Microsoft® FORTRAN

Microsoft CodeView® and Utilities User's Guide

Microsoft®

#828/1

Microsoft® CodeView® and Utilities

USER'S GUIDE

VERSION 2.3

FOR MS® OS/2 AND MS-DOS®
OPERATING SYSTEMS

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 the software on any medium except as specifically allowed in the license or nondisclosure agreement. No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose without the express written permission of Microsoft.

©Copyright Microsoft Corporation, 1987, 1989. All rights reserved.
Simultaneously published in the U.S. and Canada.

Printed and bound in the United States of America.

Microsoft, MS, MS-DOS, XENIX, and CodeView are registered trademarks of Microsoft Corporation.

AT&T is a registered trademark of American Telephone and Telegraph Company.

Eagle is a registered trademark of Eagle Computer, Inc.

IBM is a registered trademark of International Business Machines Corporation.

Intel is a registered trademark of Intel Corporation.

Lotus is a registered trademark of Lotus Development Corporation.

Tandy is a registered trademark of Tandy Corporation.

Document No. LN0801A-500-R00-0889
Part No. 07824
10 9 8 7 6 5 4 3 2 1

Table of Contents Overview

Introduction xxiii

Part One The CodeView Debugger

Chapter 1 Getting Started 5
Chapter 2 The CodeView Display 33
Chapter 3 Using Dialog Commands 61
Chapter 4 CodeView Expressions 65
Chapter 5 Executing Code 87
Chapter 6 Examining Data and Expressions 99
Chapter 7 Managing Breakpoints 133
Chapter 8 Managing Watch Statements 141
Chapter 9 Examining Code 159
Chapter 10 Modifying Code or Data 171
Chapter 11 CodeView Control Commands 191
Chapter 12 Debugging in Protected Mode 209

Part Two Utilities

Chapter 13 Linking Object Files with LINK 225
Chapter 14 Incremental Linking with ILINK 261
Chapter 15 Managing Libraries with LIB 269
Chapter 16 NMAKE 285
Chapter 17 Using Other Utilities 307
Chapter 18 Linking for Windows and OS/2 Systems 315
Chapter 19 Using Module-Definition Files 321
Chapter 20 Creating Dual-Mode Programs with BIND 341
Chapter 21 Using EXEHDR 347

Appendixes

Appendix A Regular Expressions 353
Appendix B Using Exit Codes 357
Appendix C Error Messages 361

Glossary 415

Index 427

Table of Contents

#823/1

Introduction

New Features of the CodeView® Debugger xxii
About this Manual xxiii
Document Conventions xxv

PART 1 The CodeView Debugger

Chapter 1 Getting Started 5
1.1 Restrictions 5
1.2 The CodeView Environment 6
1.3 Preparing Programs for the CodeView Debugger 6
1.3.1 Programming Considerations 7
1.3.2 CodeView Compile Options 8
1.3.3 CodeView Link Options 8
1.3.4 Preparing C Programs 9
1.3.5 Preparing FORTRAN Programs 11
1.3.6 Preparing BASIC Programs 12
1.3.7 Preparing Pascal Programs 13
1.3.8 Preparing Assembly Programs 14
1.4 Starting the CodeView Debugger 17
1.5 Using CodeView Options 20
1.5.1 Using Two Video Adapters 22
1.5.2 Using the Enhanced Graphics Adapter's 43-Line Mode 23
1.5.3 Using 50-Line Mode 23
1.5.4 Starting with a Black-and-White Display 24
1.5.5 Specifying Start-Up Commands 24

1.5.6 Handling Interrupt Trapping (DOS Only) . . . 25
1.5.7 Using Expanded Memory (DOS Only) . . . 26
1.5.8 Setting the Screen-Exchange Mode (DOS Only) . . . 26
1.5.9 Loading Information from Dynamic-Link Libraries (OS/2 Only) . . . 28
1.5.10 Turning Off the Mouse . . . 29
1.5.11 Debugging Multiple Processes (OS/2 only) . . . 29
1.5.12 Extending EGA Compatibility . . . 29
1.5.13 Using Debug Registers (386 Only) . . . 30
1.5.14 Enabling Window or Sequential Mode . . . 30
1.6 Debugging Large Programs . . . 31
1.7 Working with Older Versions of the Assembler . . . 31

