2.2	creat	332
F.3.2.3	fopen, freopen	333
F.3.2.4	iscsym, iscsymf	333
F.3.2.5	max	333
F.3.2.6	min	334
F.3.2.7	movmem	334
F.3.2.8	open	334
F.3.2.9	setmem	335
F.3.2.10	setnbuf	335

F.3.2.11	stcis, stcisin, stclen, stpbrk, stpchr, stscmp	335
F.3.3	Differences in Assembly-Language Interface	336
F.3.3.1	Register-Usage Conventions	336
F.3.3.2	Stack Setup and Subroutine Entry/Exit Code	338
F.3.3.3	Global-Variable Naming Conventions	341
F.3.3.4	Segment Usage and Naming	342

F.1 Introduction

This appendix describes differences between Version 4.0 of the Microsoft C Compiler and earlier versions of the Microsoft C Compiler. If you have an earlier version of the compiler, or if you have written programs for an earlier version, this chapter can help you convert your previous source code. The actions necessary to convert source code depend on which earlier version you have.

Version 4.0 is an update of Version 3.0. Generally, the two versions are compatible: your C source code written for Version 3.0 should compile without change on the Version 4.0 compiler, although there are cases of erroneous C constructs allowed in Version 3.0 that are not allowed in Version 4.0, and changes in the emerging ANSI C standard may force changes in the treatment of octal and hexadecimal constants (for more information, see the *Microsoft C Compiler Language Reference*). In some cases you may be able to enhance your programs by revising them to take advantage of new library routines and other features available with Version 4.0.

On the other hand, the compiler in versions prior to 3.0 is completely different from the compiler in versions 3.0 and 4.0. Source code for these versions may require significant changes. The differences fall into three categories: language definition differences, run-time-library differences, and assembly-language-interface differences.

F.2 Differences between Versions 3.0 and 4.0

The Microsoft C Compiler, Version 4.0, has been enhanced to provide more flexible programming and to match the emerging ANSI standard for the C language. Changes fall into the following categories:

- Enhancements and additions to the compiler software to allow for more flexible programming, improved code generation, and increased support for the developing ANSI standard
- Changes in the language syntax
- New language features specific to the MS-DOS implementation
- New library routines and include files

These features and the changes required to take advantage of them are discussed in the next few sections.

F.2.1 Enhancements and Additions to the Compiler Software

The compiler software has been enhanced to make it easier to use. Enhancements include the following:

- New options for **MSC** and **LINK**
- Improved code optimization
- New memory models (**compact** and **huge**)
- Source listings
- Numbered error messages
- Huge arrays, allowing a single array to be larger than 64K
- Three new utilities: **MAKE**, **SETENV**, and **CODEVIEW**

These changes should have no effect on your source code, but you may need to revise existing batch files or **MAKE** description files to make them work correctly with Version 4.0.

See Chapter 3, "Compiling," for information on changes to the syntax of the **MSC** command line.

F.2.2 Changes in the Language Syntax

Some changes have been made to the C language syntax to make it conform more closely to the new ANSI standard. Most of these changes do not affect source code written for the Version 3.0 compiler. The changes are summarized below:

- The `\a` escape sequence now represents the bell (or alert) character. You can make your source code more portable by using `\a` instead of `\x7`. See Section 2.24, "Escape Sequences," of the *Microsoft C Compiler Language Reference*.
- The **signed** keyword has been added. The **signed** keyword can be used to specify signed items. This keyword is particularly useful for declaring signed **char** types in programs compiled with the **/J** option (**/J** changes the default mode for the **char** type to **unsigned**). See Section 4.2, "Type Specifiers," of the *Microsoft C Compiler Language Reference*.

- The syntax for making function calls with a variable number of arguments has changed. The following two declarations contrast the old form and the new form:

```

int func (int,);      /* Forward declaration in
                      ** old syntax
                      */
int func (int,...);   /* Forward declaration in
                      ** new syntax
                      */

```

This change was made to conform to changes in the ANSI standard for the C language. Both forms are supported in Version 4.0 of the Microsoft C Compiler, so you do not have to change existing source code. Microsoft recommends the use of the new form in all programs.

- The compiler formerly allowed arbitrary strings of characters after a syntactically correct preprocessor command. To conform to the new ANSI standard, this is no longer allowed, and causes the compiler to generate the following warning message:

```
#endif      Block ends here
```

Such strings must now be enclosed in comment delimiters, as in the following example:

```
#endif      /* Block ends here */
```

- Names of types defined with **typedef** are no longer considered keywords, as they were in Version 3.0. These names are now in the same naming class as names of functions and variables, and can be redefined in a nested block. See Section 3.6, “Naming Classes,” of the *Microsoft C Compiler Language Reference*.
- The **#pragma** directive is now supported. A “pragma” is an instruction to the compiler. Its syntax is similar to the syntax of preprocessor directives, but its purpose is different. The syntax is as follows:

```
# pragma charstring
```

The only pragma instruction supported in the Microsoft C Compiler, Version 4.0, is the **check_stack** pragma. This pragma is specific to MS-DOS, and is discussed in greater detail in Section 9.10.1, “Removing Stack Probes.”

- Hexadecimal and octal integer constants are handled differently in Version 4.0 than in Version 3.0. See the *Microsoft C Compiler Language Reference* for more information.
- The extended keywords **fortran**, **pascal**, **cdecl**, **far**, **near**, and **huge** are enabled by default in Version 4.0. They can be disabled by giving the /Za option on the command line.
- Two new reserved words, **const** and **volatile**, will be implemented in future releases.
- In Version 3.0, when a short pointer is converted to type **long int**, it is first converted to type **short int**, then to **long int**; as a result, in Version 3.0 the expression in the **if** statement evaluates as true in the following fragment:

```
char *ptr = NULL;
long i;

i = (long) ptr;
if (i == 0L) {
    .
    .
    .
}
```

In Version 4.0, the conversion of short pointers to long integers has been changed so it conforms to the order in which the compiler does all other conversions that increase the length of a variable, namely, first the size, then the mode. (For example, the compiler converts a variable with type **char** to type **unsigned long** by first converting it to **signed long**, then to **unsigned long**.) Because of this change, the preceding code now converts **ptr** to a far pointer by loading the appropriate segment register value, then changing that to a long integer. The expression following the **if** statement would most likely be false in Version 4.0, since the segment registers do not usually contain zero.

F.2.3 New Features for the MS-DOS Implementation of C

The MS-DOS implementation of the C compiler has been enhanced to include the following features:

- Two new memory models: huge and compact
- The **huge**, **signed**, and **cdecl** keywords
- A pragma (**check_stack**) to control stack checking
- The **/J** option to change the default mode for the **char** type to unsigned
- The **/Gc** option to specify the alternate call/return sequence and naming conventions used in Microsoft Pascal and Microsoft FORTRAN

All these features are discussed in Chapter 8, “Working with Memory Models,” and Chapter 9, “Advanced Topics.” In most cases, they will not affect existing source code. However, you may be able to improve your existing programs by modifying them to take advantage of the new memory models or the **huge** keyword.

F.2.4 New Library Routines and Include Files

New library routines and include files have been added to Version 4.0 of the Microsoft C Compiler. In some cases you may wish to modify existing source code to take advantage of new library routines and include files. The new library routines are listed below:

alloca	fmsbintoieee	_nmalloc	strnicmp
_clear87	_fmsize	_nmsize	strstr
_control87	_fpreset	onexit	strtod
dieeetomsbin	_freect	remove	strtol
difftime	_malloc	rmtmp	tempnam
dmsbintoieee	hfree	setvbuf	tmpfile
execpe	lfind	spawnlpe	tmpnam
execvpe	lsearch	spawnvpe	vfprintf
_expand	_memavl	stackavail	vprintf
_ffree	memicmp	_status87	vsprintf
fieeetomsbin	_msize	strerror	
_fmalloc	_nfree	strcmp	

The new include files are listed below:

File	Purpose
float.h	Defines values used in floating-point operations
limits.h	Defines upper and lower limits for various types

stdarg.h	Defines a complete set of typedefs and macros that can be used to write portable programs that can handle functions with variable-length argument lists; designed to be compatible with the proposed ANSI standard for C
stddef.h	Defines standard values such as NULL and errno
varargs.h	Defines a complete set of typedefs and macros that can be used to write portable programs that can handle functions with variable-length argument lists; designed to be compatible with UNIX System V

For more information about the new library routines and include files, see the *Microsoft C Compiler Library Reference*.

F.2.5 Changes in Library-Routine Syntax

In order to conform to the developing ANSI standard, the order of the parameters in the **rename** function has been changed. The syntax for Version 3.0 is as follows:

rename(*newname*, *oldname*)

The following is the syntax for Version 4.0:

rename(*oldname*, *newname*)

F.3 Differences Between Version 4.0 and Versions Prior to 3.0

The changes made since 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 were eliminated or changed in Version 3.0. The changes in versions 3.0 and later 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 Microsoft C programs developed with versions prior to Version 3.0 under more recent versions of the compiler.

The following sections describe language, run-time-library, and assembly-language differences in detail, and outline strategies for converting existing programs.

F.3.1 Language-Definition Differences

This section lists differences in the definition of the C language between versions 3.0 and later, and prior 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 versions 3.0 and later. 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 versions 3.0 and later. In versions 3.0 and later, you can suppress compilation of program sections that contain comments by using a preprocessor directive (#if).
2.2	Under versions 3.0 and later, 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 versions 3.0 and later.
2.4.3	Multicharacter char constants are allowed under versions 2.03 and earlier, but are not permitted under versions 3.0

and later. Code containing multicharacter **char** constants will not compile correctly under versions 3.0 and later.

2.5

Every C string is unique. A string can initialize an array and can be modified at run time. Versions 3.0 and later give every string separate storage, whether or not a string is identical to another string in the program. Versions 2.03 and earlier detect whether or not 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 versions 3.0 and later.

4

Versions 3.0 and later implement the **char** type as a signed quantity, and provide 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 versions 3.0 and later.

Versions 3.0 and later implement the **unsigned long** type, a feature not provided in previous versions.

The enumeration type is also provided in versions 3.0 and later. Previous versions did not offer this feature.

7.1

In versions 3.0 and later, 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 versions 3.0 and later.

In versions 3.0 and later, 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 versions 3.0 and later.

- 7.2 In versions 3.0 and later, 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 versions 3.0 and later. In versions 2.03 and earlier, the right operand is converted to the type of the left operand.
- 8.5 Versions 3.0 and later allocate bit fields low order to high order, whereas versions 2.03 and earlier allocate bit fields high order to low order.
- 11.2 Versions 3.0 and later differ from earlier versions in their treatment of uninitialized variables declared outside 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 is initialized to 0 when the program is loaded. In previous versions, the variable was allocated storage and initialized at compile time.

12

Versions 3.0 and later add several new features to the C preprocessor. The special constant expression **defined(*identifier*)** can follow any **#if** or **#elif** directive. For example, the following two lines have the same effect:

```
#if defined(ANYTHING)  
#ifdef ANYTHING
```

The new **#elif** directive allows for “else-if” clauses in **#if** blocks.

In versions 3.0 and later, the number sign (#) introducing the preprocessor directive can be preceded on the same line by any combination of tabs and spaces. In previous versions, the number sign had to be the first character of the line.

Macro definitions can occupy more than one line in versions 3.0 and later. The new-line character is preceded by a backslash (\) to indicate continuation. Earlier versions do not allow continuation.

14.1

Under versions 3.0 and later, 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

Versions 3.0 and later allow 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]) {
        .
        .
        .
    }
}

```

Versions 3.0 and later accept 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

Versions 3.0 and later make 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 versions 3.0 and later 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.

F.3.2 Run-Time-Library Differences

Many of the library routines documented in versions 2.03 and earlier will run without change under versions 3.0 and later. However, some routines are supported in versions 3.0 and later 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 versions 3.0 and later. 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 under versions 3.0 and later. The following routines from Version 2.03 are *not* supported in any form under versions 3.0 and later:

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 versions 3.0 and later 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	stcism	stscmp
fopen	max	setmem	stclen	
freopen	min	setnbuf	stnbrk	

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.

F.3.2.1 **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**.

F.3.2.2 creat

The previous version of this function differs from the versions 3.0 and later in two ways. In versions 3.0 and later, 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 through the use of the bitwise-OR operator (`|`).

Versions 3.0 and later maintain 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**. The **BINMODE.OBJ** file is discussed in Section 9.12, "Controlling Binary and Text Modes."

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 versions 3.0 and later; **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 VERSIONS 3.0 AND LATER */
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 versions 3.0 and later. 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.

F.3.2.3 `fopen`, `freopen`

In versions 3.0 and later, when a file is opened for appending ("a" or "at" type), all write operations occur at the end of the file. Although the file pointer can be repositioned using `fseek` or `rewind`, the file pointer is always returned to the end of the file before any write operation is carried out.

In versions 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 occur 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.

F.3.2.4 `iscsym`, `iscsymf`

These macros are extensions to the character-classification (`ctype`) macros. Versions 3.0 and later do 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 versions 3.0 and later do 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.

F.3.2.5 `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`.

F.3.2.6 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**.

F.3.2.7 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)
```

F.3.2.8 open

The **open** routine has the same basic form and function in versions 3.0 and later as it does in earlier versions, with the following 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.

F.3.2.9 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)
```

F.3.2.10 setnbuf

The **setnbuf** routine sets up an empty buffer and is equivalent to the following call:

```
setbuf (stream, NULL);
```

Versions 3.0 and later support the **setnbuf** routine through the following definition in **v2tov3.h**:

```
#define setnbuf (stream)      setbuf (stream, NULL)
```

F.3.2.11 stcisin, stcisin, stclen, stpbrk, stpchr, stscmp

These routines are renamed in versions 3.0 and later, but otherwise function the same as in versions 2.03 and earlier. The names in versions 3.0 and later are as follows:

Version 2.03 Name	Version 3.0 Name
stcisin	strspn
stcisin	strcspn
stclen	strlen
stpbrk	strupbrk
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 stcisin(s1, s2)      strspn(s1, s2)
#define stcism(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)
```

F.3.3 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 versions 3.0 and later. Much of the information in this section is also presented in Section 10.2, "Assembly-Language Interface"; this discussion attempts to consolidate the information to make the task of conversion easier. For additional assembly-language information not found below, see Section 10.2.

Assembly-language routines that are compatible with versions 2.03 and earlier differ from Version 3.0-compatible routines in the following 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.

F.3.3.1 Register-Usage Conventions

The S- and P-model programs of Version 2.03 correspond to the small- and medium-model programs of versions 3.0 and later. 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 versions 3.0 and later allows data in segments outside **DS** to be accessed. In mixed-model programs, the compiler uses **ES** to reference data outside the data segment (**DS**). Therefore, **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** flag in the **/A** option.)

Versions 3.0 and later also expect 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.

Versions 3.0 and later implement register variables. These were not available in previous versions. The Version 4.0 (and 3.0) compilers allow 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 handles 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 F.3.3.2 for an example.

F.3.3.2 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, versions 3.0 and later differ 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 F.1 and F.2 show the stack frame setups under Version 2.03 and Version 3.0, respectively.

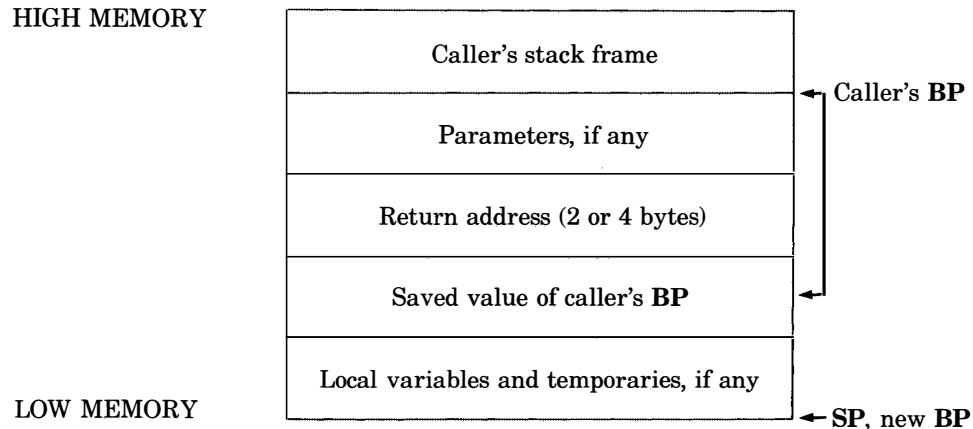


Figure F.1 Version 2.03 Stack Frame Setup

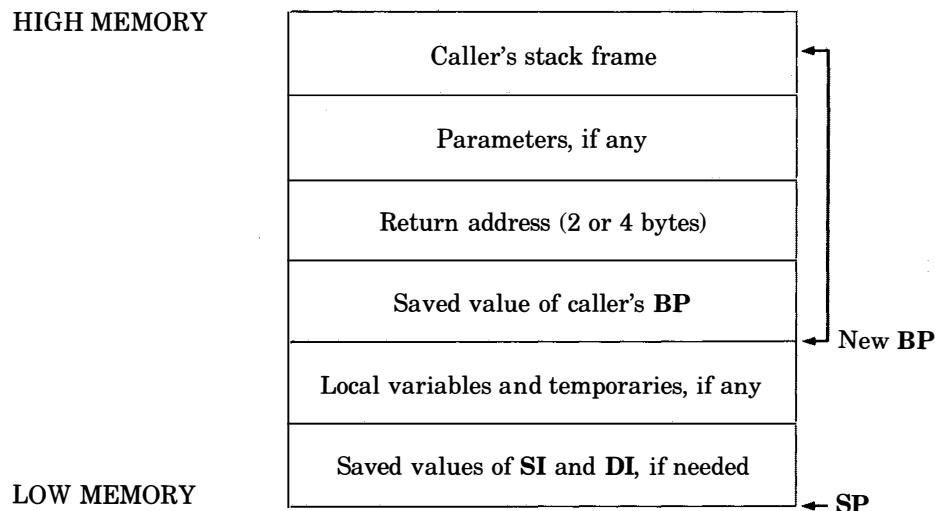


Figure F.2 Version 3.0 Stack Frame Setup

The differences in these two setups are reflected in the entry and exit code sequences for subroutine calls and in the locations for local variables and parameters.