Chapter 2 The CodeView Display . . . 33
2.1 Using Window Mode . . . 33
2.1.1 Executing Window Commands with the Keyboard . . . 35
2.1.2 Executing Window Commands with the Mouse . . . 41
2.1.3 Using Menu Selections . . . 46
2.1.4 Using On-Line Help . . . 58
2.2 Using Sequential Mode . . . 58

Chapter 3 Using Dialog Commands . . . 61
3.1 Entering Commands and Arguments . . . 61
3.1.1 Using Special Keys . . . 62
3.1.2 Using the Command Buffer . . . 62
3.2 Format for CodeView Commands and Arguments . . . 63
3.3 Selecting Text for Use with Commands . . . 64

Chapter 4 CodeView Expressions 65

4.1 C Expressions 66

4.1.1 C Symbols 67

4.1.2 C Constants 67

4.1.3 C Strings 68

4.2 FORTRAN Expressions 69

4.2.1 FORTRAN Symbols 70

4.2.2 FORTRAN Constants 70

4.2.3 FORTRAN Strings 71

4.2.4 FORTRAN Intrinsic Functions 72

4.3 BASIC Expressions 73

4.3.1 BASIC Symbols 74

4.3.2 BASIC Constants 75

4.3.3 BASIC Strings 76

4.3.4 BASIC Intrinsic Functions 76

4.4 Assembly Expressions 77

4.5 Line Numbers 79

4.6 Registers and Addresses 80

4.6.1 Registers 80

4.6.2 Addresses 81

4.6.3 Address Ranges 82

4.7 Memory Operators 83

4.7.1 Accessing Bytes (BY) 83

4.7.2 Accessing Words (WO) 84

4.7.3 Accessing Double Words (DW) 85

4.8 Switching Expression Evaluators 85

Chapter 5 Executing Code 87

5.1 Window and Sequential Mode Commands 87

5.2 Trace Command 88

5.3 Program Step Command 90

5.4 Go Command 92

5.5 Execute Command 95

5.6 Restart Command 96

Chapter 6 Examining Data and Expressions 99

6.1 Display Expression Command 100

6.2 The Graphic Display Command 107

6.2.1 Invoking the Graphic Display Command 108

6.2.2 Changing the Display 109

6.3 Examine Symbols Command 111

6.4 Dump Commands 117

6.4.1 Dump 118

6.4.2 Dump Bytes 119

6.4.3 Dump ASCII 119

6.4.4 Dump Integers 120

6.4.5 Dump Unsigned Integers 121

6.4.6 Dump Words 121

6.4.7 Dump Double Words 122

6.4.8 Dump Short Reals 122

6.4.9 Dump Long Reals 123

6.4.10 Dump 10-Byte Reals 124

6.5 Compare Memory Command 125

6.6 Search Memory Command 126

6.7 Port Input Command 127

6.8 Register Command 128

6.9 8087 Command 129

Chapter 7 Managing Breakpoints 133

7.1 Breakpoint Set Command 133

7.2 Breakpoint Clear Command 136

7.3 Breakpoint Disable Command 137

7.4 Breakpoint Enable Command 138

7.5 Breakpoint List Command 139

Chapter 8 Managing Watch Statements 141

8.1 Watch Statement Commands 141

8.2 Setting Watch-Expression and Watch-Memory Statements 142

8.3 Setting Watchpoints 146

8.4 Setting Tracepoints 148

8.5 Deleting Watch Statements 152

8.6 Listing Watchpoints and Tracepoints 154

8.7 C Examples 155

8.8 FORTRAN Examples 156

8.9 Assembly Examples 156

Chapter 9 Examining Code 159

9.1 Set Mode Command 159

9.2 Unassemble Command 161

9.3 View Command 163

9.4 Current Location Command 165

9.5 Stack Trace Command 166

Chapter 10 Modifying Code or Data 171

10.1 Assemble Command 171

10.2 Enter Commands 175

10.2.1 Enter Command 178

10.2.2 Enter Bytes Command 178

10.2.3 Enter ASCII Command . . . 179
10.2.4 Enter Integers Command . . . 179
10.2.5 Enter Unsigned Integers Command . . . 180
10.2.6 Enter Words Command . . . 181
10.2.7 Enter Double Words Command . . . 181
10.2.8 Enter Short Reals Command . . . 182
10.2.9 Enter Long Reals Command . . . 183
10.2.10 Enter 10-Byte Reals Command . . . 184
10.3 Fill Memory Command . . . 184
10.4 Move Memory Command . . . 185
10.5 Port Output Command . . . 186
10.6 Register Command . . . 187