In versions 3.0 and later, 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.

Versions 3.0 and later use 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 or the **check_stack** pragma), 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 versions 3.0 and later also differ from previous versions. In versions 3.0 and later, 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 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.

The following is the entry/exit sequence for 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

```

The following is the entry/exit sequence for 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

```

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 4.0 (or 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.

F.3.3.3 Global-Variable Naming Conventions

In versions 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 versions 3.0 and later, 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 versions 3.0 and later means that the name of any assembly routine called from a Version 4.0 (or 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 versions 3.0 and later, they are. However, when invoking the linker directly (through the **LINK** command) with a Version 4.0 (or 3.0) program, case is ignored by default. You can take advantage of this behavior when linking programs from versions 2.03 and earlier. By contrast, the Version 4.0 (or 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.

F.3.3.4 Segment Usage and Naming

The structure of a Version 4.0 (or 3.0) program in memory differs slightly from the Version 2.03 structure. Versions 3.0 and later have 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 versions 3.0 and later, use a different layout from Version 3.0. The Version 2.03 D and L models use essentially the same layout as do Version 4.0 (or 3.0) programs, with the following two exceptions:

1. There is no equivalent to the Version 4.0 (or 3.0) segment for far data.
2. In versions 2.03 and earlier, **SS** always points to the stack segment base instead of **DS**. This is similar to specifying the letter **u** with the memory-model (**/A**) option in versions 3.0 and later.

The Version 2.03 S- and P-model layouts and the Version 4.0 (or 3.0) layout are shown in figures F.3 and F.4, respectively.

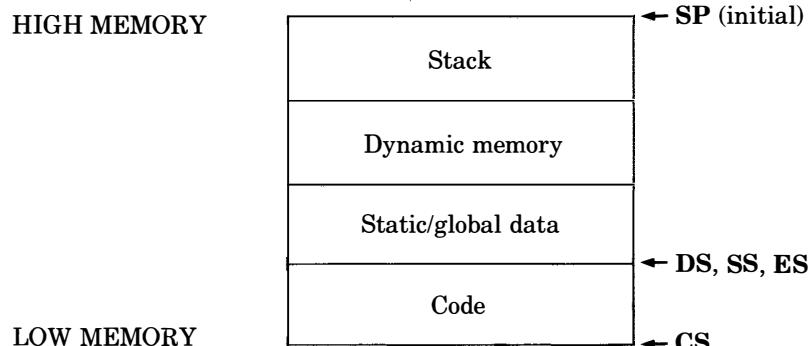


Figure F.3 Version 2.03 Layout for the S and P Models

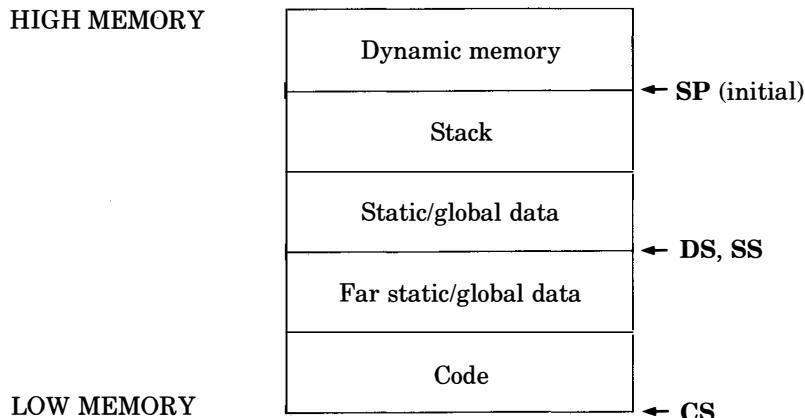


Figure F.4 Layouts for the 3.0 and 4.0 Versions

There are two main differences between the layouts shown above. First, in Version 2.03, the stack resides above dynamic memory. In versions 3.0 and later, 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 versions 3.0 and later, **ES** does not necessarily contain the same value as **DS**. Versions 3.0 and later support 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 versions 3.0 and later, 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 versions 3.0 and later, all code segments have class **CODE**. In small model (Version 3.0 and Version 4.0) and compact model (Version 4.0 only), the code segments are all named **_TEXT** by default; in medium and large models, each compiland forms a segment named *modulename_TEXT*.
- In Version 2.03 S-model programs, the code segment is placed in a group named **PGROUP**. In small-model (Version 3.0 and Version 4.0) and compact-model programs (Version 4.0 only), 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 10.2.1, “Segment Model,” in Chapter 10, “Interfaces with Other Languages,” for more information on what kinds of data items are stored in each of the segments.

Appendix G

Writing Portable Programs

G.1	Introduction	347
G.2	Program Portability	348
G.3	Machine Hardware	348
G.3.1	Byte Length	348
G.3.2	Word Length	349
G.3.3	Storage Alignment	349
G.3.4	Byte Order in a Word	350
G.3.5	Bit Fields	351
G.3.6	Pointers	352
G.3.7	Address Space	353
G.3.8	Character Set	353
G.4	Compiler Differences	354
G.4.1	Signed/Unsigned char and Sign Extension	354
G.4.2	Shift Operations	354
G.4.3	Identifier Length	355
G.4.4	Register Variables	355
G.4.5	Type Conversion	356
G.4.6	Functions with a Variable Number of Arguments	357
G.4.7	Side Effects and Evaluation Order	357
G.5	Environment Differences	358
G.6	Portability of Data	359
G.7	Byte-Ordering Summary	360

1

2

3

G.1 Introduction

The standard definition of the C programming language leaves many details to be decided in specific implementations of the language. These unspecified features of the language detract from its portability and must be studied when attempting to write portable C code.

Most of the issues affecting C portability arise from differences in either target-machine hardware or compilers. C was designed to compile efficient code for the target machine (initially a PDP-11), so many of the language features not precisely defined are those that reflect a particular machine's hardware characteristics.

This appendix highlights the various aspects of C that may not be portable across different machines and compilers. It also briefly discusses the portability of a C program in terms of its environment. The environment is determined by the system calls and library routines a program uses during execution, file path names it requires, and other items not guaranteed to be constant across different systems.

The C language has been implemented on many different computers with widely different hardware characteristics, from small 8-bit microprocessors to large mainframes. This appendix is concerned with the portability of C code in the MS-DOS and XENIX programming environments. This is a more restricted problem to consider, since all MS-DOS and XENIX operating systems to date run on hardware with the following basic characteristics:

- ASCII character set
- 8-bit bytes
- 2-byte or 4-byte integers
- Two's complement arithmetic

These features are not formally defined for the language and may not be found in all implementations of C. However, the remainder of this appendix is devoted to those systems where these basic assumptions hold.

The C language definition contains no specification of how input and output are performed. These specifications are left to system calls and library routines on individual systems. Within XENIX systems there are system calls and library routines that can be considered portable. This version of the Microsoft C Compiler includes system calls and library routines that

can be considered portable across XENIX and MS-DOS systems. The run-time library for the Microsoft C Compiler for MS-DOS is composed primarily of XENIX-compatible routines. By restricting the use of XENIX routines to those included in the MS-DOS library, the XENIX programmer can develop MS-DOS programs in the XENIX environment; C programs written on MS-DOS are easily portable to XENIX.

This appendix is not intended as a C-language primer. It is assumed that the reader is familiar with C, and with the basic architecture of common microprocessors.

G.2 Program Portability

A program is portable if it can be compiled and run successfully on different machines without alteration. There are many ways to write portable programs. The first is to avoid using inherently nonportable language features. The second is to isolate any nonportable interactions with the environment, such as I/O to nonstandard devices. For example, programs should avoid hard-coded path names unless a path name is common to all systems.

Files required at compile time (such as include files) may also introduce nonportability if the path names used are not the same on all machines. In some cases, include files containing machine-specific definitions can be used to make the source code itself portable.

G.3 Machine Hardware

Differences in the hardware of the various target machines and differences in the corresponding C compilers cause the greatest number of portability problems. This section lists problems commonly encountered.

G.3.1 Byte Length

By definition, the **char** data type in C must be large enough to hold as positive integers all members of a machine's character set. For the machines described in this appendix, the **char** size is exactly an 8-bit byte.

G.3.2 Word Length

The size of the basic data types for a given implementation are not formally defined in the C language. Thus they often follow the most natural size for the underlying machine. It is safe to assume that **short** is no longer than **long**. Beyond that, no assumptions are portable. For example, on some machines **short** is the same length as **int**, whereas on others **long** is the same length as **int**.

Programs that need to know the size of a particular data type should avoid hard-coded constants where possible. Such information can usually be written in a fairly portable way. For example, the maximum positive integer (on a two's complement machine) can be obtained with the following:

```
#define MAXPOS ((int)((unsigned)-1) >> 1))
```

This is preferable to the following code:

```
#ifdef PDP11
#define MAXPOS 32767
#else
.
.
.
#endif
```

To find the number of bytes in an **int**, use **sizeof(int)** rather than 2, 4, or some other nonportable constant.

G.3.3 Storage Alignment

The C language defines no particular layout for storage of data items relative to each other. The layout for storage of elements of structures, or unions within the structure or union, is also left undefined by the language.

Some processors require that data types longer than one byte be aligned on even byte address boundaries. Others, such as the 8086/8088, have no such hardware restriction. However, even with these machines, most compilers generate code that aligns words, structures, arrays, and long words on even addresses or on even long-word addresses. Therefore, the following code sequence may give different results, depending on specific alignment requirements on different machines:

```

struct stag {
    char c;
    int i;
};
printf("%d\n", sizeof(struct stag));

```

The principal implications of this variation in data storage are that data accessed as nonprimitive data types are not portable, and code that makes use of knowledge of the layout on a particular machine is not portable.

Therefore, unions containing structures are nonportable if the union is used to access the same data in different ways. Unions are only likely to be portable if they are used only to store different data in the same space at different times. For example, if the following union were used to obtain 4 bytes from a long word, the code would not be portable:

```

union {
    char c[4];
    long lw;
} u;

```

The **sizeof** operator should always be used when reading and writing structures, as follows:

```

struct s_tag st;
.
.
.
write(fd, &st, sizeof(st));

```

Using the **sizeof** operator ensures portability of the source code, but does not produce a portable data file. Portability of data is discussed in a later section.

G.3.4 Byte Order in a Word

The variation in byte order in a word affects the portability of data more than the portability of source code. However, any program that makes use of knowledge of the internal byte order in a word is not portable. For example, on some XENIX systems there is an include file *misc.h* that contains the following structure declaration:

```

/*
 * structure to access an
 * integer in bytes
 */
struct {
    char lobyte;
    char hibyte;
};
```

With certain less restrictive compilers, this declaration could be used to access the high- and low-order bytes of an integer separately and in a completely nonportable way. The correct way to do this is to use mask and shift operations to extract the required byte.

```
#define LOBYTE(i) (i & 0xff)
#define HIBYTE(i) ((i >> 8) & 0xff)
```

These definitions provide a portable way to extract the least-significant and the next-least-significant bytes of an integer. Since the **int** type can be either 2 or 4 bytes, depending on the machine, even these definitions do not provide a completely portable way to access the bytes of an **int**.

One result of the byte-ordering problem is that the following code sequence will not always perform as intended:

```
int c = 0;
read(fd, &c, 1);
```

On machines where the low-order byte is stored first, the value of **c** is the byte value read. On other machines, the byte is read into some byte other than the low-order one, so the value of **c** is different.

G.3.5 Bit Fields

Bit fields are not implemented in all C compilers. The Microsoft C Compiler implements bit fields and allows them to have any length up to the size of a **long**. However, in many implementations no bit field may be larger than an **int**, and no bit field can overlap an **int** boundary. If necessary, the compiler will leave gaps and move to the next **int** boundary. To ensure portability no individual field should exceed 16 bits.

The C language makes no guarantees about whether bit fields are assigned left to right or right to left. Therefore, while bit fields may be useful for storing flags and other small data items, their use in unions to dissect bits from other data is definitely nonportable.

G.3.6 Pointers

The C language is fairly generous in allowing manipulation of pointers, to the extent that most compilers will not object to nonportable pointer operations. A common nonportable use of pointers is the use of casts to assign one pointer to another pointer of a different data type. This practice almost always makes some assumption about the internal byte ordering and layout of the data type, and is therefore nonportable. In the following code, the byte order in the given array is not portable:

```
char c[4];
long *lp;

lp = (long *)&c[0];
*lp = 0x12345678L;
```

Code like this is very rarely necessary or valid. It is acceptable, however, when using the **malloc** function to allocate space for variables that do not have **char** type. The routine is declared as type **char ***, and the return value is cast to the type to be stored in the allocated memory. If this type is not **char ***, then a compiler may issue a warning concerning illegal type conversion. In addition, the **malloc** function is designed always to return a starting address suitable for storing all types of data. A compiler may not know this, so it may give an additional warning about possible data alignment problems. In the following example, **malloc** is used to obtain memory for an array of 50 integers:

```
extern char *malloc( );
int *ip;

ip = (int *)malloc(50);
```

This example will elicit a warning message from some compilers.

The *Microsoft C Compiler Language Reference* states that a pointer can be assigned (or cast) to an integer large enough to hold it. Note that the size of the **int** type depends on the given machine and implementation. This type is a **long** on some machines, and a **short** on others. The size may also be modified by **near** and **far** declarations. In general, do not assume that the following statement will always be true:

```
sizeof(char *) == sizeof(int)
```

For example, the following construction is nonportable, assuming that the function identifier `func` is not previously declared:

```
int p;  
p = (char *) func( );
```

This example assumes that a `char` pointer has the same length as an `int`.

In most implementations, the null pointer value `NULL` is defined to be the `int` value 0. The length of the zero value can lead to problems for functions that expect pointer arguments longer than an `int`. For portable code, always use the following form to pass a `NULL` value of the correct size:

```
func( (char *)NULL );
```

G.3.7 Address Space

The address space available to a program varies considerably from system to system. Some small processors allow only 64K for program text and data combined. Others allow up to 64K of data and 64K of program text. Larger machines may allow considerably more text and possibly more data as well.

Large programs, or programs that require large data areas, may have portability problems on small machines.

G.3.8 Character Set

The C language does not require the use of the ASCII character set. In fact, the only character-set requirements are that all characters must fit in the `char` data type, and all characters must have positive values.

In the ASCII character set, all characters have values between 0 and 127 and thus can be represented in 7 bits. On an 8-bits-per-byte machine they are all positive, regardless of whether `char` is treated as signed or unsigned.

A set of character-classification macros is included as part of the run-time library for the Microsoft C Compiler. These macros should be used for most tests on character quantities. The macros are defined in the include file **CTYPE.H**, and described in the *Microsoft C Compiler Run-Time Library Reference*. They appear on the pages headed **isalnum-isascii** and **iscntrl-isxdigit**.

The character-classification macros provide insulation from the internal structure of the character set. In addition, the names of the macros are often more meaningful than the equivalent line of code. Compare the following two lines:

```
if (isupper (c))  
if ((c >= 'A') && (c <= 'Z'))
```

With some of the other macros, such as **isxdigit** to test for a hexadecimal digit, the advantage is even greater. Also, the internal implementation of the macros makes them more efficient than an explicit test with an **if** statement.

G.4 Compiler Differences

There are a number of C compilers running under various operating systems. The main areas of differences between compilers are outlined in this section.

G.4.1 Signed/Unsigned char and Sign Extension

The current state of the signed versus unsigned **char** problem is best described as unsatisfactory. The sign-extension problem is a serious barrier to writing portable C, and the best solution at present is to write defensive code that does not rely on particular implementation features.

G.4.2 Shift Operations

The left-shift operator (**<<**) shifts its operand a number of bits left, filling vacated bits with zeros. This is called a logical shift. The right-shift operator (**>>**) when applied to an unsigned quantity, performs a logical-shift operation, but when it is applied to a signed quantity, the vacated bits may be filled with zeros (logical shift) or with sign bits (arithmetic shift). The decision is implementation dependent, and code that uses knowledge of a particular implementation is nonportable.

With compilers that use arithmetic right shift, it is necessary to shift and mask the appropriate number of high-order bits to avoid sign extension, as follows:

```
char c;  
c = (c >> 3) & 0x1f;
```

~ You can also avoid sign extension by using the divide operator (/), as follows:

```
char c;  
c = c / 8;
```

G.4.3 Identifier Length

The use of long symbols and identifier names will cause portability problems with some compilers. To avoid these problems, a program should keep the following symbols as short as possible:

- C preprocessor symbols
- C local symbols
- C external symbols

~ Some loaders also place restrictions on the number of unique characters in C external symbols. Symbols unique in the first six characters are unique to most C-language processors.

In some C implementations, uppercase and lowercase letters are not distinct in identifiers.

G.4.4 Register Variables

The number and type of register variables in a function depend on the machine hardware and the compiler. Excess and invalid register declarations are treated as nonregister declarations and should not cause a portability problem. On an 8086 or 8088 processor, up to two register declarations are significant, and they must be applied to types of size **int** or smaller.

~ Since the compiler ignores excess variables of register type, the most important register type variables should be declared first. Therefore, if any are ignored, they will be the least important ones.

G.4.5 Type Conversion

The C language has some rules for implicit type conversion; it also allows explicit type conversions by type casting. The most common portability problem in implicit type conversion is unexpected sign extension. This is a potential problem whenever something of type **char** is compared with an **int**.