Chapter 11 CodeView Control Commands . . . 191

11.1 Help Command . . . 191
11.2 Quit Command . . . 192
11.3 Radix Command . . . 193
11.4 Redraw Command . . . 195
11.5 Screen Exchange Command . . . 195
11.6 Search Command . . . 196
11.7 Shell Escape Command . . . 198
11.8 Tab Set Command . . . 200
11.9 Option Command . . . 201
11.10 Redirection Commands . . . 203
11.10.1 Redirecting CodeView Input . . . 203
11.10.2 Redirecting CodeView Output . . . 204
11.10.3 Redirecting CodeView Input and Output . . . 205
11.10.4 Commands Used with Redirection . . . 205

Chapter 12 Debugging in Protected Mode 209

12.1 Using CodeView in Different Modes 210

12.2 Debugging Dynamic-Link Libraries 210

12.3 Debugging Multiple Processes 211

12.3.1 Viewing Status 212

12.3.2 Switching to a Child Process 213

12.4 Debugging Multiple Threads 213

12.5 The Thread Command 215

12.5.1 Legal Values for Specifier 215

12.5.2 Legal Values for Command 216

12.5.3 Entries to the Thread Command 218

12.5.4 Effect of Threads on CodeView Commands 219

PART 2 Utilities

Chapter 13 Linking Object Files with LINK 225

13.1 Determining Linker Output 225

13.2 Specifying Files for Linking 226

13.2.1 Specifying File Names 227

13.2.2 Linking with the LINK Command Line 228

13.2.3 Linking with the LINK Prompts 230

13.2.4 Linking with a Response File 232

13.2.5 How LINK Searches for Libraries 233

13.2.6 LINK Memory Requirements 235

13.3 Specifying Linker Options 236

13.3.1 Aligning Segment Data (/A) 237

13.3.2 Running in Batch Mode (/BA) 237

13.3.3 Producing a .COM File (/BI) 238

13.3.4 Preparing for Debugging (/CO) 238

13.3.5 Setting the Maximum Allocation Space (/CP) 239
13.3.6 Ordering Segments (/DO) 239
13.3.7 Controlling Data Loading (/DS) 240
13.3.8 Packing Executable Files (/E) 241
13.3.9 Optimizing Far Calls (/F) 241
13.3.10 Viewing the Options List (/HE) 242
13.3.11 Controlling Executable-File Loading (/HI) 243
13.3.12 Preparing for Incremental Linking (/INC) 243
13.3.13 Displaying Linker Process Information (/INF) 243
13.3.14 Including Line Numbers in the Map File (/LI) 244
13.3.15 Listing Public Symbols (/M) 245
13.3.16 Ignoring Default Libraries (/NOD) 245
13.3.17 Ignoring Extended Dictionary (/NOE) 245
13.3.18 Disabling Far-Call Optimization (/NOF) 246
13.3.19 Preserving Compatibility (/NOG) 246
13.3.20 Preserving Case Sensitivity (/NOI) 246
13.3.21 Ordering Segments without Inserting NULL Bytes (/NON) 247
13.3.22 Disabling Segment Packing (/NOP) 247
13.3.23 Setting the Overlay Interrupt (/O) 247
13.3.24 Packing Contiguous Data Segments (/PACKC) 248
13.3.25 Packing Contiguous Data Segments (/PACKD) 249
13.3.26 Padding Code Segments (/PADC) 249
13.3.27 Padding Data Segments (/PADD) 250
13.3.28 Pausing during Linking (/PAU) 251
13.3.29 Specifying User Libraries for Quick Languages (/Q) 251
13.3.30 Setting Maximum Number of Segments (/SE) 252

13.3.31 Controlling Stack Size (/ST) . . . 253
13.3.32 Issuing Fixup Warnings (/W) . . . 253
13.4 Selecting Options with the LINK Environment Variable . . . 254
13.5 Linker Operation . . . 254
13.5.1 Alignment of Segments . . . 255
13.5.2 Frame Number . . . 255
13.5.3 Order of Segments . . . 256
13.5.4 Combined Segments . . . 256
13.5.5 Groups . . . 257
13.5.6 Fixups . . . 257
13.6 Using Overlays . . . 258
13.6.1 Restrictions on Overlays . . . 259
13.6.2 Overlay-Manager Prompts . . . 259