The following example will never evaluate true on a machine that sign-extends **char** types but treats hexadecimal numbers as unsigned:

```
char c;  
  
if (c == 0x80) {  
    .  
    .  
    .  
}
```

The following construction is also nonportable:

```
char c;  
unsigned int u;  
  
if (u == (unsigned)c) {  
    .  
    .  
}
```

Two problems can arise in the preceding example:

1. The **char** type may be considered either signed or unsigned, depending on the implementation.
2. For implementations that consider the **char** type to be signed, two different methods of carrying out the conversion are possible: the **char** value may be sign extended to **int** type first, then converted to **unsigned** type; or the **char** type may be converted to an unsigned type of the same size, then zero extended to **int** length.

The only safe comparison between **char** type and an **int** is the following:

```

int c;

if (c == 'x') {
    .
    .
    .
}

```

This comparison is reliable because C guarantees all character constants to be positive.

Type conversion also occurs when arguments are passed to functions. Types **char** and **short** become **int**. Extending the **char** type can produce unexpected results. For example, the following program gives -128 on some machines:

```

char c = 128;
printf ("%d\n", c);

```

The unexpected negative value is produced because **c** is converted to **int** when it is passed to the **printf** function. The function itself has no knowledge of the original type of the argument and is expecting an **int**. The correct way to handle this situation is to code defensively and allow for the possibility of sign extension, as in the following example:

```

char c = 128;
printf ("%d\n", c & 0xff);

```

G.4.6 Functions with a Variable Number of Arguments

Functions with a variable number of arguments present a particular portability problem if the type of the arguments is also variable. In such cases the code is dependent on the size of various data types. For portability, these cases should be avoided.

G.4.7 Side Effects and Evaluation Order

The C language makes few guarantees about the order of evaluation of operands in an expression or arguments to a function call. Therefore, the following statement is almost never portable:

```
func (i++, i++);
```

Even the following statement is unwise if **func** is ever likely to be replaced by a macro, since the macro may use *i* more than once:

```
func(i++);
```

Certain XENIX-compatible macros commonly appear in user programs; some of these use their argument only once, and therefore can safely be called with a side-effect argument. To determine whether a macro handles side effects correctly, examine the code for that macro to see whether or not the argument is evaluated more than once.

Operands to the following operators are guaranteed to be evaluated left to right:

```
,      &&      ||      ?      :
```

Note that the comma operator here is a separator for two C statements. A list of items separated by commas in a declaration list is not guaranteed to be processed left to right. Therefore, the following declaration on an 8086 or 8088 processor, where only two register variables may be declared, could give any two of the four variables **register** type, depending on the compiler:

```
register int a, b, c, d;
```

To give register storage to the most important variables, use separate declaration statements and declare the most important variables first. The order of processing of individual declaration statements is guaranteed to be sequential in the following statements:

```
register int a;  
register int b;  
register int c;  
register int d;
```

G.5 Environment Differences

Most programs make system calls and use library routines for various services. This section indicates some of those routines that are not always portable and those that particularly aid portability.

System calls specific to an operating system are not portable if they are not present on all other operating-system implementations of C. Most of the system calls defined in the Microsoft MS-DOS run-time library are compatible with XENIX system calls and are thus portable to a XENIX environment.

Any program that contains hard-coded path names to files or directories, or that contains user identifier numbers, log-in names, terminal lines or other system-dependent parameters, is nonportable. These types of constants should be in header files, passed as command-line arguments, or obtained from the environment.

Note that the members of the **printf** family of functions, including **fprintf**, **fscanf**, **printf**, **sprintf**, **scanf**, **vprintf**, **vprintf**, **vsprintf**, and **sscanf**, have evolved in several ways, and some features are not completely portable. Some of the format-conversion characters have changed their meanings, in particular those relating to uppercase and lowercase in the output of hexadecimal numbers and the specification of **long** integers on 16-bit word machines. The Microsoft C specifications for these routines are given in the *Microsoft C Compiler Run-Time Library Reference*.

G.6 Portability of Data

- Data files are almost always nonportable across different central-processing-unit (CPU) architectures. As mentioned above, structures, unions, and arrays have varying internal layout and padding requirements on different machines. In addition, byte ordering within words and actual word length may differ.

The only way to achieve data-file portability is to write and read data files as one-dimensional character arrays. This procedure prevents alignment and padding problems if the data is written and read as characters, and interpreted that way. Thus ASCII text files can usually be moved between different machine types without significant problems.

G.7 Byte-Ordering Summary

Tables G.1 and G.2 summarize byte ordering for **short** and **long** types, respectively. The following conventions are used in these tables:

1. The lowest physically-addressed byte of the data item is **a0**; **a1** has the byte address **a0 + 1**, and so on.
2. The least-significant byte of the data item is **b0**; **b1** is the next-least-significant, and so on.

Since byte ordering is machine specific, any program that actually makes use of the following information is guaranteed to be nonportable:

Table G.1
Byte Ordering for Short Types

CPU	Byte Order
	a0 a1
8086	b0 b1
80286	b0 b1
PDP-11	b0 b1
VAX-11	b0 b1
M68000	b1 b0
Z8000	b1 b0

Table G.2
Byte Ordering for Long Types

CPU	Byte Order
	a0 a1 a2 a3
8086	b0 b1 b2 b3
80286	b0 b1 b2 b3
PDP-11	b2 b3 b0 b1
VAX-11	b0 b1 b2 b3
M68000	b3 b2 b1 b0
Z8000	b3 b2 b1 b0

)

)

)

Appendix H

Error Messages

H.1	Introduction	365
H.2	Run-Time Error Messages	365
H.2.1	Run-Time-Library Error Messages	366
H.2.2	Floating-Point Exceptions	368
H.2.3	Run-Time Limits	370
H.3	Compiler Error Messages	371
H.3.1	Warning Error Messages	373
H.3.2	Fatal Error Messages	382
H.3.3	Compilation Error Messages	387
H.3.4	Command-Line Error Messages	404
H.3.5	Compiler Limits	409
H.4	LINK Error Messages	410
H.5	Library-Manager Error Messages	417
H.6	MAKE Error Messages	421
H.7	EXEPACK Error Messages	423
H.8	EXEMOD Error Messages	424
H.9	SETENV Error Messages	425

()

()

()

H.1 Introduction

This appendix lists error messages you may encounter as you develop a program, and gives a brief description of actions you can take to correct the errors. The first section lists run-time errors. Run-time errors are errors that may be encountered when running an executable file developed with the C compiler.

The remaining sections describe errors generated by the following programs:

- The Microsoft C Compiler
- The Microsoft Overlay Linker (**LINK**)
- The Microsoft Library Manager (**LIB**)
- The Microsoft Program Maintenance Utility (**MAKE**)
- The Microsoft EXE File Compression Utility (**EXEPACK**)
- The Microsoft EXE File Header Utility (**EXEMOD**)
- The Microsoft Environment Manager (**SETENV**)

H.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 H.2.2.
3. Error messages generated by program calls 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 Compiler 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 Compiler Run-Time Library Reference* for a description of the math routines and corresponding error messages.

H.2.1 Run-Time-Library Error Messages

The following messages may be generated at run time when your program has serious errors:

Number	Run-Time-Library Error Message
2000	<p>Stack overflow</p> <p>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 code 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.</p>
2001	<p>Null pointer assignment</p> <p>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. Note 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.</p> <p>This error does not cause your program to terminate; the error message is printed following the normal termination of the program.</p> <p>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.</p>

Number	Run-Time-Library Error Message
2002	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:
	<ol style="list-style-type: none"><li data-bbox="608 385 1311 732">1. A format string for one of the routines in the printf or scanf families 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.<li data-bbox="608 748 1327 897">2. The xLIBFP.LIB or xLIBFA.LIB library (where <i>x</i> is S, M, C, L, or H, depending on the memory model) was specified after xLIBC.LIB in the linking stage. You must relink the program with the correct library specification.
2003	Integer divide by 0
	An attempt was made to divide an integer by 0, giving an undefined result.
2004	DOS 2.0 or later required
	The C compiler cannot run on versions of MS-DOS prior to 2.0.
2005	Not enough memory on exec
2006	Bad format on exec
2007	Bad environment on exec
	Errors 2005 through 2007 occur when a child process spawned by one of the exec library routines fails, and MS-DOS is unable to return control to the parent process.

Number	Run-Time-Library Error Message
2008	Not enough space for arguments
	See explanation under error 2009.
2009	Not enough space for environment
	Errors 2008 and 2009 both occur at start-up if there is enough memory to load the program, but not enough room for the argv and/or envp vectors. To avoid this problem, you can rewrite the _setargv or _setenvp routines (see Section 5.2.2, "Suppressing Command-Line Processing," for more information).
2011	Unknown error
	Note the circumstances of the failure and notify Microsoft Corporation using the Software Problem Report at the back of this manual.
2100	Floating point error: <i>message</i>
	This error message is generated whenever a floating-point exception occurs. The <i>message</i> describes the particular floating-point exception that occurred. The messages that can appear with this error are listed and described in the next section.

H.2.2 Floating-Point Exceptions

The error messages listed below correspond to exceptions generated by the 8087/80287 hardware. These messages appear with error 2100, "Floating point error," as described in the previous section. 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
Inexact	Exception masked

For information on how to change the floating-point control word, see the reference pages for `_control87` in the *Microsoft C Compiler Run-Time Library Reference Manual*.

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 NaNs 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 Compiler 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 invalid 8087/80287 instruction or an 8087/80287 instruction not supported by the emulator.

H.2.3 Run-Time Limits

Table H.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 H.1
Program Limits at Run Time

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 files available for the program to open.

H.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 Chapter 3, “Compiling.” The list of warning messages in Section H.3.1 includes 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 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. These messages should not appear no matter what your source program contains. If they do, please report the condition to Microsoft, using the Software Problem Report at the back of this manual. The following messages indicate internal compiler errors:

warning 0: UNKNOWN WARNING
Contact Microsoft Technical Support

fatal error 0: UNKNOWN FATAL ERROR
Contact Microsoft Technical Support

fatal error 1: Internal Compiler Error
(compiler file '*filename*', line *linenumber*)
Contact Microsoft Technical Support

error 0: UNKNOWN ERROR
Contact Microsoft Technical Support

error 124: CODE GENERATION ERROR
Contact Microsoft Technical Support

command line error 0: UNKNOWN COMMAND LINE ERROR
Contact Microsoft Technical Support

These messages are described in more detail in their respective sections.

Error messages in the warning-, fatal-, and compilation-error categories have the same basic format, as follows:

filename (linenumber): messagetype errornumber: message

In this format, *filename* is the name of the source file being compiled. The *linenumber* identifies the line of the file containing the error, and *messagetype* is one of the following: warning, fatal error, or error (for compilation errors). The *errornumber* is the number of the error and *message* is a description of the error or warning.

Command-line error messages have a similar format, but they do not contain references to file names or line numbers; their *messagetype* is Command line error.

The messages for each category are listed below in numerical order, with a brief explanation of each error. To look up an error message, first determine the message category, then find the error number.

Section H.3.5, "Compiler Limits," summarizes limits imposed by the Microsoft C Compiler (for example, the maximum size of a macro definition).

H.3.1 Warning Error Messages

The messages listed in this section indicate potential problems but do not hinder compilation and linking. The number in parentheses at the end of each error-message description gives the minimum warning level that must be set for the message to appear.

Number	Warning Error Message
0	UNKNOWN WARNING Contact Microsoft Technical Support. An unknown error condition has been detected by the compiler. Please report this condition to Microsoft, using the Software Problem Report form at the back of this manual.
1	macro ' <i>identifier</i> ' requires parameters The given <i>identifier</i> was defined as a macro taking one or more arguments, but the <i>identifier</i> is used in the program without arguments. (1)
2	too many actual parameters for macro ' <i>identifier</i> ' 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. (1)

Number	Warning Error Message
3	'not enough actual parameters for macro ' <i>identifier</i> '
	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. (1)
4	missing close parenthesis after 'defined'
	The closing parenthesis is missing from an #if defined phrase. (1)
5	' <i>identifier</i> ' : redefinition
	The given <i>identifier</i> is redefined. (1)
6	#undef expected an identifier
	The name of the identifier whose definition is to be removed must be given with the #undef directive.
9	string too big, trailing chars truncated
	A string exceeds the compiler limit on string size. To correct this problem, you must break the string into two or more strings. (1)
11	' <i>identifier</i> ' truncated to ' <i>identifier</i> '
	Only the first 31 characters of an identifier are significant. (1)
13	constant too big
	Information is lost because a constant value is too large to be represented in the type to which it is assigned. (1)
14	' <i>identifier</i> ' : bitfield type must be unsigned
	Bit fields must be declared as unsigned integral types. A conversion has been supplied. (1)
15	' <i>identifier</i> ' : bitfield type must be integral
	Bit fields must be declared as unsigned integral types. A conversion has been supplied. (1)

Number	Warning Error Message
16	' <i>identifier</i> ' : no function return type [2]
	Function <i>identifier</i> has not yet been declared or defined, so no return type is known. The default return type (int) will be assumed. (2)
17	cast of int expression to far pointer
	A far pointer represents a full segmented address. On an 8086/8088 processor, casting an int value to a far pointer may produce an address with a meaningless segment value. (1)
20	too many actual parameters
	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. (1)
21	too few actual parameters
	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. (1)
22	pointer mismatch: parameter <i>n</i>
	The given parameter has a different pointer type than is specified in the argument-type list or the function definition. (1)
24	different types : parameter <i>n</i>
	The type of the given parameter in a function call does not agree with the argument-type list or the function definition. (1)
25	function declaration specified variable args
	The argument-type list in a function declaration ends with a comma, or a comma followed by ellipsis dots (...), indicating that the function can take a variable number of arguments, but no formal parameters for the function are declared. (1)

Number	Warning Error Message
26	function was declared with formal arguments The function was declared to take arguments, but the function definition does not declare formal parameters. (1)
27	function was declared without formal argument list 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. (1)
28	parameter <i>n</i> declaration different The type of the given parameter does not agree with the corresponding type in the argument-type list or with the corresponding formal parameter. (1)
29	declared parameter list different from definition The argument-type list given in a function declaration does not agree with the types of the formal parameters given in the function definition. (1)
30	first parameter list is longer than the second A function is declared more than once and the argument-type lists in the declarations differ. (1)
31	second parameter list is longer than the first A function is declared more than once, and the argument-type lists in the declarations differ. (1)
32	unnamed struct/union as parameter 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. (1)

Number	Warning Error Message
33	function must return a value
	A function is expected to return a value unless it is declared as void . (2)
34	sizeof returns 0
	The sizeof operator is applied to an operand that yields a size of zero. (1)
35	no return value
	A function declared to return a value does not do so. (2)
36	unexpected formal parameter list
	A formal parameter list is given in a function declaration and is ignored. (1)
37	' <i>identifier</i> ' : formal parameters ignored
	Formal parameters appeared in a function declaration, as in the following example:
	<code>extern int *f(a,b,c);</code>
	The formal parameters are ignored. (1)
38	' <i>identifier</i> ' : formal parameter has bad storage class
	Formal parameters must have auto or register storage class. (1)
39	' <i>identifier</i> ' : function used as an argument
	A formal parameter to a function is declared to be a function, which is illegal. The formal parameter is converted to a function pointer. (1)
40	near/far/huge on ' <i>identifier</i> ' ignored
	The near or far keyword has no effect in the declaration of the given <i>identifier</i> and is ignored. (1)

Number	Warning Error Message
41	formal parameter ' <i>identifier</i> ' is redefined The given formal parameter is redefined in the function body, making the corresponding actual argument unavailable in the function. (1)
42	' <i>identifier</i> ' : has bad storage class 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. (1)
43	' <i>identifier</i> ' : void type changed to int Only functions may be declared to have void type. (1)
44	huge on ' <i>identifier</i> ' ignored, must be an array The huge keyword can only be used in array declarations. (1)
45	' <i>identifier</i> ' : array bounds overflow Too many initializers are present for the given array. The excess initializers are ignored. (1)
46	'&' on function/array, ignored You cannot apply the address-of operator to a function or array identifier. (1)
47	' <i>operator</i> ' : different levels of indirection An expression involving the specified operator has inconsistent levels of indirection, as in the following examples: (1) char **p; char *q; . . . p = q;

Number	Warning Error Message
48	array's declared subscripts different An array is declared twice with differing sizes. The larger size is used. (1)
49	'operator' : indirection to different types The indirection operator (*) is used in an expression to access values of different types. (1)
51	data conversion Two data items in an expression had different types, causing the type of one item to be converted. (2)
52	different enum types Two different enum types are used in an expression. (1)
53	at least one void operand An expression with type void is used as an operand. (1)
54	'operator' : illegal with enums You may not use the given operator with an enum value. The enum value is converted to int type. (1)
56	overflow in constant arithmetic The result of an operation exceeds 0xFFFFFFFF. (1)
57	overflow in constant multiplication The result of an operation exceeds 0xFFFFFFFF. (1)
58	address of frame variable taken, DS != SS Program was compiled with the default data segment (DS) not equal to the stack segment (SS) and user attempted to point to a frame variable with a near pointer. (1)
59	conversion lost segment 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. (1)

Number	Warning Error Message
60	conversion of a long address to a short address 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. (1)
61	long/short mismatch in arguments: conversion supplied Actual and formal arguments of a function differ in base type; actual argument will be converted to type of formal parameter. (1)
62	near/far mismatch in arguments: conversion supplied Actual and formal arguments of a function differ in pointer size; actual argument will be converted to size of formal parameter. (1)
63	function ' <i>identifier</i> ' too large for post-optimizer The named function was not optimized because insufficient space was available. To correct this problem, reduce the size of the function by dividing it into two or more smaller functions. (0)
64	procedure too large-, skipping <i>description</i> optimization and continuing 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 dividing it into two or more smaller functions. The <i>description</i> in this message may be any of the following: (0) loop inversion branch sequence cross jump
65	recoverable heap overflow in post optimizer - some optimizations may be missed

Number	Warning Error Message
	Some optimizations are skipped because insufficient space is available for optimization. To correct this problem, reduce the size of the function by dividing it into two or more smaller functions. (0)
66	local symbol table overflow - some local symbols may be missing in listings
	Listing generator ran out of heap space for local variables, so source listing may not contain symbol-table information for all local variables.
67	unexpected characters following ' <i>identifier</i> ' directive - newline expected
	There are extra characters following a preprocessor directive, such as the following:
	#endif NO_EXT_KEYS
	This is accepted in Version 3.0, but not in 4.0; Version 4.0 requires comment delimiters, such as the following:
	#endif /* NO_EXT_KEYS */
68	unknown pragma
	The pragma used is unrecognized and ignored by the compiler.
69	conversion of near pointer to long integer
	A near pointer is being converted to a long integer, which involves first extending the high-order word with the current data-segment value, <i>not</i> 0, as in Version 3.0.
72	missing semi-colon
	Missing a semicolon following last member of structure or union.