Chapter 14 Incremental Linking with ILINK . . . 261

14.1 Definitions . . . 261
14.2 Guidelines for Using ILINK . . . 262
14.3 The Development Process . . . 263
14.4 Running ILINK . . . 264
14.4.1 Files Required for Using ILINK . . . 264
14.4.2 The ILINK Command Line . . . 265
14.5 How ILINK Works . . . 266
14.6 Incremental Violations . . . 266
14.6.1 Changing Libraries . . . 266
14.6.2 Exceeding Code/Data Padding . . . 267
14.6.3 Moving/Deleting Data Symbols . . . 267
14.6.4 Deleting Code Symbols . . . 267
14.6.5 Changing Segment Definitions . . . 267
14.6.6 Adding CodeView Debugger Information . . . 267

Chapter 15 Managing Libraries with LIB 269

15.1 Managing Libraries 270

15.1.1 Managing Libraries with the LIB Command Line 270

15.1.2 Managing Libraries with the LIB Prompts 276

15.1.3 Managing Libraries with a Response File 277

15.1.4 Terminating the LIB Session 278

15.2 Performing Library-Management Tasks with LIB 278

15.2.1 Creating a Library File 279

15.2.2 Changing a Library File 280

15.2.3 Adding Library Modules 280

15.2.4 Deleting Library Modules 280

15.2.5 Replacing Library Modules 281

15.2.6 Copying Library Modules 281

15.2.7 Moving Library Modules (Extracting) 281

15.2.8 Combining Libraries 281

15.2.9 Creating a Cross-Reference-Listing File 282

15.2.10 Performing Consistency Checks 282

15.2.11 Setting the Library-Page Size 283

Chapter 16 NMAKE 285

16.1 Invoking NMAKE 285

16.1.1 Using a Command Line to Invoke NMAKE 286

16.1.2 Using a Command File to Invoke NMAKE 286

16.2 NMAKE Options 287

16.3 Description Files 288

16.3.1 Description Blocks 289

16.3.2 Macros 293

16.3.3 Inference Rules 297

16.3.4 Directives 300
16.3.5 Pseudotargets 302
16.4 Response-File Generation 304
16.5 Differences between NMAKE and MAKE 305

Chapter 17 Using Other Utilities 307
17.1 Modifying Program Headers with the EXEMOD Utility 307
17.2 Enlarging the DOS Environment with the SETENV Utility 311
17.3 Saving Memory with the CVPACK Utility 312

Chapter 18 Linking for Windows and OS/2 Systems 315
18.1 Dynamic-Link Libraries 315
18.2 Linking without an Import Library 316
18.3 Linking with an Import Library 317
18.4 Why Use Import Libraries? 318
18.5 Advantages of Dynamic Linking 319
18.6 Creating Import Libraries with IMPLIB 320

Chapter 19 Using Module-Definition Files 321
19.1 Module Statements 321
19.2 The NAME Statement 323
19.3 The LIBRARY Statement 324
19.4 The DESCRIPTION Statement 325
19.5 The CODE Statement 325
19.6 The DATA Statement 327
19.7 The SEGMENTS Statement 331
19.8 The STACKSIZE Statement 334
19.9 The EXPORTS Statement 334
19.10 The IMPORTS Statement 335
19.11 The STUB Statement 337

19.12 The HEAPSIZE Statement 337
19.13 The PROTMODE Statement 338
19.14 The OLD Statement 338
19.15 The REALMODE Statement 339
19.16 The EXETYPE Statement 339

Chapter 20 Creating Dual-Mode Programs with BIND 341
20.1 Binding Library Routines 342
20.2 Binding Functions as Protected Mode Only 342
20.3 The BIND Command Line 343
20.4 BIND Operation 344
20.5 Executable-File Layout 344
20.6 How to Build a Dual-Mode Application 345

Chapter 21 Using EXEHDR 347
21.1 The EXEHDR Command Line 347
21.2 EXEHDR Output 348
21.3 Output in Verbose Mode 349