H.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. EX ON

Number	Fatal Error Message
0	UNKNOWN FATAL ERROR Contact Microsoft Technical Support An unknown error condition has been detected by the compiler. Please report this condition to Microsoft, using the Software Problem Report at the back of this manual.
1	Internal Compiler Error (compiler file ' <i>filename</i> ', line <i>linenumber</i>) Contact Microsoft Technical Support Compiler has detected internal inconsistency; please report condition to Microsoft using the Software Problem Report at the back of this manual. Please include the <i>filename</i> and <i>linenumber</i> in this report; note that <i>filename</i> refers to an internal compiler file, <i>not</i> your source file.
2	out of heap space The compiler has run out of dynamic memory space. This usually means that your program has many symbols and/or complex expressions. To correct the problem, divide the file into several smaller source files, or break expressions into smaller subexpressions.
3	error count exceeds <i>n</i> ; stopping compilation Errors in the program are too numerous or too severe to allow recovery, and the compiler must terminate.
4	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. This message can also occur when a comment does not have a closing delimiter (* /), or when an #if directive occurs without a corresponding closing #endif directive.

- 6 write error on compiler intermediate file
The compiler is unable to create the intermediate files used in the compilation process. The exact reason is unknown, although the following are two common causes:
1. Too few files in the
files=number
line (the compiler requires *number* to be at least 15)
 2. Not enough space on a device containing a compiler intermediate file
- 7 unrecognized flag '*string*' in '*option*'
The given *string* in the command line *option* is not a valid option.
- 9 compiler limit
possibly a recursively defined macro
The expansion of a macro exceeds the available space.
Check to see if the macro is recursively defined, or if the expanded text is too large.
- 10 compiler limit
macro expansion too big
The expansion of a macro exceeds the available space.
- 11 recursively defined macro '*identifier*'
The given identifier is defined recursively.
- 12 bad parenthesis nesting - missing '*character*'
The parentheses in a preprocessor directive are not matched; *character* is either a left or right parenthesis.
- 13 cannot open '*filename*'
The given file cannot be opened.
- 14 too many include files
Nesting of #**include** directives exceeds the limit of ten levels.

- 15 cannot find '*filename*'
The given file does not exist or cannot be found. Make sure
your environment settings are valid and that you have given
the correct path name for the file.
- 16 #ifndef[n] def expected an identifier
You must specify an identifier with the **#ifdef** and **#ifndef**
directives.
- 17 constant term expected
The expression in an **#if** directive must evaluate to a
constant.
- 18 unexpected '#elif'
The **#elif** directive is legal only when it appears within an
#if, **#ifdef**, or **#ifndef** directive.
- 19 unexpected '#else'
The **#else** directive is legal only when it appears within an
#if, **#ifdef**, or **#ifndef** directive.
- 20 unexpected '#endif'
An **#endif** directive appears without a matching **#if**,
#ifdef, or **#ifndef** directive.
- 21 bad preprocessor command '*string*'
The characters following the number sign (#) do not form a
valid preprocessor directive.
- 22 expected '#endif'
An **#if**, **#ifdef**, or **#ifndef** directive was not terminated
with an **#endif** directive.
- 26 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.

27	DGROUP data allocation exceeds 64K Allocation of variables to the default segment exceeds 64K; in compact, medium, or huge models, use the /GT option to move items into separate segments.
32	cannot open listing file ' <i>filename</i> '
33	cannot open assembly-language output file ' <i>filename</i> '
34	cannot open source file ' <i>filename</i> ' For error messages 32, 33, 34, 36, and 37, one of the following statements about the file name or path name given (<i>filename</i>) is true: <ol style="list-style-type: none">1. The given name is not valid.2. The file with the given name cannot be opened for lack of space.3. A read-only file with the given name already exists.
35	expression too complex, please simplify The compiler cannot generate code for a complex expression; break the expression into simpler subexpressions and recompile.
36	cannot open source-listing file ' <i>filename</i> ' See note under error message 34.
37	cannot open object file ' <i>filename</i> ' See note under error message 34.
39	unrecoverable heap overflow in P3 Postoptimizer has overflowed heap and cannot continue; try recompiling with /Od option (see Section 3.12, “Optimizing”), or breaking up function containing the line causing the error.
40	Unexpected EOF in source file ' <i>filename</i> ' Compiler detected unexpected end-of-file while creating source listing or mingled source/object listing. Probable cause: source file edited during compilation. (This error

would be most likely to occur on a multitasking system, where the compilation could be done as a "background" process.)

- 41 cannot open compiler intermediate file — no more files

The compiler is unable to create intermediate files used in the compilation process because no more file handles are available. This can usually be corrected by changing the FILES=number line in **CONFIG.SYS** to allow a larger number of open files (20 is the recommended setting).

- 42 cannot open compiler intermediate file — no such file or directory

The compiler is unable to create intermediate files used in the compilation process because the TMP environment variable is set to an invalid directory or path.

- 43 cannot open compiler intermediate file

The compiler is unable to create intermediate files used in the compilation process. The exact reason is unknown.

- 44 out of disk space for compiler intermediate file

The compiler is unable to create intermediate files used in the compilation process because no more space is available. To correct the problem, make more space available on the disk and recompile.

- 45 floating point overflow

The compiler has generated a floating-point exception while doing constant arithmetic on floating-point items at compile time, as in the following example:

```
float fp_val = 1.0e100;
```

In this case, the double-precision constant 1.0e100 exceeds the maximum allowable value for a floating-point data item.

- 46 bad *option* flag, would overwrite '*string1*' with '*string2*'

The specified *option* has been given more than once, with conflicting arguments *string1* and *string2*.

- 47 too many *option* flags, '*string*'
 There were too many occurrences of the given *option*; *string* contains the occurrence of *option* causing the error.
- 48 Unknown option '*character*' in '*optionstring*'
 The specified *character* is not a valid letter for *optionstring*.
- 49 invalid numerical argument '*string*'
 A numerical argument was expected instead of *string*.

H.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.

Number	Compilation Error Message
0	UNKNOWN ERROR Contact Microsoft Technical Support An unforeseen error condition has been detected by the compiler. Please report this condition to Microsoft, using the Software Problem Report at the back of this manual.
1	newline in constant A new-line character in a character or string constant must be in the escape sequence format (\n).
2	out of macro actual parameter space Arguments to preprocessor macros may not exceed 256 bytes.
3	Missing open parenthesis after keyword 'defined' Parentheses must surround the identifier to be checked in an #if directive.

Number	Compilation Error Message
4	expected 'defined(id)'
	An #if directive has a syntax error.
5	#line expected a line number
	A #line directive lacks the mandatory line-number specification.
6	#include expected a file name
	An #include directive lacks the mandatory file-name specification.
7	#define syntax
	A #define directive has a syntax error.
8	' <i>character</i> ' : unexpected in macro definition
	A macro definition uses a character incorrectly.
9	reuse of macro formal ' <i>identifier</i> '
	The parameter list in a macro definition contains two occurrences of the same identifier.
10	' <i>character</i> ' : unexpected in formal list
	The list of formal parameters in a macro definition uses <i>character</i> incorrectly.
11	' <i>identifier</i> ' : definition too big
	Macro definitions may not exceed 256 bytes.
12	missing name following '<'
	An #include directive lacks the mandatory file-name specification.
13	missing '>'
	The closing angle bracket (>) is missing from an #include directive.

Number	Compilation Error Message
14	preprocessor command must start as first non-whitespace
	Non-white-space characters appear before the number sign (#) of a preprocessor directive on the same line.
15	too many chars in constant
	A character constant is limited to a single character or escape sequence. (Multicharacter character constants are not supported.)
16	no closing single quote
	A character constant must be enclosed in single quotation marks.
17	illegal escape sequence
	The character or characters after the escape character (\) do not form a valid escape sequence.
18	unknown character '0xn'
	The given hexadecimal number does not correspond to a character.
19	expected preprocessor command, found ' <i>character</i> '
	The character following a number sign (#) is not the first letter of a preprocessor directive.
20	bad octal number ' <i>n</i> '
	The character <i>n</i> is not a valid octal digit.
21	expected exponent value, not ' <i>character</i> '
	The exponent of a floating-point constant is not a valid number.
22	' <i>n</i> ' : too big for char
	The number <i>n</i> is too large to be represented as a character.

Number	Compilation Error Message
23	divide by 0
	The second operand in a division operation (/) evaluates to zero, giving undefined results.
24	mod by 0
	The second operand in a remainder operation (%) evaluates to zero, giving undefined results.
25	' <i>identifier</i> ' : enum/struct/union type redefinition
	The given <i>identifier</i> has already been used for an enumeration, structure, or union tag.
26	' <i>identifier</i> ' : member of enum redefinition
	The given <i>identifier</i> has already been used for an enumeration constant, either within the same enumeration type or within another enumeration type with the same visibility.
27	compiler limit : struct/union nesting
	Nesting of structure and union definitions may not exceed five levels.
28	struct/union member needs to be inside a struct/union
	Structure and union members must be declared within the structure or union; likely cause: an enumeration declaration contains a declaration of a structure member, as in the following example:

```
enum a {  
    january,  
    february,  
    int march; /* structure declaration:  
                ** illegal  
                */  
};
```

Number	Compilation Error Message
29	' <i>identifier</i> ' : fields only in structs Only structure types may contain bit fields.
30	struct/union member redefinition The same identifier was used for more than one member of the same struct or union.
31	' <i>identifier</i> ' : function cannot be struct/union member A function cannot be a member of a structure; use a pointer to a function instead.
32	' <i>identifier</i> ' : base type with near/far/huge not allowed Declarations of structure and union members cannot use the near , far , and huge keywords.
33	' <i>identifier</i> ' : field has indirection The bit field is declared as a pointer (*), which is not allowed.
34	' <i>identifier</i> ' : field type too small for number of bits The number of bits specified in the bit-field declaration exceeds the number of bits in the given base type.
35	struct/union ' <i>identifier</i> ' : unknown size Structure or union has an undefined size.
36	left of ' <i>member</i> ' must have struct/union type In this message <i>member</i> will be a member designator in one of the following forms:
	<i>->identifier</i> <i>.identifier</i>
	The expression before the member-selection operator “->” is not a pointer to a structure or union type, or the expression before the member-selection operator “.” does not evaluate to a structure or union.

Number	Compilation Error Message
37	left of '->' or '.' specifies undefined struct/union ' <i>identifier</i> '
	The expression before the member-selection operator “->” or “.” identifies a structure or union type that is not defined.
38	' <i>identifier</i> ' : not struct/union member
	The given <i>identifier</i> is used in a context that requires a structure or union member.
39	'->' requires struct/union pointer
	The expression before the member-selection operator “->” is a structure or union name, not a pointer to a structure or union as expected.
40	'.' requires struct/union name
	The expression before the member-selection operator “.” is a pointer to a structure or union, not a structure or union name as expected.
41	keyword 'enum' illegal
	The enum keyword appears in a structure or union declaration, or an enum type definition is not formed correctly.
42	keyword 'enum' required
	The enum keyword is required in declarations of enumeration types.
43	illegal break
	A break statement is legal only when it appears within a do , for , while , or switch statement.
44	illegal continue
	A continue statement is legal only when it appears within a do , for , or while statement.
45	' <i>identifier</i> ' : label redefined
	The given <i>identifier</i> appears before more than one statement in the same function.

Number	Compilation Error Message
46	illegal case
	The case keyword may only appear within a switch statement.
47	illegal default
	The default keyword may only appear within a switch statement.
48	more than one default
	A switch statement contains too many default labels (only one is allowed).
49	cast has illegal formal parameter list
	A formal parameter list is given in a type-cast expression.
50	non-integral switch expression
	Switch expressions must be integral.
51	case expression not constant
	Case expressions must be integral constants.
52	case expression not integral
	Case expressions must be integral constants.
53	case value <i>n</i> already used
	The case value <i>n</i> has already been used in this switch statement.
54	expected '(' to follow ' <i>identifier</i> '
	The context requires parentheses after the function <i>identifier</i> .
55	expected formal parameter list, not a type list
	An argument-type list appears in a function definition instead of a formal parameter list.

Number	Compilation Error Message
56	illegal expression An expression is illegal because of a previous error. (The previous error may not have produced an error message.)
57	expected constant expression The context requires a constant expression.
58	constant expression is not integral The context requires an integral constant expression.
59	syntax error : ' <i>token</i> ' The given <i>token</i> caused a syntax error.
60	syntax error : EOF The end of the file was encountered unexpectedly, causing a syntax error; this can be caused by leaving out the final closing curly brace ({}) at the end of your program.
61	syntax error : identifier ' <i>identifier</i> ' The given <i>identifier</i> caused a syntax error.
62	type ' <i>identifier</i> ' unexpected The given type is misused.
63	' <i>identifier</i> ' : not a function The given <i>identifier</i> was not declared as a function, but an attempt was made to use it as a function.
64	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.
65	' <i>identifier</i> ' : undefined The given <i>identifier</i> is not defined.

Number	Compilation Error Message
66	cast to function returning . . . is illegal An object cannot be cast to a function type.
67	cast to array type is illegal An object cannot be cast to an array type.
68	illegal cast A type used in a cast operation is not a legal type.
69	cast of 'void' term to non-void The void type may not be cast to any other type.
70	illegal sizeof operand The operand of a sizeof expression must be an identifier or a type name.
71	' <i>class</i> ' : bad storage class The given storage <i>class</i> cannot be used in this context.
72	' <i>identifier</i> ' : initialization of a function Functions cannot be initialized.
73	' <i>identifier</i> ' : cannot initialize array in function Arrays can only be initialized at the external level.
74	cannot initialize struct/union in function Structures and unions can only be initialized at the external level.
75	' <i>identifier</i> ' : array initialization needs curly braces The braces ({}) around an array initializer are missing.
76	' <i>identifier</i> ' struct/union initialization needs curly braces The braces ({}) around a structure or union initializer are missing.

Number	Compilation Error Message
77	non-integral field initializer ' <i>identifier</i> '
	An attempt is made to initialize a bit-field member of a structure with a nonintegral value.
78	too many initializers
	The number of initializers exceeds the number of objects to be initialized.
79	' <i>identifier</i> ' is an undefined struct/union
	The given <i>identifier</i> is declared as a structure or union type that has not been defined.
80	' <i>expression</i> ' was the use of the struct/union
	An undefined structure or union type variable is used in the given <i>expression</i> .
81	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.
82	redefinition of formal parameter ' <i>identifier</i> '
	A formal parameter to a function is redeclared within the function body.
83	array ' <i>identifier</i> ' already has a size
	The dimensions of the given array have already been declared.
84	function ' <i>identifier</i> ' already has a body
	The given function has already been defined.
85	' <i>identifier</i> ' : ignored
	A parameter declaration was given in a function definition for a nonexistent formal parameter.

Number	Compilation Error Message
86	' <i>identifier</i> ' : redefinition
	The given <i>identifier</i> was defined more than once.
87	' <i>identifier</i> ' : missing subscript
	The definition of an array with multiple subscripts is missing a subscript value for a dimension other than the first dimension, as in the following example:
	<pre>int func(a) char a[10][]; { . . . }</pre>
	<pre>int func(a) char a[] [5]; { . . . }</pre>
88	use of undefined struct/union ' <i>identifier</i> '
	The given <i>identifier</i> was used to refer to a structure or union type that is not defined.
89	typedef specifies a near/far function
	Conflict between near or far used in a typedef declaration and near or far of declared item, as in the following example:
	<pre>typedef int far FARFUNC(); FARFUNC near *fp;</pre>
90	function returns array
	A function may not return an array. (It may return a pointer to an array.)

Number	Compilation Error Message
91	function returns function
	A function cannot return a function. (It can return a pointer to a function.)
92	array element type cannot be function
	Arrays of functions are not allowed; however, arrays of <i>pointers</i> to functions are allowed.
94	label ' <i>identifier</i> ' was undefined
	The function does not contain a statement labeled with the given <i>identifier</i> .
95	parameter has type void
	Formal parameters and arguments to functions cannot have type void ; they can, however, have type void * (pointer to void).
96	struct/union comparison illegal
	You cannot compare two structures or unions. (You can, however, compare individual members of structures and unions.)
97	illegal initialization
	Attempted to initialize variable using nonconstant values.
98	non-address expression
	An attempt was made to initialize an item that is not an lvalue.
99	non-constant offset
	An initializer uses a nonconstant offset.
100	illegal indirection
	The indirection operator (*) was applied to a nonpointer value.