Appendixes

Appendix A Regular Expressions 353
A.1 Special Characters in Regular Expressions 353
A.2 Searching for Special Characters 354
A.3 Using the Period 354
A.4 Using Brackets 354
A.4.1 Using the Dash within Brackets 355
A.4.2 Using the Caret within Brackets 355
A.4.3 Matching Brackets within Brackets 355

A.5 Using the Asterisk 356
A.6 Matching the Start or End of a Line 356

Appendix B Using Exit Codes 357
B.1 Exit Codes with NMAKE 357
B.2 Exit Codes with DOS Batch Files 357
B.3 Exit Codes for Programs 358
B.3.1 LINK Exit Codes 358
B.3.2 LIB Exit Codes 359
B.3.3 NMAKE Exit Codes 359
B.3.4 EXEMOD and SETENV Exit Codes 359
B.3.5 CVPACK Exit Codes 360

Appendix C Error Messages 361
C.1 CodeView Error Messages 361
C.2 LINK Error Messages 372
C.2.1 LINK Fatal Error Messages 372
C.2.2 LINK Nonfatal Error Messages 382
C.2.3 LINK Warning Messages 386
C.3 ILINK Error Messages 391
C.3.1 ILINK Fatal Errors 391
C.3.2 Incremental Violations 396
C.3.3 ILINK Warning Messages 398
C.4 LIB Error Messages 399
C.4.1 Fatal LIB Error Messages 399
C.4.2 Nonfatal LIB Error Messages 403
C.4.3 Warning LIB Error Messages 404
C.5 NMAKE Error Messages 405
C.5.1 Fatal NMAKE Error Messages 405
C.5.2 Warning NMAKE Error Messages 411

C.6 EXEMOD Error Messages 412
C.6.1 Fatal EXEMOD Error Messages 412
C.6.2 Warning EXEMOD Error Messages 413
C.7 SETENV Error Messages 413

Glossary 415

Index 427

Figures

Figure 1.1	CodeView Start-up Screen in Window Mode	19
Figure 2.1	Elements of the CodeView Debugging Screen	34
Figure 2.2	The File Menu	46
Figure 2.3	The View Menu	48
Figure 2.4	The Search Menu	49
Figure 2.5	The Run Menu	51
Figure 2.6	The Watch Menu	52
Figure 2.7	The Options Menu	54
Figure 2.8	The Language Menu	56
Figure 2.9	The Calls Menu	57
Figure 6.1	Viewing Structures with Graphic Display	108
Figure 6.2	Viewing a Simple Variable with Graphic Display	109
Figure 8.1	Watch Statements in the Watch Window	145
Figure 8.2	Watchpoints in the Watch Window	148
Figure 8.3	Tracepoints in the Watch Window	152
Figure 8.4	C Watch Statements	154
Figure 8.5	FORTRAN Watch Statements	155
Figure 8.6	Assembly Watch Statements	157
Figure 12.1	Multiple-Thread Program	214
Figure 18.1	Linking without an Import Library	316
Figure 18.2	Linking with an Import Library	317
Figure 20.1	OS/2 Executable-File Header	345

Tables

Table 1.1	Default Exchange and Display Modes	27
Table 4.1	CodeView C Expression Operators	66
Table 4.2	C Radix Examples	68
Table 4.3	CodeView FORTRAN Operators	69
Table 4.4	FORTRAN Radix Examples	71
Table 4.5	FORTRAN Intrinsic Functions Supported by the CodeView Debugger	73
Table 4.6	CodeView BASIC Operators	73
Table 4.7	BASIC Radix Examples	76
Table 4.8	BASIC Intrinsic Functions Supported by the CodeView Debugger	77
Table 4.9	Registers	81
Table 6.1	CodeView Format Specifiers	100
Table 10.1	Flag-Value Mnemonics	188
Table 16.1	Predefined Inference Rules	299

Introduction

Welcome to the Microsoft® CodeView® debugger and development utilities. These are executable programs that help you develop software written with the Microsoft BASIC, C, FORTRAN, and Pascal compilers, as well as with the Microsoft Macro Assembler.

The Microsoft CodeView debugger is a powerful, window-oriented tool that enables you to track down logical errors in programs; it allows you to analyze a program as the program is actually running. The CodeView debugger will display source code or assembly code, indicate which line is about to be executed, dynamically watch the values of variables (local or global), switch screens to display program output, and perform many other related functions. The debugger can be easily learned and used by assembly and high-level-language programmers alike.