Number	Compilation Error Message
101	'&' on constant
	The “address-of” operator requires an lvalue as its operand.
102	'&' requires lvalue
	The address-of operator can only be applied to lvalue expressions.
103	'&' on register variable
	Register variables cannot have their addresses taken.
104	'&' on bit field ignored
	Bit fields cannot have their addresses taken.
105	'operator' needs lvalue
	The given <i>operator</i> must have an lvalue operand.
106	'operator' : left operand must be lvalue
	The left operand of the given <i>operator</i> must be an lvalue.
107	illegal index, indirection not allowed
	A subscript was applied to an expression that does not evaluate to a pointer.
108	non-integral index
	Only integral expressions are allowed in array subscripts.
109	subscript on non-array
	A subscript was used on a variable that is not an array.
110	'+' : 2 pointers
	Two pointers cannot be added.
111	pointer + non-integral value
	Only integral values may be added to pointers.

Number	Compilation Error Message
112	illegal pointer subtraction
	Only pointers that point to the same type may be subtracted.
113	'-' : right operand pointer
	The right-hand operand in a subtraction operation (-) is a pointer, but the left-hand operand is not.
114	'operator' : pointer on left; needs integral right
	The left operand of the given <i>operator</i> is a pointer; the right operand must be an integral value.
115	'identifier' : incompatible types
	An expression contains types that are not compatible.
116	<i>operator</i> : bad <i>left-or-right</i> operand
	The specified operand of the given <i>operator</i> is illegal for that operator.
117	'operator' : illegal for struct/union
	Structure and union type values are not allowed with the given <i>operator</i> .
118	negative subscript
	A value defining an array size was negative.
119	'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:
	<pre>typedef int *P_INT; typedef short *P_SHORT; /* this declaration is illegal */ P_SHORT P_INT p;</pre>

Number	Compilation Error Message
120	'void' illegal with all types
	The void type cannot be used in declarations with other types.
121	typedef specifies different enum
	Attempted to use a type declared in a typedef statement to specify both an enum type and another type.
122	typedef specifies different struct
	Attempted to use a type declared in a typedef statement to specify both a struct type and another type.
123	typedef specifies different union
	Attempted to use a type declared in a typedef statement to specify both a union type and another type.
124	CODE GENERATION ERROR
	Contact Microsoft Technical Support
	The compiler could not generate code for an expression. Usually this occurs with a complex expression. Try rearranging the expression. Please report this error using the Software Problem Report at the back of this manual.
125	allocation exceeds 64K for ' <i>identifier</i> '
	The given item exceeds the limit of 64K. The only items that are allowed to exceed 64K are huge arrays.
126	auto allocation exceeds 32K
	The space allocated for the local variables of a function exceeds the limit of 32K.
127	parameter allocation exceeds 32K
	The storage space required for the parameters to a function exceeds the limit of 32K.

Number	Compilation Error Message
128	<p>huge '<i>identifier</i>' cannot be aligned to segment boundary</p> <p>The given array violates one of the restrictions imposed on huge arrays; see Section 8.2.5, "Creating Huge Model Programs," for more information on these restrictions.</p>
129	<p>static function '<i>identifier</i>' not found</p> <p>A forward reference was made to a static function that is never defined.</p>
130	<p>#line expected a string containing the file name</p> <p>Invalid syntax for #line directive (missing file name).</p>
131	<p>attributes specify more than one near/far/huge</p> <p>More than one near, far, or huge attribute applied to an item, as in the following example:</p>
132	<pre>typedef int near NINT; NINT far a; /* Illegal */</pre> <p>syntax error : unexpected identifier</p> <p>Identifier seen in syntactically illegal context.</p>
133	<p>array '<i>identifier</i>' : unknown size</p> <p>Attempt to declare unsized array as local variable, as in the following example:</p>
	<pre>int mat_add(array1) int array1[]; /* Legal */ { int array2[]; /* Illegal */ . . }</pre>

Number	Compilation Error Message
134	symbol too large Size of huge array exceeds compiler limit (2^{32} bytes).
135	missing ')' in macro expansion A macro reference with arguments is missing a closing parenthesis.
137	empty character constant The illegal character constant '' was used.
138	unmatched close comment '*/' Compiler detected */ without matching /*. This usually indicates an attempt to use illegal nested comments.
139	type following ' <i>type</i> ' is illegal Illegal type combination, such as the following:
	long char a; /* Illegal */
140	argument type cannot be function returning '...' A function is declared as a formal parameter of another function, as in the following example:
	int func1(a) int a(); /* Illegal */
141	value out of range for enum constant An enum constant has a value outside the range of values allowed for type int.
142	ellipsis requires three periods The compiler has detected the token .. and assumes ... was intended.

H.3.4 Command-Line Error 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.

Number	Command Line Error Message
0	UNKNOWN COMMAND LINE ERROR Contact Microsoft Technical Support An unknown error condition has been detected by the compiler. Please report this condition to Microsoft using the Software Problem Report at the back of this manual.
1	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.
2	listing has precedence over assembly output Two different listing options were chosen; the assembly listing is not created.
3	a previously defined model specification has been overridden Two different memory models are specified; the model specified later is used.
4	unknown -A subswitch ' <i>letter</i> ' A letter given with the /A option is not recognized.
5	only one memory model allowed You must choose one memory model; you cannot specify more than one.
6	missing source file name You must give the name of the source file to be compiled.
7	too many commas Too many commas appear on the command line.

Number	Command Line Error Message
8	comma needed before ' <i>filename</i> ' The fields in the command line must be separated by commas.
9	a file name (not a path name) is required The name of a directory is given where the name of a file is required.
10	ignoring unknown flag ' <i>string</i> ' One of the options given on the command line is not recognized and is ignored.
12	too many <i>option</i> flags, ' <i>string</i> ' Too many letters are given with a specific option (for example, with the /O option).
13	unknown option <i>character</i> in ' <i>optionstring</i> ' One of the letters in the given option is not recognized.
15	80186/286 selected over 8086 for code generation Both the /G0 option and either the /G1 or /G2 option are given; /G1 or /G2 takes precedence.
16	optimizing for space over time This message confirms that the /Os option is used for optimizing:
17	unknown floating point option The specified floating-point option (an /FP option) is not one of the five valid options.
18	only one floating point model allowed You can give only one of the five floating-point (/FP) options on the command line.

Number	Command Line Error Message
19	could not execute ' <i>filename</i> '
	Found the specified <i>filename</i> , but could not execute it for some reason (most likely cause: bad .EXE file format).
20	could not execute ' <i>filename</i> '
	Please insert diskette and hit any key
	Could not find the given <i>filename</i> in the current working directory or any of the other directories named in the PATH statement.
21	too many linker flags on command line
	Attempted to pass more than 128 separate options and object files to the linker (for CL only).
22	only one of -P/-E/-EP allowed, -P selected
	Only one preprocessor output option can be specified at one time.
23	-C ignored (must also specify -P or -E or -EP)
	The -C option must be used in conjunction with one of the preprocessor output flags, -E , -EP , or -P .
24	too many open files, cannot redirect ' <i>filename</i> '
	No more file handles available to redirect the output of the -P option to a file. Try editing your CONFIG.SYS file and increasing the value <i>num</i> on the line <i>files=num</i> (if <i>num</i> is less than 20).
25	-Md not allowed with -ND
	The -Au option (SS != DS , load DS) requires a new name for the default data segment.
26	unknown 'option' substring 'character'
	Unknown substring <i>character</i> used with the given <i>option</i> .

Number	Command Line Error Message
27	incomplete model specification
	The -A <i>string</i> option requires all three characters (data-pointer size, code-pointer size, and segment setup) in <i>string</i> .
28	-ND not allowed with -Ad
	Cannot rename default data segment unless the -Au option (SS != DS , load DS) is given.
29	-ND not allowed with -Aw
	Cannot rename default data segment unless the -Au option (SS != DS , load DS) is given.
30	non-standard model -- defaulting to small-model libraries
	Nonstandard memory model has been specified with the -A <i>string</i> option. The library search records in the object model are set to use the small-model libraries.
31	threshold only for far/huge data, ignored
	The -Gt option cannot be used in memory models that have near data pointers. It can be used only in compact, large, and huge models.
32	assembly files are not handled
	File name with extension .ASM specified. Compiler cannot invoke MASM automatically, so it cannot assemble such files.
34	-Gp not implemented, ignored
	MS-DOS version of compiler does not support profiling.
35	-Gw and -ND <i>name</i> are not compatible
	Cannot rename the default data segment to <i>name</i> when -Gw given, since -Gw also requires -Aw .
36	-Gw and -Au flags are incompatible
	Cannot use the -Au option (SS != DS , load DS) with -Gw , since -Gw also requires -Aw .

Number	Command Line Error Message
37	Preprocessing overrides source listing Only preprocessor listing generated, since cannot generate both a source listing and a preprocessor listing at the same time.
38	function declarations override source listing Cannot generate both a source-listing file and the function prototype declarations at the same time.
39	cannot open linker cmd file Cannot open the response file used to pass object-file names and options to the linker; one possible cause: there is another file which is read only and has the same name as the response file.
43	combined listing has precedence over object listing When -Fc is specified along with either -Fl or -Fa , the combined listing (-Fc) will be created.
44	cannot overwrite the source file Source file specified as an output-file name. The compiler will not allow the source file to be overwritten by one of the compiler output files.
46	-Gc option requires extended keywords to be enabled (-Ze) The -Gc option requires the extended keyword cdecl to be enabled if library functions are to be accessible.
47	invalid numerical argument ' <i>string</i> ' Non-numerical string was specified following an option that requires a numerical argument.

H.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 H.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 H.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 <i>Microsoft C Compiler Language Reference</i> 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 Maximum number of actual arguments to a macro definition Maximum length of an actual preprocessor argument Maximum level of nesting for #if, #ifdef, and #ifndef directives Maximum level of nesting for include files	512 bytes 8 arguments 256 bytes 32 levels 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.

H.4 LINK Error Messages

This section lists the error messages that can occur when linking programs with the Microsoft Overlay Linker, **LINK**. The messages are in alphabetical order.

Ambiguous switch error: "option"

User did not enter a unique option name after the option indicator (/).
For example, the command

```
LINK /N main;
```

will generate this error, since **LINK** can't tell which of the three options beginning with the letter "N" you intended to use. See Chapter 4, "Linking," for more information on **LINK** options.

Array element size mismatch

A far communal array has been declared with two or more different array-element sizes (for example, declared once as an array of characters and once as an array of real numbers). This error cannot occur with object files produced by the assembler. It only occurs with the Microsoft C Compiler and any other compiler that supports far communal arrays.

Attempt to put segment *name* in more than one group
in file *filename*

A segment was declared to be a member of two different groups.
Correct the source and recreate the object files.

Bad value for cparMaxAlloc

The number specified using the **/CPARMAXALLOC** option is not in the range 1 to 65535. See Section 4.6.10, "Setting the Maximum Allocation Space."

Cannot find library: *filename.lib*. Enter new file spec:

The linker cannot find *filename.lib*. The user should respond to the prompt with a new file name, a new path specification, or both.

Cannot open list file

The disk or the root directory is full. Delete or move files to make space.

Cannot open response file

LINK cannot find the response file specified by the user. This usually indicates a typing mistake.

Cannot nest response files

User named a response file within a response file.

Cannot open run file

The disk or the root directory is full. Delete or move files to make space.

Cannot open temporary file

The disk or the root directory is full. Delete or move files to make space.

Cannot reopen list file

User did not actually replace the original disk when asked to. Restart the linker.

Common area longer than 65536 bytes

User's program has more than 64K of communal variables. This error cannot appear with object files generated by the assembler. It only occurs with programs produced by the Microsoft C Compiler or other compilers that support communal variables.

Data record too large

LEDATA record (in an object module) contains more than 1024 bytes of data. This is a translator error. Note the translator (compiler or assembler) that produced the incorrect object module and the circumstances. Notify Microsoft using the Software Problem Report at the back of this manual. **LEDATA** is an MS-DOS term. It is explained in the *MS-DOS Programmer's Reference Manual* and in some other MS-DOS reference books.

Dup record too large

LIDATA record (in an object module) contains more than 512 bytes of data. Most likely, an assembly module contains a structure definition that is very complex, or a series of deeply nested **DUP** operators, such as the following example:

```
alpha    DB      10 DUP(11 DUP(12 DUP(13 DUP(...))))
```

Simplify and reassemble. **LIDATA** is an MS-DOS term. It is explained in the *MS-DOS Programmer's Reference Manual* and in some other MS-DOS reference books.

filename is not a valid library

The file specified as a library file is invalid. **LINK** will abort.

Fixup overflow near *number*
in segment *name* in *filename* offset *number*

Some possible causes are as follows:

- A group is larger than 64K.
- 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, as in the following example:

```
code    SEGMENT public 'CODE'
        EXTRN   main:far
start  PROC    far
        call    main
        ret
start  ENDP
code   ENDS
```

The following construction is preferred:

```
        EXTRN   main:far
code   SEGMENT public 'CODE'
start  PROC    far
        call    main
        ret
start  ENDP
code   ENDS
```

Revise the source and recreate the object file.

Incorrect DOS version, use DOS 2.0 or later

LINK will not run on versions of MS-DOS or PC-DOS prior to 2.0.
Reboot your system with a valid version, and try linking again.

Insufficient stack space

There is not enough memory to run the linker.

Interrupt number exceeds 255

A number greater than 255 has been given as a value for the
/OVERLAYINTERRUPT option. The number must be in the range
0 to 255. You should not use the **/OVERLAYINTERRUPT** option
with the Microsoft C Compiler.

Invalid numeric switch specification

An incorrect value was entered for one of the linker switches (options).
For example, a character string was entered for an option that requires
a numeric value.

Invalid object module

One of the object modules is invalid. Try recompiling. If the error per-
sists, contact Microsoft using the Software Problem Report at the back
of this manual.

NEAR/HUGE conflict

Conflicting near and huge definitions for a communal variable. This
error can only occur with programs produced by Microsoft C or other
compilers that support communal variables.

Nested left parentheses

User has made a typing mistake while specifying the contents of an
overlay on the command line. See Section 4.5, "Using Overlays."

No object modules specified

User failed to supply the linker with any object-file names.

Out of space on list file

Disk on which list file is being written is full. Free more space on the
disk and try again.

Out of space on run file

Disk on which .EXE file is being written is full. Free more space on the disk and try again.

Out of space on scratch file

Disk in default drive is full. Delete some files on that disk, or replace with another disk, and restart the linker.

Overlay manager symbol already defined: *name*

User has defined a symbol name that conflicts with one of the special overlay-manager names. Change the incorrect name and relink.

MASM does not have an overlay manager, so this problem can only occur if you are linking with a library from a high-level language that supports overlays.

Relocation table overflow

More than 32768 long calls, long jumps, or other long pointers in the user's program. Rewrite program, replacing long references with short references where possible, and recreate object module. Note: Pascal and FORTRAN users should first try turning off the debugging option.

Segment limit set too high

The limit on the number of segments allowed was set too high (more than 1024) using the /SEGMENTS option. See Section 4.6.11, "Controlling Segments."

Segment limit too high

There is insufficient memory for the linker to allocate tables to describe the number of segments requested (the default of 128 or the value specified with the /SEGMENTS option). Try linking again using the /SEGMENTS option to select a smaller number of segments (for example, 64 if the default was used previously), or free some memory by eliminating resident programs or shells.

Segment size exceeds 64K

User has a small-model program with more than 64K of code, or user has a medium-model program with more than 64K of data. Try compiling and linking medium or large model.

Stack size exceeds 65536 bytes

The size specified for the stack using the /STACK option is more than 65536 bytes. See Section 4.6.9, "Controlling Stack Size."

Symbol table overflow

The user's program has more than 256K of symbolic information (publics, externals, segments, groups, classes, files, etc.). Combine modules and/or segments and recreate the object files. Eliminate as many public symbols as possible.

Terminated by user

The user entered CONTROL-C.

Too many external symbols in one module

User's object module specified more than the limit of 1023 external symbols. Break up the module.

Too many group-, segment-, and class-names in one module

User's program contains too many group, segment, and class names. Reduce the number of groups, segments, or classes, and recreate the object files.

Too many groups

User's program defines more than 20 groups, not counting **DGROUP**. Reduce the number of groups.

Too many GRPDEFs in one module

LINK encountered more than 21 group definitions (**GRPDEF**) in a single module. Reduce the number of **GRPDEF**s or split the module. The term **GRPDEF** is explained in the *MS-DOS Programmer's Reference Manual* and in some other reference books on MS-DOS.

Too many libraries

User tried to link with more than 16 libraries. Combine libraries, or use modules that require fewer libraries.

Too many overlays

User's program defines more than 63 overlays. Reduce the number of overlays.

Too many segments

The user's program has more than the maximum number of segments as specified by the default of 128 or by the **SEGMENTS** option. Relink using the **/SEGMENTS** option with an appropriate number of segments. See Section 4.6.11, "Controlling Segments."

Too many segments in one module

The user's object module has more than 255 segments. Split the modules or combine segments.

Too many TYPDEFs

An object module contains more than 255 **TYPDEF** records. These records describe communal variables. This error can only appear with programs produced by the Microsoft C Compiler or other compilers that support communal variables. **TYPDEF** is an MS-DOS term. It is explained in the *MS-DOS Programmer's Reference Manual* and in some other reference books on MS-DOS.

Unexpected end-of-file on library

The disk containing the library has probably been removed. Replace the disk containing the library and try again.

Unexpected end-of-file on scratch file

Disk with temporary output file was removed. See Section 4.2.11, "The Temporary File," in Chapter 4, "Linking."

Unmatched left parenthesis

User has made a typing mistake while specifying the contents of an overlay on the command line.

Unmatched right parenthesis

User has made a typing mistake while specifying the contents of an overlay on the command line.