The utilities are important at various stages of software development. After you use a compiler or assembler to produce one or more object files, use LINK to produce an executable file. (When a program is made into an executable file, it is finally in the form that can be loaded and executed by DOS.) In the process of linking, you may use software libraries. The LIB utility enables you to create, examine, and maintain these libraries. The process of compiling and linking can be automated—to a large degree—with the MAKE utility; MAKE keeps track of which source files have been changed, and then executes just the commands necessary to update the program.

Other utilities help you maintain executable files once they have been created. You can use EXEMOD to examine or modify the file's header. The executable-file header indicates stack size, load size, and other important items used by DOS each time it executes the file.

Finally, you can use the SETENV and ERROUT utilities to modify the DOS environment itself.

NOTE *Microsoft documentation uses the term "OS/2" to refer to the OS/2 systems—Microsoft® Operating System/2 (MS® OS/2) and IBM® OS/2. Similarly, the term "DOS" refers to both the MS-DOS® and IBM Personal Computer DOS operating systems. The name of a specific operating system is used when it is necessary to note features that are unique to that system.*

New Features of the CodeView® Debugger

- Structure, pointer, and array display

  A new dialog box supports visual display of structures and arrays. Each member or element is displayed. You can also use this dialog box to examine local variables and nested structures and to trace pointer references.

- Multilanguage expression evaluation

  The CodeView debugger has a built-in language interpreter that can evaluate either C, BASIC, or FORTRAN expressions.

- 386 support

  The CodeView debugger now supports debugging of code written specifically for the 386 processor. You can now decode and assemble 386 instructions, as well as view 386 registers.

- Expanded memory support

  If you have expanded memory, you can substantially reduce the amount of main memory required to debug a program. Many programs that were previously too large can now be run with the CodeView debugger.

- 8087 emulator support

  If you do not have an 8087 coprocessor in your machine, you can link to a Microsoft emulator library and take advantage of the 7 command. The debugger will display pseudo-8087 registers, as if you did have a math coprocessor in your machine.

- Overlays and library modules

  The debugger is now fully compatible with programs that use overlays. You can also debug library modules.

- New commands

  The SYMDEB (symbolic debugger) commands—Compare, Fill, Move, Input, and Output—have been added to the CodeView debugger's repertoire. The Option command provides more power for redirected input and start-up commands.

About this Manual

This manual is intended as a companion volume to Microsoft language manuals. It is not language specific, except where examples are required; and in those cases, examples from several languages are typically given.

The manual is divided into two parts, followed by appendixes: Part 1 (chapters 1-12) explains how to use the CodeView debugger to examine and locate program errors; Part 2 (chapters 13-21) explains how to use each of the utilities, including LINK, ILINK, LIB, NMAKE, EXEMOD, SETENV, and BIND. The appendixes at the end of the manual discuss exit codes and error messages for the CodeView debugger and all the utilities.

The following list indicates where to find different kinds of information in the manual. The list is by no means exhaustive, but is intended to serve as a starting place, particularly for the new user of the CodeView debugger.

Information	Location
Examining and locating program errors	Part 1 (chapters 1-12), "The CodeView Debugger," describes methods to help you track down errors in a program and analyze it while it runs. Exit codes and error messages are discussed in the appendixes at the back of this manual.
Starting a debugging session	Chapter 1, "Getting Started," tells you how to compile and link programs so that you can run them with the debugger. It also explains each CodeView command-line option.
Using the CodeView interface	Chapter 2, "The CodeView Display," describes how to use the CodeView windows, pop-up menus, and the mouse.
Specifying the CodeView commands	Chapter 3, "Using Dialog Commands," presents the general form of commands, while Chapter 4, "CodeView Expressions," describes how to build complex expressions for use in commands.
Controlling execution of your program	Chapter 5, "Executing Code," explains the basics of controlling program execution with the CodeView debugger; Chapter 7, "Managing Breakpoints," explains how to use breakpoints to suspend execution.
Watching the value of variables or expressions	Chapter 6, "Examining Data and Expressions," shows how to display values; Chapter 8, "Managing Watch Statements," shows how to place variables in a window where you can watch their values change as the program runs.

Using the utilities	Part 2 (chapters 13-21), "Utilities," describes the various utilities for producing and maintaining executable files, and for other tasks. Exit codes and error messages for the utilities are discussed in the appendixes at the back of this manual.
Creating executable files	Chapter 13, "Linking Object Files with LINK."
Using the incremental linker for faster linking	Chapter 14, "Incremental Linking with ILINK."
Managing software libraries	Chapter 15, "Managing Libraries with LIB."
Automating projects that have several modules	Chapter 16, "NMAKE."
Using EXEMOD, SETENV, and CVPACK	Chapter 17, "Using Other Utilities."
Dynamic linking under OS/2	Chapter 18, "Linking for Windows and OS/2 Systems."
Module-definition statements	Chapter 19, "Using Module-Definition Files."
Binding programs to run under both protected mode and real mode	Chapter 20, "Creating Dual-Mode Programs with BIND."
Viewing the contents of a segmented .EXE file header	Chapter 21, "Using EXEHDR."
Specifying expressions for the CodeView Search command	Appendix A, "Regular Expressions."
Codes returned to DOS by each utility	Appendix B, "Exit Codes."
A list of error messages	Appendix C, "Error Messages."

Important *There may be additional information about the CodeView debugger in the README.DOC file. This file will describe changes made to the program after the manual was printed.*

Document Conventions

The following document conventions are used throughout this manual and apply in particular to syntax displays.

Example	Description
BP	**Boldface type** always marks standard features of either programming languages (keywords, operators, and functions) or CodeView sequential-mode commands. These terms and punctuation marks must be typed exactly as shown in order to have effect. However, the use of uppercase or lowercase letters is not always significant. Case-sensitive terms are noted in text.
number	*Placeholders* are words in italics that indicate a general kind of information; you are expected to provide the actual value. For example, consider the syntax display for the CodeView Radix command: **N***number* This syntax display asks that you enter the Radix command by typing `N`, immediately followed by some value for *number*. You could, for example, type in the entry `N8`; but you could not legally type in the word "number" itself.
`WordPtr`	`This font` is used to indicate all example programs, user input, and screen output.
Program . . . Fragment	A column of three dots tells you part of a program has been intentionally omitted.
[[*optional items*]]	Double brackets enclose optional fields in command-line and option syntax. Consider the following command-line syntax: **R** [[*register*]] [[[=]]*value*]] The double brackets around the placeholders indicate that you may enter a *register* and you may enter a *value*. The equal sign (=) indicates that you may place an equal sign before the *value*, but only if you specify a *value*.

[[*choice1* | *choice2*]]	The vertical bar indicates that you may enter one of the entries shown on either side of the bar. The following command-line syntax illustrates the use of a vertical bar: **DB** [[*address* | *range*]] The bar indicates that following the Dump Bytes command (**DB**), you can specify either an *address* or *range*. Since both are in double brackets, you can also give the command with no argument.
"Watch point"	The first time a new term is defined, it is enclosed in quotation marks.
ALT	Small capital letters are used for the names of keys and key sequences, such as ENTER, CTRL+C, and ALT+F.
Sample screens	Sample screens are shown in black and white. Your screens will look like this if you have a monochrome monitor, or if you use the /B option in the CodeView command line (see Section 1.5.4, "Starting with a Black-and-White Display").

#828/1

PART 1

The CodeView® Debugger

College
Force Academy
2000
#828/1

PART 1

The CodeView Debugger

Part 1 explains the use of the CodeView debugger. Commands, display, and interface of the debugger are presented here, while other material relevant to the debugger such as error messages and exit codes is presented in the appendixes.

Chapter 1 explains how to create a C, Fortran, BASIC, Pascal or assembly program that can be run with CodeView; it also explains how to start the debugger and select various command line options.

Chapter 2 discusses the CodeView display screen and interface, including function keys, keyboard commands, and the mouse.

Chapters 3 –12 of Part 1 describe how to use each of the Codeview commands and expressions.

CHAPTERS

1 Getting Started 5

2 The CodeView Display 33

3 Using Dialog Commands 61

4 CodeView Expressions 65

5 Executing Code 87

6 Examining Data and Expressions 99

7 Managing Breakpoints 133

8 Managing Watch Statements 141

9 Examining Code 159

10 Modifying Code or Data 171

11 CodeView Control Commands 191

12 Debugging in Protected Mode 209

#828 /1

CHAPTER 1 Getting Started

Getting started with the CodeView debugger requires several simple steps. First you must prepare a special-format executable file for the program you wish to debug; then you can invoke the debugger. You may also wish to specify options that will affect the debugger's operation.