Unrecognized switch error: *option*

User entered an unrecognized character after the option indicator (/), as in the following example:

```
LINK /ABCDEF main;
```

Unresolved externals

A symbol was declared external in one module, but it was not declared public in the module in which it was defined. This should not happen in Microsoft C, except with misspelled or case-sensitive names, but it could happen with an assembly-language module. A symbol must be defined and declared public (using the **PUBLIC** directive) in one and only one module before it can be used as an external symbol (using the **EXTRN** directive) by other modules. Linking with versions of the linker earlier than 2.4 might cause this message, since these older linkers search libraries only once.

VM.TMP is an illegal file name and has been ignored

User has specified **VM.TMP** as an object-file name. Rename file and link again.

Warning: no stack segment

User's program contains no stack segment specified with **stack** combine type. This message should not appear with modules compiled with the Microsoft C Compiler, but it could appear with an assembly-language module. Normally, every program should have a stack segment with the combine type specified as **stack**. You can ignore this message if you have a specific reason for not defining a stack or for defining one without the **stack** combine type. Linking with versions of the linker earlier than 2.4 might cause this message, since these older linkers search libraries only once.

Warning: too many public symbols

The **/MAP** option was used to request a sorted listing of public symbols in the map file, but there are too many symbols to sort. The linker will produce an unsorted listing of the public symbols.

H.5 Library-Manager Error Messages

The following error messages may be displayed by the Microsoft Library Manager, **LIB**:

cannot create extract file *filename*

The disk or root directory is full, or the extract file specified by *filename* already exists with read-only protection. Make space on the disk or change the protection of the extract file.

cannot create new library

The disk or root directory is full, or the library file already exists with read-only protection. Make space on the disk or change the protection of the library file.

cannot open response file

The given response file was not found.

cannot open VM.TMP

The disk or root directory is full. Delete or move files to make space.

cannot read from VM

Note the circumstances of the failure and notify Microsoft using the Software Problem Report at the back of this manual.

cannot rename old library

LIB could not rename the old library to have a **.BAK** extension because the **.BAK** version already exists with read-only protection. Change the protection on the old **.BAK** version.

cannot reopen library

The old library could not be reopened after it was renamed to have a **.BAK** extension.

cannot write to VM

Note the circumstances of the failure and notify Microsoft using the Software Problem Report at the back of this manual.

comma or newline expected

A comma or carriage return was expected in the command line, but did not appear, or a comma was not expected and appeared in an inappropriate place, as in the following line:

```
LIB math.lib,-mod1+mod2;
```

The line should have been entered as follows:

```
LIB math.lib -mod1+mod2;
```

error writing to cross reference file

The disk or root directory is full. Delete or move files to make space.

error writing to new library

The disk or root directory is full. Delete or move files to make space.

Free: not allocated

Note the circumstances of the failure and notify Microsoft using the Software Problem Report at the back of this manual.

insufficient memory

LIB does not have enough memory to run. Remove any shells or resident programs and try again, or add more memory.

internal failure

Note the circumstances of the failure and notify Microsoft using the Software Problem Report at the back of this manual.

invalid library

The library does not conform to the format expected by **LIB**.

Invalid object module *name* near *location*
in file *libraryname*

The module specified by *name* is not a valid object module.

Mark: not allocated

Note the circumstances of the failure and notify Microsoft using the Software Problem Report at the back of this manual.

missing terminator

The response to the "Output library:" prompt was not terminated by a carriage return, or the last line of the response file used to start **LIB** does not end with a carriage return.

no more virtual memory

Note the circumstances of the failure and notify Microsoft using the Software Problem Report at the back of this manual.

page size too small

Page size specified with the **/PAGESIZE** option must be 16 bytes or larger.

syntax error

The given command did not follow correct **LIB** syntax as specified in Chapter 6, "Managing Libraries."

syntax error (bad input)

The given command did not follow correct **LIB** syntax as specified in Chapter 6, "Managing Libraries."

syntax error (bad file spec)

A command operator such as a minus sign (-) was given without a following module name.

syntax error (switch name expected)

A forward slash (/) was given without the **PAGESIZE** option.

syntax error (switch val expected)

The **/PAGESIZE** option was given without a following value.

too many symbols

The maximum number of symbols allowed in a library file is 4609.

unexpected EOF on command input

An end-of-file character was received prematurely in response to a prompt.

unknown switch

An unknown option was given. The **/PAGESIZE** option is the only one currently recognized by **LIB**.

write to extract file failed

The disk or root directory is full. Delete or move files to make space.

write to library file failed

The disk or root directory is full. Delete or move files to make space.

H.6 MAKE Error Messages

Most error messages displayed by the Microsoft Program Maintenance Utility, **MAKE**, have the following form:

filename linenumber : message

The *filename* is the **MAKE** description file. The *linenumber* is the line where the error occurred. If an error occurs after **MAKE** has finished reading through the file, the *linenumber* will be listed as 1 even though this will not be the correct line number. The *message* is one of the error messages listed below:

Exec not available on DOS 1.x

MAKE requires MS-DOS Version 2.0 or later.

expansion too big

A line with macros expands to longer than 512 bytes. Try rewriting the **MAKE** description file to use two short lines instead of one long one.

line too long

A line in the make description file is longer than 128 characters. Try rewriting the make description file to use two short lines instead of one long one.

make: *command - errorcode*

One of the programs or commands called in the make description file was not able to execute correctly. **MAKE** terminates and displays the command followed by the code of the error that caused it to fail. Error codes are described in Appendix E, "Using Exit Codes."

make: colon missing in '*filename*'

A line that should be a target/dependent line lacks a colon indicating the separation between target and dependent. **MAKE** expects any line following a blank line to be a target/dependent line.

make: dependent '*filename*' does not exist,
target '*filename*' not built

MAKE could not continue because a required dependent file did not exist. Make sure all named files are present and that they are spelled correctly in the **MAKE** description file.

make: infinitely recursive macro

A circular chain of macros was defined, as in the following example:

```
A=$ (B)  
B=$ (C)  
C=$ (A)
```

make: multiple source

An inference rule has been defined more than once.

make: out of memory

MAKE has run out of memory for processing the **MAKE** description file. Try to reduce the size of the **MAKE** description file by reorganizing or splitting it.

make: out of space

MAKE has run out of memory for processing the **MAKE** description file. Try to reduce the size of the **MAKE** description file by reorganizing or splitting it.

make: syntax error

The **MAKE** description file has a line beginning with an equal sign (=).

make: target does not exist '*filename*'

This usually does not indicate an error. It warns the user that the target file did not exist. **MAKE** executes any commands given in the target/dependent description since in many cases the target file will be created by a later command in the **MAKE** description file.

Stack overflow

Recursive macros have used up all available memory. Reduce the number or levels of nested macros.

usage: make [/n] [/d] [/i] [/s] [name=value ...] file

MAKE has not been invoked correctly. Try entering the command line again with the syntax shown in the message.

H.7 EXEPACK Error Messages

The Microsoft EXE File Compression Utility, **EXEPACK**, generates the following error messages:

exepack: (warning) omitting debug data from output file
EXEPACK strips symbolic debug information from the input file.

exepack: can't change load-high program

When the minimum allocation value and the maximum allocation value are both zero, the file cannot be compressed.

exepack: error reading relocation table

The file cannot be compressed because the relocation table cannot be found or is invalid.

exepack: invalid .EXE file (actual length < reported)

The second and third fields in the file header indicate a file size greater than the actual size.

exepack: invalid .EXE file (bad header)

The given file is not an executable file or has an invalid file header.

exepack: *filename*: No such file or directory

The file specified by *filename* cannot be found.

exepack: *filename*: Permission denied

The file specified by *filename* is a read-only file.

exepack: out of memory

The **EXEPACK** utility does not have enough memory to operate.

Out of space on output file

The disk or root directory is full. Delete or move files to make space.

exepack: too many segments in relocation table

The given file is too large to be compressed in the available system memory.

usage: exepack <infile> <outfile>

The **EXEPACK** command line was not specified properly. Try again using the syntax shown.

You may also encounter MS-DOS error messages if the **EXEPACK** program cannot read from, write to, or create a file.

H.8 EXEMOD Error Messages

The Microsoft EXE File-Header Utility, **EXEMOD**, generates the following error messages:

exemod: can't change load-high program

When the minimum allocation value and the maximum allocation value are both zero, the file cannot be modified.

exemod: file not .EXE

EXEMOD automatically appends the **.EXE** extension to any file name without an extension; in this case, no file with the given name and an **.EXE** extension could be found.

exemod: invalid .EXE file (actual length < reported)

The second and third fields in the file header indicate a file size greater than the actual size.

exemod: invalid .EXE file (bad header)

The specified file is not an executable file or has an invalid file header.

exemod: min > max (correcting max)

If the minimum allocation value is greater than the maximum allocation value, the maximum allocation value is adjusted. This is a warning message only; the modification is still performed.

exemod: min < stack (correcting min)

If the minimum allocation value is not enough to accommodate the stack (either the original stack request or the modified request), the minimum allocation value is adjusted. This is a warning message only; the modification is still performed.

exemod: *filename*: No such file or directory

The file specified by *filename* cannot be found.

exemod: *filename*: Permission denied

The file specified by *filename* is a read-only file.

exemod: (warning) packed file

The given file is a packed file. This is a warning only. **EXEMOD** will still modify the file. The values shown if you ask for a display of MS-DOS header values will be the values after the packed file is expanded.

usage: exemod file [-/h] [-/stack n] [-/max n] [-/min n]

The **EXEMOD** command line was not specified properly. Try again using the syntax shown. Note that the option indicator can be either a slash (/) or a dash (-). The single brackets ([]) in the error message indicate that your choice of the item within them is optional.

The **EXEMOD** utility also produces error messages when the file header is not in recognizable .EXE format, or if errors occur in reading from, or writing to, a file.

H.9 SETENV Error Messages

The Microsoft Environment Table Utility, **SETENV**, generates the following error messages:

setenv: Envsize must be <= 65520

You specified an environment size greater than 65520, the maximum size allowed.

setenv: Envsize must be >= 160

You specified an environment size less than 160, the minimum size allowed.

setenv: Maximum for Version 3.1 = 992

The user specified a file that was recognized as **COMMAND.COM** for MS-DOS, Version 3.1, and gave an environment size greater than 992, the maximum allowed for that version.

setenv: <filename>: No such file or directory

The specified file was not found, or it was a directory or some other special file.

setenv: <filename>: Permission denied

The specified file is a read-only file.

setenv: unrecognizable COMMAND.COM

The **COMMAND.COM** file was not one of the accepted versions
(PC-DOS, versions 2.0, 2.1, 2.11, 3.0, and 3.1).

usage: setenv <command.com> [envsize]

The command line was not specified properly. This usually indicates
that the wrong number of arguments was given. Try again with the
syntax shown in the message.

Index

- \$ (dollar sign), 325
- & (ampersand), in LIB command, 145, 149
- * (asterisk)
 - CL command, used in, 292
 - example, command symbol, 149
 - LIB command symbol, used as, 146, 152
 - wild-card character, 33, 134
- + (plus sign)
 - command symbol, example, 147, 149
 - LIB command symbol, used as, 145, 151
 - LINK command, used in, 101
- , (comma) MSC command line, used in, 58
 - (dash) option character
 - (minus sign), as LIB command symbol, 145, 147, 149, 152
 - * (minus sign-asterisk), as LIB command symbol, 146, 153
 - + (minus sign-plus sign), as LIB command symbol, 145, 147, 152
- / (forward slash)
 - option character, 60
 - LINK option character, used as, 111
- ; (semicolon) in
 - LIB command, 144, 147, 150
 - LINK command, 101
 - MSC command, 57
- ? (question mark)
 - CL command, used in, 292
 - wild-card character, 134
- _ (underscore) in
 - global names, 341
 - identifiers, 224

- 80186/80188 processor, 40, 84
- 80286 processor, 40, 84
- 8087/80287 coprocessor, 39, 79, 80
 - control of, 81
 - in-line instructions, 80, 81
 - library, 32, 198
 - suppression of, 200

- 87.LIB, 31, 80, 198

- /A option, 186, 187
- abs function, 331
- /AC option, 175
- Address space, 353
- Addresses
 - parameters passed by, 234
 - passing, 231
- /AL option, 176
- Alias checking, 91, 329
- Align type, 123, 217
- Alignment. *See* storage alignment
- allmem routine, 330
- Alternate calling sequence, 221
- Alternate floating-point library, 32
- Alternate math library, 82, 198
- /AM option, 175
- Ampersand (&), in LIB command, 145, 149
- argc variable, 132
- Arguments
 - command line, 135
 - conversion, 219
 - F options, 293
 - LINK options, 112
 - macros, 409
 - MSC options, 60
 - procedure in mixed-language programming, 262
 - pushing, 219
- to main function. *See* main function
- variable number, to a function, 357
- wild card, on command line, 134
- Argument-type list, 77
- argv variable, 132
- Array identifiers, in previous versions of the compiler, 326
- Array limits, in previous versions of the compiler, 328
- Arrays, in mixed-language programming, 256
- /AS option, 174
- ASCII character codes, 269

- Assembly-language interface, 213
 - differences, 336
 - calling conventions, 337
 - case significance, 342
 - classes, 344
 - entry sequence, 338
 - exit sequence, 338
 - groups, 344
 - naming conventions, 341
 - register usage, 336
 - segment model, 342
 - segment names, 342, 344
 - stack checking, 339
 - stack setup, 338
 - program example, 226
- Assembly-listing file, 64, 68
- Asterisk (*)
 - CL command, used in, 292
 - LIB command symbol, 146, 152
 - wild-card character, 33, 134
- Attributes
 - calling conventions, 231
 - mixed-language programming, used in, 231
- AUTOEXEC.BAT file, 37
- AUX, 54
- /Aw option, Windows applications, 209
- Backup procedures, 17
- Batch files, 38, 46, 311
- BEGDATA class name, 120
- Bibliography, 11
- Binary mode, 33, 205
- BINMODE.OBJ, 33, 205
- Bit fields, 327, 351
- Bold font, 9-10
- Boolean, in mixed-language programming, 248
- BP register, 221, 224, 338
- Brackets, use of, 11
- BSS class name, 120
 - _BSS segment, 215
- Buffers, in CONFIG.SYS file, 39
- Byte length, 348
- Byte order, 350, 360
- C calling sequence, 219
- /C option, 75
- c option, 294, 296
- C primer, 11
- C programming language, 11
 - C1.EXE, 30, 34
 - C2.EXE, 30, 34
 - C3.EXE, 30, 34
- Calling conventions
 - described, 230
 - mixed-language programming, choice of, 230
 - mixed-language programming, specification of, 231
 - previous versions of the compiler, 337
- Calling sequence
 - alternate, 221
 - C, 219
- Capital letters, small, 11
- Capital letters, use of, 10
- Carriage-return-line-feed (CR-LF) translation, 205
- Case sensitivity, preservation of, 116
- Case
 - MS-DOS, 41
 - significance, 225
 - absence of, 116
 - file names, 53
 - LINK, 98, 116
 - names, 236
 - previous versions of the compiler, 342
 - XENIX, 41
- c_common segment, 215
- cdecl keyword, 193
- char constants, in previous versions of the compiler, 325
- char type
 - default changing, 196
 - previous versions of the compiler, 326
- Character-classification macros, 353
- Character set, 353
- check_stack pragma, 201
- _chkstk routine, 340
- chksym, 138
- CL command, 51, 98, 291
 - F options, 292
 - linking, 294
 - syntax, 291
- CL options
 - c, 242, 296
 - F, 296

CL options (*continued*)
 -Fe, 292, 296
 -Fm, 292, 296
 -Fs, 292
 -link, 295, 296
 XENIX compatible, 296

Class names, 217
 BEGDATA, 120
 BSS, 120
 CODE, 120
 FAR_BSS, 216
 FAR_DATA, 216
 STACK, 120

Class type, 124

Classes, 217, 344

CL.EXE, 30, 31

CLIBC.LIB, 33

CLIBFA.LIB, 82

CLIBFP.LIB, 33, 80, 198

/CO option, 115

CODE class name, 120

Code pointers, 186

Code-segment restrictions, 93

Code segments. *See* Text segments

Code size, optimization, 91

Combine class, 217

Combine type
 COMMON, 125
 PRIVATE, 125
 PUBLIC, 125
 STACK, 125

Combined-listing file, 68

Comma (,), in MSC command line, 58

Command
 prompts
 list file, 153
 symbols
 asterisk (*), 146, 149, 152
 minus sign (-), 145, 147, 149, 152
 minus sign-asterisk (-*), 146, 153
 minus sign-plus sign (-+), 145,
 147, 152
 plus sign (+), 145, 147, 149, 151

Command characters
 LIB, summary, 283
 LINK, summary, 280

Command line
 error messages, 372, 404
 length, maximum, 371
 LIB, 147
 LINK, 102

Command line (*continued*)
 messages, 86
 method, MSC, 57

Command-line arguments
 executable file, 131
 suppressing processing of, 135
 wild cards, 134

Commands
 CL. *See* CL command
 IF ERRORLEVEL, 47, 88
 notational conventions, 10
 PATH, 17, 35, 36, 37, 166
 SET, 17, 35, 36, 37
 summary, 273

Comments
 preservation of, 75
 previous versions of the compiler, 325

Compact model, 175
 library files, 31

Compatibility
 87.LIB, 82
 EM.LIB, 82
 floating-point options, 83
 XENIX options, 296

Compilation
 conditional, 71
 large programs, 92
 mixed-language programming, 264

Compile-only option, 294, 296

Compiler
 command line, partial, 58
 differences, 319, 354
 alias checking, 329
 array identifiers, 326
 array limits, 328
 bit fields, 327
 char constants, 325
 char type, 326
 comments, 325
 enum type, 326
 equality operators, 327
 function identifiers, 326
 identifiers, 325
 language definition, 325
 logical AND and OR operators,
 329
 lvalue expressions, 327
 macro definitions, 328
 preprocessor, 328
 relational operators, 327