This chapter describes how to produce executable files in the CodeView format using C, FORTRAN, BASIC, Pascal, or assembly language and describes how to load a program into the CodeView debugger. The chapter lists restrictions and programming considerations with regard to the debugger, which you may want to consult before compiling or assembling. Finally, the chapter describes how to use the debugger with Microsoft or IBM Macro Assembler, Versions 1.0 through 4.0.

1.1 Restrictions

This list describes files that are not directly supported by the debugger. The following restrictions apply generally to the use of the CodeView debugger, regardless of the language being used.

Restriction	Explanation
Include files	You will not be able to use the CodeView debugger to debug source code in include files.
Packed files	CodeView symbolic information cannot be put into a packed file.
.COM files	Files with the extension .COM can be debugged in assembly mode only; they can never contain symbolic information.

Memory-resident programs	The CodeView debugger can only work with disk-resident .EXE and .COM files. Debugging of memory-resident files is not supported.
Programs that alter the environment	Programs that run under the CodeView debugger can read the DOS environment, but they cannot permanently change it. Upon exit from CodeView, all changes to the environment are lost.
Program Segment Prefix (PSP)	The CodeView debugger automatically preprocesses a program's PSP the same way a C program does; quote marks are removed, and exactly one space is left between command-line arguments. This preprocessing only creates a problem if you are debugging a program not written in C— one that tries to access command-line arguments.

Some of the features now allowed by CodeView include debugging of library modules and debugging of overlaid code.

1.2 The CodeView Environment

Work with CodeView can be optimized in several ways.

The CVPACK utility compresses CodeView information in an executable file. See Section 17.4 for directions on how to use CVPACK.

In addition, the /E option enables CodeView to take advantage of expanded memory. Command-line options are described in Section 1.5.

1.3 Preparing Programs for the CodeView Debugger

You must compile and link with the correct options in order to use a program with the CodeView debugger. These options direct the compiler and the linker to produce an executable file, which contains line-number information and a symbol table, in addition to the executable code.

NOTE *For the sake of brevity, Sections 1.3.1-1.3.3 use the term "compiling" to refer to the process of producing object modules. However, almost everything said about compiling in this section applies equally well to assembling. Exceptions are noted in Section 1.3.8, "Preparing Assembly Programs."*

Not all compiler and linker versions support CodeView options. (See the section on the appropriate language below for information about compiler versions.) Also, you will need to use the Microsoft Overlay Linker (Version 3.6 or later) or the Microsoft Segmented-Executable Linker. If you try to debug an executable file that was not compiled and linked with CodeView options, or if you use a compiler that does not support these options, then you will only be able to use the debugger in assembly mode. This means the CodeView debugger will not be able to display source code or understand source-level symbols, such as symbols for functions and variables.

1.3.1 Programming Considerations

Any source code that is legal in C, FORTRAN, BASIC, Pascal, or Microsoft Macro Assembler can be compiled or assembled to create an executable file and then debugged with the CodeView debugger. However, some programming practices make debugging more difficult.

Each of the Microsoft languages listed above permits you to put code in separate include files and read the files into your source file by using an include directive. However, you will not be able to use the CodeView debugger to debug source code in include files. The preferred method of developing programs is to create separate object modules and then link the object modules with your program. The CodeView debugger supports the debugging of separate object modules in the same session.

Also, the CodeView debugger is more effective and easier to use if you put each source statement on a separate line. A number of languages (C and BASIC in particular) permit you to place more than one statement on a single line of the source file. This practice does not prevent the CodeView debugger from functioning. However, the debugger must treat the line as a single unit; it cannot break the line down into separate statements. Therefore, if you have three statements on the same line, you will not be able to put a breakpoint or freeze execution on the individual statements. The best you will be able to do is to freeze execution at the beginning of the three statements or at the beginning of the next line.

Some languages (C and assembly in particular) support a type of macro expansion. However, the CodeView debugger will not help you debug macros in source mode. You will need to expand the macros yourself before debugging them; otherwise, the debugger