- differences (*continued*)
 - strings, 326
 - structure identifiers, 327
 - structures and unions, 328
 - type casts, 327
 - uninitialized variables, external level, 377
 - union identifiers, 327
 - unsigned-long type, 326
 - versions 2.03 and earlier, 324
- documentation, 4
- error, in code generation, 372
- error messages, 371
 - command line, 372, 404
 - compilation, 372
 - fatal, 371, 382
 - internal, 86, 372
 - warning, 371, 373
- exit codes, 88
- limits, 409
- mixed-language programming,
 - versions required for, 229
- naming conventions, 69, 109
- options. *See* MSC options
- passes, 30
- prompts, response to, 52
- summary, 273
- termination, 52
- Complex numbers, in mixed-language programming, 260
- CON, 54
- CONFIG.SYS file, 38
- Consistency check, 144, 154
- CONST segment, 215
- Constant expressions, 328
- Constants
 - definition of, 71
 - size, maximum, 409
- Control, of
 - binary and text modes, 205
 - data loading, 121
 - floating-point operations, 198
 - LINK, 111
 - preprocessor, 70, 71, 73, 75
 - run-file loading, 121
 - segments, 119
 - stack size, 117
- Control program, 30
- CONTROL-C, 52, 102, 150
- Conventions, notational, 9
- Conversion, 219
- Conversion (*continued*)
 - pointer arguments, 183
 - short pointers to long integers, 322
- Converting from previous versions of the compiler. *See* Assembly-language interface; Compiler differences; Differences from previous versions of the compiler
- Coprocessor
 - See also* 8087/80287 coprocessor
 - 8087/80287 versions, 38, 79
 - suppressing use of, 200
- /CPARMAXALLOC option, 118
- creat function, 331
- CR-LF (carriage-return-line-feed) translation, 205
- Cross-reference listing (LIB), 146, 153
- CRT0.OBJ, 32, 117
- CS register, 221, 225
- CSETARGV.OBJ, 33, 134
- Ctype macros, 353
- Customized memory models, 185
- CVARSTCK.OBJ, 34
- /D option, 71
 - MAKE, 162, 285
- Dash (-), as MSC option character, 60
- Data
 - loading, 121
 - passed at execution, 131
 - portability, 359
 - segment, 188, 216
 - default, 206, 215
 - default name, 207
 - naming, 207
 - restrictions, 93
 - sharing, in mixed-language programming, 262
 - threshold, setting, 206
 - types, mixed-language programming, 243
 - types, size of, 349
- Data files
 - binary, 33
 - text, 33
- Data pointers, 186
 - _DATA segment, 207, 215
- Data threshold, setting, 206
- Debugging, preparation for, 89

- Declaration
 functions with near and far, 181
 procedures in mixed-language
 programming, 241
- Declarations, maximum level of
 nesting, 409
- Default
 data segment, 215
 file extensions
 LINK, 98
 MSC, 53
 libraries, 32, 100, 106
 changing, 107
 ignoring, 116
 linking, order of, 199
 overriding, 116, 198
 search path, 100, 106
 suppression of, 195, 199
- parameters, passing, 231
- responses
 LIB, 150
 LINK, 101
 MSC, 57
 overriding MSC, 53
- defined(identifier) constant expression, 328
- Definition of constants and macros, 71
- Denormal, 369
- Denormal numbers, 82
- Description file, 159
- Device names, 54
- DGROUP group, 120, 216
- DI register, 221, 224, 225, 337
- Differences from previous versions of
 the compiler. *See* Assembly-
 language interface; Compiler
 differences
- Direction flag, 225, 337
- Disk
 backing up, 17
 contents, 18, 40
 swapping, 57, 112
- Documentation, compiler, 4
- Dollar sign (\$), 325
- /DOSSEG option, 120
- DS (data segment) register, 216, 221,
 225, 336, 343
- DS. *See* Data, segment
- /DSALLOCATE option, 121
- /E option, 74
- # elif directive, 328
- Ellipsis dots, use of, 10
- EM.LIB, 31, 198
- EMOEM.ASM, 34, 82
- Emulator, 79, 198
 in-line instructions, 81
 library, 32
- Entry sequence, 221, 338
- enum type, in previous versions of the
 compiler, 326
- environ variable, 133
- Environment table, 133
 size, maximum, 371
 suppression of processing, 135
- Environment
 batch files, setting up with, 46
 changing, 38
 enlargement, 307
 portability, 358
 setting up, 34
 variable names, notational
 conventions, 10
- variables, 17, 34, 35, 273
 defining, 36
 INCLUDE, 35
 LIB, 35
 overriding, 38
 PATH, 35
 TMP, 35
- envp variable, 132
- /EP option, 74
- Equality operators, in previous versions
 of the compiler, 327
- Error messages, 365
 compilation, 86, 371
 compiler, 85, 371
 command line, 86, 372, 404
 fatal, 86, 371, 382
 internal, 86, 372
 warning, 86, 88, 371, 373
- EXEMOD, 424, 425
- EXEPACK, 423
 floating-point exceptions, 368
- LIB, 417
- LINK, 410
- MAKE, 421
- mixed-language programming, used
 in, 265
- run time, 365
- Errorlevel codes. *See* Exit codes

- ES register, 217, 336, 343
- Evaluation order, 357
- Exception, 369
- Exclude option, 76
- Executable
 - files, 18, 30, 131
 - command-line arguments, 131
 - compression, 303
 - modification of, 304
 - naming, 99, 293
 - packing, 113
 - passing data to, 131
 - search path, 34, 35
 - image, 123
- Execution of programs, 131
- Execution-time optimization, 91
- EXEMOD, 304
 - error messages, 424
 - /H option, 315
 - /MAX option, 305
 - /MIN option, 305
 - /STACK option, 305
 - summary, 287
- EXEMOD.EXE, 30
- EXEPACK, 303
 - error messages, 423
 - stripping symbolic debug information, 113, 304
 - summary, 286
- /EXEPACK option, 113
- EXEPACK.EXE, 30
- Exit code, 47, 311
- Exit codes
 - CodeView, 313
 - EXEMOD, 314
 - EXEPACK, 314
- Exit sequence, 224, 338
- Extensions, default
 - .LIB, 144
 - LINK, 98
 - MSC, 53
- External names, 194
- /F option, 296
 - arguments, 293
 - CL command, used in, 292
- /Fa option, 64, 68, 222, 226, 228
- far keyword, 177, 193
 - declaration of functions, used in, 181
 - library routines, used with, 179
- Far pointers, 177
- FAR_BSS, 216
- FAR_DATA class, 216
- Fatal error messages, 86, 371, 382
- /Fc option, 64, 68, 222, 226, 228
- /Fe option, 292, 296
- File extensions. *See* Extensions
- File-name conventions, 53
 - LINK, 98
- File names
 - notational conventions, 10
 - special, 54
- Files, batch. *See* Batch files
- Files, data. *See* Data files
- Files, executable. *See* Executable, files
- Files, include. *See* Include files
- Files, library. *See* Library files
- Files
 - CONFIG.SYS file, 38
 - library, 31
 - locating, 34
 - mixed-language programming, used in, 264
 - number open, maximum, 371
 - organization
 - floppy-disk system, 25
 - hard-disk system, 22
 - other, 21
 - size, maximum, 371
- Files, temporary. *See* Temporary files
- Fix-ups, 126
- /Fl option, 64
- Floating point not loaded, 81, 367
- Floating point
 - libraries, 32, 80, 82
 - operations, 80, 198
 - compatibility, 83
 - default, 81
 - efficiency, maximum, with coprocessor, 80
 - efficiency, maximum, without coprocessor, 82
 - error messages, 368
 - flexibility, maximum, 83
 - floating-point exceptions, 368
 - function calls, 80, 82
 - in-line instructions, 80, 81
 - precision, maximum, with coprocessor, 80
 - precision, maximum, without coprocessor, 81

- Floating point (*continued*)
 - options, 79
 - /Fm option, 242, 296
 - /Fo option, 63
 - fopen function, 333
 - fortran keyword, 193, 221
 - Forward slash (/)
 - LINK option character, 111
 - MSC option character, 60
 - /FPa option, 79, 82, 198
 - /FPc option, 79, 82, 198
 - /FPc87 option, 79, 80, 198
 - /FPi option, 79, 81, 198
 - /FPi87 option, 79, 80, 198
 - Frame number, 124
 - Frame pointer, 338
 - freopen function, 333
 - /Fs option, 64, 292
 - Function identifiers, in previous versions of the compiler, 326
 - Function declarations, generation of, 77
 - Functions, with variable number of arguments, 357
 - Functions, declaration with near and far, 181
- /G0 option, 84
- /G1 option, 84
- /G2 option, 84
- /Gc option, 221
- getenv, 133
- getmem routine, 330
- Global symbols. *See* Public symbols
- Global variables, naming conventions, 341
- Groups, 125, 216
 - DGROUP, 120, 216
 - previous versions of the compiler, 344
- /Gs option, 201
- /Gt option, 206
- /Gw option, Windows applications, 209
- /H option, 194, 305
- Hancock, Les, 11
- Heap, 214
- /HELP option, 112
- /HIGH option, 122
- Huge arrays, in mixed-language programming, 256
- huge keyword, 177, 193
- library routines, used with, 179
- Huge-model library files, 31
- /I option, 75, 162, 285
- Identifier length, 355
- Identifiers
 - length, maximum, 409
 - notational conventions, 10
 - predefined, 72
 - removal of definitions, 73
 - previous versions of the compiler, 325, 326
- IF ERRORLEVEL command, 47, 88
- # include directive, 31
- Include files, 18, 31
 - directory specification, 75
 - nesting, maximum level of, 409
 - portability problems, 348
 - search path, 35, 75, 76
 - v2tov3.h, 330
- INCLUDE variable, 35, 75, 76
- Inference rules, 165
- Infinities, 82
- Input, and mixed-language programming, 264
- Installation of compiler software, 17
- Instruction set
 - 80186/80188 processors, 84
 - 80286 processor, 84
 - 8086/8088 processors, 84
- Instructions, in-line, 80, 81
- Integers
 - mixed-language programming, used in, 244
 - summary of sizes, 278
- INTERFACE statement
 - mixed-language programming, used in, 238
- Interfaces, with other languages, 213
- iscsym, 333
- iscsymf macros, 333
- Italics, 10
- /J option, 196

Index

Kernighan, Brian W., 11
Key sequences, notational conventions, 11
Keywords
 calling conventions, specification of, 231
 fortran, 221
 mixed-language programming, used in, 226
 pascal, 221
 special, 193
Kilobyte, 92
Kochan, Stephen, 11
Krieger, Morris, 11

Language-definition differences, 325
Large model, 176
 compilation, in mixed-language programming, 230
 library files, 31
Large programs, 92
Learning to Program in C, 11
Length, of identifiers, 355
.LIB, 144
LIB
 backup library file, 143
 command characters, summary, 280
 error messages, 417
 Exit codes, 314
 modification methods, 143
 operations, order of, 142
 /PAGESIZE option, 154
 summary, 283
 termination, 150
 variable, 35, 100, 106
LIB command
 command symbols, 145
 command-line method, 147
 default responses, 150
 library module
 addition, 145, 151
 deletion, 145, 152
 extraction, 146, 153
 extraction and deletion, 146, 153
 replacement, 145, 152
 line extension, 145
 prompts, 143
 response-file method, 148
 termination, 150
LIB.EXE, 30

Libraries
 8087/80287, 198
 alternate math, 82, 198
 changing, at link time, 198
 combining, 145, 151, 153
 creation, 141, 150, 196
 default, 100, 106, 107, 116
 emulator, 198
 floating point, 80
 mixed-model programs, 189
 modification, 141, 151
 page size, 154
 prompt, 100
 search path, 100, 106
Library file does not exist. Create?, 151
Library files, 20
 compact model, 31
 default, 32
 floating point, 31
 huge model, 31
 large model, 31
 medium model, 31
 notational convention, 33
 search path, 35
 small model, 31
 standard C, 32
Library functions, changes to syntax, 324
Library listing, 146, 153
Library manager, 30
 command line, 147
 response file, 148
 response to prompts, 143
Library manager utility. *See LIB*
Library name prompt, 144
Library search path, 100, 106
Library support, for near, far, and huge, 179
Limits
 compiler, 409
 run time, 370
Line extension, 101, 145, 149
Line-number option, 89
Line numbers, display in linker listing file, 114
/LINENUMBERS option, 114
LINK
 align type, 123
 C files, used with, 105
 error messages, 410
 exit codes, 313

- LINK (*continued*)
 groups, 125
 operation, 123
 options
 /EXEPACK, 113
 /HELP, 112
 separation of entries, 101
 starting, 97
 temporary output file, 105
 termination, 102
- LINK command
 default responses, 101
 line extension, 101
 prompts, 97
 separation of entries, 101
 termination, 102
- LINK command characters
 summary, 280
- link option, 295, 296
- LINK options, 111
 abbreviations, 111
 arguments, numerical, 112
 case sensitivity, 116
 /CO, 115
 compatibility preservation, 122
 control of
 data loading, 122
 run-file loading, 122
 segments, 119
 stack size, 117
 /CPARMAXALLOC, 118
 debugging, 115
 default libraries, ignoring, 116
 /DOSSEG, 120
 /DSALLOCATE, 121
 /HIGH, 122
 line number, displaying, 114
 /LINENUMBERS, 114
 /MAP, 114
 map file, 114
 /NOD (no default library search), 84,
 116, 198, 199
 /NOG (no group association), 122
 /NOI (no ignore case), 116
 overlay interrupt setting, 120
 /OVERLAYINTERRUPT, 120
 paragraph space allocation, 118
 /PAUSE, 112
 pausing, 112
 segment order, 120
 /SEGMENTS, 119
- LINK options (*continued*)
 /STACK, 117
 summary, 280
 unsuitable for use with C programs, 107
- Linker summary, 280
- Linker utility. *See* LINK
- Linker, 30
 command line, 102
 response, 97
 response file, 103
- LINK.EXE, 30, 34, 35
- Linking
 C program files, used with, 105
 CL command, use in, 294
 default libraries, overriding, 198
 mixed-language programming, used with, 264
- LINT_ARGS, 78
- List File prompt, 99
- List file prompt, 146, 153
- Listing file
 assembly listing, 64, 68
 combined listing, 68
 LIB, 142, 146, 153
 LINK, 99, 107, 114
 object listing, 64
 preprocessed listing, 74
 source listing, 64
- LLIBC.LIB, 33
- LLIBFA.LIB, 82, 198
- LLIBFP.LIB, 33, 80, 198
- Logical AND and OR operators, in previous versions of the compiler, 329
- Logical values, in mixed-language programming, 261
- Long addresses, parameters passed by, 235
- Long pointers. *See* Far pointers
- LSETARGV.OBJ, 33, 134
- Lvalue expressions, in previous versions of the compiler, 327
- LVARSTCK.OBJ, 34
- Macro definitions
 MAKE, 163
 size, maximum, 409
- Macros
 arguments, maximum number, 409

- Macros (*continued*)
 character classification, 353
 defined, 71
 notational conventions, 10
main function, 105
 arguments to, 132
MAKE
 dependent file, 160
 described, 159
 description file, 159
 error messages, 421
 example, 167
 exit codes, 88, 311, 314
 inference rules, 165
 invocation, 161
 macro definitions, 163
 macro names, special, 165
 messages, 162
 target file, 160
MAKE description file, 159
MAKE options, 162
 /D, 162, 285
 /I, 162, 285
 /N, 162, 285
 /S, 162, 285
Manifest constants, notational
 conventions, 10
Map file, 114
 naming, 293
/MAP option, 114
max macro, 333
/MAX option, EXEMOD, 305
Medium model, 175
Medium-model library files, 31
Memory allocation, from the stack, 34
Memory model
 compact, 31
 customized, 185
 default, 174
 huge, 31
 large, 31
 medium, 31
 mixed, 177, 185, 186, 188
 mixed-language programming, used
 with, 230
 notational convention for files, 33
 small, 31
Memory-model options, 185
 code-pointer size, 186
 data-pointer size, 186
 segment setup, 188
Memory models, standard, 17, 278
M_I86, 72
M_I86xM, 72
Microsoft C, XENIX version, and
 mixed-language programming, 229
Microsoft LIB. *See LIB*
Microsoft LINK. *See LINK*
min macro, 334
/MIN option, 305
Minimum allocation value, control of,
 305
Minus sign (-), LIB command symbol,
 146, 152
Minus sign-asterisk (-*), LIB command
 symbol, 146, 152
Minus sign-plus sign (-+), LIB
 command symbol, 145, 152
Mixed-language programming
 advantages, 229
 arrays, 256
 attributes, 231
 booleans, 248
 C, from, 242
 calling conventions, 230
 characters, 248
 compilation, 264
 compiler versions required, 229
 complex numbers, 260
 data sharing, 262
 data types, 243
 data types, using tables of, 243
 files, 264
 FORTRAN, from, 238, 240
 input, 264
 integers, 244
 keywords, 231
 linking, 264
 logical values, 261
 memory models, 230
 Microsoft C, XENIX version, used
 with, 229
 output, 264
 parameters, passing, 230
 Pascal, from, 241
 pointers, 254
 procedure arguments, 262
 procedure parameters, 262
 procedure pointers, 262
 real numbers, 248
 records, 259
 return values, 262

- Mixed-language programming
(continued)
- set type, 259
 - stack, use of, 230
 - structs, 259
 - uses, 229
 - writing to the terminal, 264
- Mixed-memory models, 177, 186, 188
- MLIBC.LIB, 33
- MLIBFA.LIB, 82, 198
- MLIBFP.LIB, 33, 80, 198
- Modules, naming, 207
- movmem routine, 334
- MSC command
- command line, partial, 58
 - command line, use of, 57
 - exclude option, 76
 - prompts, response to, 52
 - /w option, 89
 - /X option, 76
- MSC exit codes, 312
- MSC option character (/), 60
- MSC option character (-), 60
- MSC options, 60
- 80186/80188 and 80286 processors, use of, 84
 - /A, 186, 187
 - /AC, 175
 - /AH, 176
 - /AL, 176
 - /AM, 175
 - arguments to, 60
 - /AS, 174
 - assembly listing, 64, 68
 - /C, 75
 - case of, 60
 - char type default, changing, 196
 - combined listing, 68
 - comments, preservation of, 75
 - constants and macros, defining, 71
 - /D option, 71
 - data segments, naming, 207
 - data threshold, setting, 206
 - default-library selection, suppression of, 195
 - /E, 74
 - /EP, 74
 - external names, restricting length of, 194
 - /Fa option, 64, 68
 - /Fc option, 64, 68
- MSC options *(continued)*
- /Fl option, 64
 - floating point, 79, 80, 81, 198
 - /Fo, 63
 - /FPa, 79, 82, 198
 - /FPC, 79, 82, 198
 - /FPC87, 79, 80, 198
 - /FPi, 79, 81, 198
 - /FPi87, 79, 80, 198
 - /Fs option, 64
 - function declaration generation, 77
 - /G0, 84
 - /G1, 84
 - /G2, 84
 - /Gs, 201
 - /Gt, 206
 - /H, 194
 - /I, 75
 - include files, search for, 75
 - /J, 196
 - line numbers, 89
 - listing, 62
 - memory model
 - code-pointer size, 186
 - compact, 175
 - customized, 185
 - data-pointer size, 186
 - huge, 176
 - large, 176
 - medium, 175
 - mixed, 185, 186, 187
 - segments, setting up, 188
 - small, 174 - modules, naming, 207
 - /ND, 207
 - /NM, 207
 - /NT, 207
 - /O, 90
 - object file
 - labeling, 195
 - naming, 63 - /Od, 89
 - optimization, 90, 201, 203
 - /Ox, 203
 - /P, 74
 - predefined identifiers, removing definitions of, 73
 - preprocessed listing, 74
 - preprocessor, 70, 71, 73, 75
 - spaces in, 60
 - special keywords, disabling, 178, 193

- MSC options (*continued*)
 stack probes, removal, 201
 structure members, packing, 193
 summary, 273
 syntax error identification, 77
 text segments, naming, 207
 /U, 73
 /u, 73
 /V, 195
 /W, 88
 /w, 88
 warning level, setting, 88
 /X, 75
 XENIX-compatible, 297
 /Za, 178, 193
 /Zd, 89
 /Zg, 77
 /Zi, 89
 /Zl, 195
 /Zp, 193
 /Zs, 77
MSC prompts, 52
 default responses, 57
MSC.EXE, 30, 34
MSDOS identifier, 72
MS-DOS
 case sensitivity, 41
 program header, 305
MSETARGV.OBJ, 33, 134
MVARSTCK.OBJ, 34
- /N option, MAKE, 162, 285
Names
 executable file, 99
 length, in FORTRAN, 236
 module, 207
 object file, 63
 segment. *See* Segment, names
Naming conventions, 224, 236, 341
 compiler, 69, 109
 previous versions of the compiler, 341
 segments, 208
Naming the executable file, 292
Naming the map file, 293
NANs, 82
/ND option, 207
near keyword, 177, 193
 declaring functions, used in, 181
 library routines, used with, 179
- Near pointers, 177
Nesting
 declarations, 409
 include files, 409
 preprocessor directives, 409
/NM option, 207
NO87 variable, 200
/NOD (no default library search)
 option, 84, 116, 194, 199
/NODEFAULTLIBRARYSEARCH
 option, 116
NO_EXT_KEYS, 73
/NOGROUPASSOCIATION option, 122
/NOI (no ignore case) option, 116
/NOIGNORECASE option, 116
Notational conventions, 9
/NT option, 207
NUL, 54
NUL., 146, 153
Null-pointer assignment, 137, 366
NULL segment, 137, 215, 366
 _nullcheck, 138
NUL.MAP, 99
- /O option, 90
O_BINARY, 332
Object file name prompt, 55
Object file, 141
 labeling, 195
 naming, 63
Object-listing file, 64
Object listing prompt, 56
Object module, 141
 copying from a library, 152
 deletion from a library, 145, 152
 extraction and deletion from a library, 146, 153
 inclusion in a library, 145, 151
Object Modules prompt, 98
/Od option, 89
Offset, parameters passed by, 235
open function, 334
Operations prompt, 144
Operators
 logical AND and OR, 329
 previous versions of the compiler
 equality, 327
 relational, 327
Optimization, 90

Optimization (*continued*)
 advanced, 201
 alias checking, relaxation of, 91
 code size, 91
 default, 90
 disabling, 91
 execution time, 91
 maximum, 203
 options, 90, 203
 stack probes, removal, 201
 Optimizing. *See* Optimization
 Option character (/), 111
 Optional fields, notational conventions, 11
 Options
See also MSC options
 LINK. *See* LINK options
 MSC, 60
 arguments to, 60
 case of, 60
 spaces in, 60
 summary, 273
 O_RAW, 332
 Order of evaluation, 357
 Output, and mixed-language programming, 264
 Output library prompt, 146
 Overlay manager prompts, 110
 /OVERLAYINTERRUPT option, 120
 Overlays, 109
 interrupt number, setting, 120
 Overview, 3
 /Ox option, 203

 /P option, 74
 /PAGESIZE option, 154
 Paragraph space, 118
 Parameters
 default calling conventions, 231
 long address, passed by, 235
 procedure, in mixed-language programming, 262
 reference, passed by, 230, 231
 short address, passed by, 235
 value, passed by, 230, 231, 234
 variable number of, 230
 variable numbers, passed by, 236
 Partial command line, MSC, 58
 pascal keyword, 193, 221
 Passing data at execution, 131

PATH command, 17, 35, 36, 37, 166
 AUTOEXEC.BAT file, used in, 37
 batch files, used in, 38
 Path names
 portability problems, 348
 PATH variable, 35, 36, 131
 /PAUSE option, 112
 peek routine, 330
 Plum, Thomas, 11
 Plus sign (+)
 LIB command symbol, 145, 151
 LINK command, used in, 101
 Pointer
 arguments, size conversion, 183
 manipulation, 352
 Pointers
 code. *See* Code pointers
 data. *See* Data pointers
 far. *See* Far pointers
 mixed-language programming, 255
 near. *See* Near pointers
 procedure, in mixed-language programming, 262
 summary of sizes, 278
 poke routine, 330
 Portability, 348
 address space, 353
 bit fields, 351
 byte length, 349
 byte order, 350, 360
 case distinction, 355
 character set, 353
 data, 359
 environment, 358
 evaluation order, 357
 functions, with variable number of arguments, 357
 identifier length, 355
 pointer manipulation, 352
 problems, hardware, 348
 problems, include files, 348
 problems, path names, 348
 register variables, 355
 shift operations, 354
 side effects, 357
 sign extension, 354
 signed and unsigned char, 354
 size of data types, 349
 storage alignment, 349
 type conversion, 356
 word length, 349

Index

- Practice session, 41
- Pragmas, check_stack, 201
- Preprocessor
 - defined(identifier) constant expression, 328
 - # elif directive, 328
 - macro arguments, maximum number of, 409
 - macro definition, maximum size of, 409
 - nesting, maximum level of, 409
 - options, 70
 - comments, preservation, 75
 - /D, 71
 - predefined identifiers, removing definitions of, 73
 - previous versions of the compiler, 328
- PRN, 54
- Procedure pointers
 - mixed-language programming, 255, 262
- Processor
 - 80186/80188 versions, 40, 84
 - 80286 version, 40, 84
 - 8086/8088 versions, 84
 - 8087/80287 versions, 38
- Product names, notational conventions, 11
- Program fragments, notational conventions, 10
- Program header, inspection of, 305
- Program maintainer. *See* MAKE
- Programming examples, notational conventions, 10
- Prompts
 - MSC, 52
 - notational conventions, 11
- Public names. *See* External names
- Public symbols, 114
- putenv, 133

- Question mark (?)
 - CL command, use in, 292
 - wild-card character, 33, 134
- Quick setup
 - floppy disk, 25
 - hard disk, 22
- Quotation marks, use of, 11

- rbrk routine, 330
- README.DOC file, 34
- Real numbers, in mixed-language programming, 248
- Records, in mixed-language programming, 259
- Register
 - usage conventions, in previous versions of the compiler, 336 variables, 337, 355
- Registers, 225
 - BP, 221, 224, 338
 - CS, 221, 225
 - DI, 221, 224, 225, 337
 - DS, 216, 221, 225, 336, 343
 - ES, 217, 336, 343
 - SI, 221, 224, 225, 336
 - SP, 340
 - SS, 216, 221, 225, 337
- Relational operators, in previous versions of the compiler, 327
- Relocation information, 123
- repmem routine, 330
- Response file
 - LIB, 148
 - LINK, 105
- Return codes. *See* Exit codes
- Return-value conventions, 222
- Return values, in mixed-language programming, 262
- Ritchie, Dennis M., 11
- rlsmem routine, 330
- rstmem routine, 330
- Run file
 - See also* Executable files
 - loading, 122
 - prompt, 99
- Run time
 - error messages, 365
 - libraries, 141
 - library differences, 330
 - abs, 331
 - allmem, 330
 - creat, 331
 - fopen, 333
 - freopen, 333
 - getmem, 330
 - iscsym, 333
 - iscsymf, 333
 - max, 333
 - min, 334

- library differences (*continued*)

 movmem, 334

 open, 334

 peek, 330

 poke, 330

 rbrk, 330

 repmem, 330

 rlsmem, 330

 rstmem, 330

 setmem, 335

 setnbuf, 335

 sizmem, 330

 stcarg, 330

 stccpy, 330

 stcd_i, 330

 stch_i, 330

 stci_d, 330

 stcis, 335

 stcisin, 335

 stclen, 335

 stcpam, 330

 stcpm, 330

 stcu_d, 330

 stpblk, 331

 stpbrk, 335

 stpchr, 335

 stpsym, 331

 stptok, 331

 stscmp, 335

 stspfp, 330

 v2tov3.h, 330
- Running programs, 131
- /S option, MAKE, 162, 285
- Sample floppy-disk setup, 40
- Sample hard-disk setup, 35
- Sample setup, 40
- Schustack, Steve, 11
- Search path

 See also Standard search paths

 changing, 38

 files

 executable, 34, 35

 include, 35, 75

 library, 35

 libraries, 100, 106
- Segment

 model, 188, 213, 342

 names, 203, 279, 344

 previous versions of the compiler, 342
- Segment (*continued*)

 naming conventions, 208

 order, 120

/SEGMENTS option, 119
- Segments, 214

 align type, 217

 _BSS, 215

 _c_common, 215

 class name, 217

 combine class, 217

 combining, 125

 CONST, 215

 _DATA, 215

 data, 188, 216

 data threshold, setting, 206

 default name, 207

 names, 209

 NULL, 137, 215, 366

 number allowed, 119

 setting up, 188

 stack, 188

 STACK, 215

 _TEXT, 216

 text, 216

 default name, 207

 naming, 207
- Semicolon (;) in

 LIB command, 144, 147, 150

 LINK command, 101

 MSC command, 57
- SET command, 17, 35, 36

 AUTOEXEC.BAT file, 37

 batch files, 38
- Set type (Pascal), in mixed-language programming, 259
- _setargv, 135
- SETENV utility, 307

 error messages, 425

 exit codes, 315

 _setenvp, 135
- setmem routine, 335
- setnbuf routine, 335
- Shift operations, 354
- Short addresses, parameters passed by, 235
- Short pointers

 See also Near pointers

 conversion to long integers, 322
- SI register, 221, 224, 225, 337
- Side effects, 357
- Sign extension, 354

- Signed char, 354
Size of data types, 349
sizmem routine, 330
SLIBC.LIB, 32
SLIBFA.LIB, 32, 82, 198
SLIBFP.LIB, 32, 80, 198
Small capitals, use of, 11
Small model, 174
Small-model library files, 31
Source file name prompt, 55
Source-listing file, 64
Source listing prompt, 55
SP register, 340
Spaces, in MSC options, 60
Special file names, 54
Special keywords, 178, 193
 disabling, 178, 193
Special macro names, MAKE, 165
SS. *See*, Stack segment
SS register, 221, 225, 336
SS (stack segment) register, 216
SSETARGV.OBJ, 33, 134
Stack
 memory allocation from, 34
 mixed-language programming, used
 in, 230
 order, 219
 overflow, 366
 probes, 201
 segment, 188
 setup, in previous versions of the
 compiler, 338
 size
 control, 305
 default for C programs, 117
 setting, 117
Stack checking, in previous versions of
 the compiler, 339
STACK class name, 120
/STACK option, 117
/STACK option, EXEMOD, 305
STACK segment, 215
Standard C library, 32
Standard memory models,
 summary, 278
Standard places, 35
 changing, 76
 ignoring, 76
 libraries, 100, 106
Standard search paths, 35
Start-up routine, 32, 105, 117

Statements
 INTERFACE, in mixed-language
 programming, 238
 WRITE, in mixed-language
 programming, 264
stcarg routine, 330
stccpy routine, 330
stcd_i routine, 330
stch_i routine, 330
stci_d routine, 330
stcis routine, 335
stcisin routine, 335
stclen routine, 335
stcpam routine, 330
stcpdm routine, 330
stcu_d routine, 330
STDARGV, 134
Storage alignment, 349
stpblk routine, 331
stpbrk routine, 335
stpchr routine, 335
stpsym routine, 331
stptok routine, 331
Strings
 length, maximum, 409
 notational conventions, 11
 previous versions of the compiler,
 326
Structs, in mixed-language
 programming, 259
Structure identifiers, in previous
 versions of the compiler, 327
Structures
 packing, 193
 previous versions of the compiler,
 328
stscmp routine, 335
stspfp routine, 331
Super arrays, in mixed-language
 programming, 256
Suppressing command-line processing,
 135
Suppressing null-pointer checks, 137
Suppressing processing of environment
 table, 135
SVARSTCK.OBJ, 34
Swapping disks, 57
Switches. *See* Options
Syntax checking, 76
Syntax conventions. *See* Notational
 conventions

Syntax errors, 77
 SYS subdirectory, 31
 System-level definitions, 31

Target/dependent descriptions, 159
 Temporary files, 35, 409
 Terminal, writing to, in mixed-language programming, 264
 _TEXT, 207
 Text mode, 33, 205
 _TEXT segment, 216
 Text segment
 default name, 207
 naming, 207
 Text segments, 216
 TMP variable, 35
 TOOLS.INI file, 166
 Type casts, in previous versions of the compiler, 327
 Type checking, 78
 Type conversion, 356

/U option, 73
 /u option, 73
 Underflow, 370
 Underscore (_) in
 global names, 341
 identifiers, 224
 significance, in names, 237
 Uninitialized variables, in previous versions of the compiler, 327
 Union identifiers, in previous versions of the compiler, 327
 Unions, in previous versions of the compiler, 328
 Unsigned char, 354
 Unsigned char type, 196
 unsigned long type, in previous versions of the compiler, 326
 Uppercase letters, use of, 10
 User's Guide, organization, 4
 Utilities
 EXEMOD. *See* EXEMOD
 EXEMOD. *See* EXEMOD.EXE
 EXEPACK. *See* EXEPACK
 EXEPACK. *See* EXEPACK.EXE
 LIB. *See* LIB.EXE
 library manager. *See* LIB
 LINK. *See* LINK.EXE

Utilities (*continued*)
 linker. *See* LINK

/V option, 195
 Value
 parameters passed by, 234
 passing parameters by, 230, 231
 Variable numbers of parameters, passing, 236
 Variables
 communal, 327
 environment, 34
 global, naming conventions, 341
 register, 336
 uninitialized, in previous versions of the compiler, 327
 Variables, environment. *See* Environment, variables
 Variables, register, 355
 Variations in C, 11
 VM.TMP, 104

/W option, 88
 /w option, 88
 Warning error messages, 371, 373
 Warning level option, 88
 Warning levels, 88
 Warning messages, 86, 88
 WARNING: NO STACK SEGMENT, 117
 Wild-card arguments, 134
 Wild-card characters, 33
 in CL command, 292
 Windows applications
 /Aw option, 209
 /Gw option, 209
 Word length, 349
 WRITE statement, and mixed-language programming, 264

/X option, 75
 XENIX, case sensitivity, 41
 XENIX-compatible options, 297

/Za option, 178, 193
 /Zd option, 89
 /Zg option, 77

Index

- /Zi option, 89
- /Zl option, 195
- /Zp option, 193
- /Zs option, 77



16011 NE 36th Way, Box 97017, Redmond, WA 98073-9717

Software Problem Report

Name _____

Street _____

City _____ State _____ Zip _____

Phone _____ Date _____

Instructions

Use this form to report software bugs, documentation errors, or suggested enhancements. Mail the form to Microsoft.

Category

Software Problem

Documentation Problem

(Document #_____)

Software Enhancement

Other

Software Description

Microsoft Product _____

Rev. _____ Registration # _____

Operating System _____

Rev. _____ Supplier _____

Other Software Used _____

Rev. _____ Supplier _____

Hardware Description

Manufacturer _____ CPU _____ Memory _____ KB

Disk Size _____" Density: Sides:

Single _____ Single _____

Double _____ Double _____

Peripherals _____

Problem Description

Describe the problem. (Also describe how to reproduce it, and your diagnosis and suggested correction.) Attach a listing if available.

Microsoft Use Only

Tech Support _____

Date Received _____

Routing Code _____

Date Resolved _____

Report Number _____

Action Taken: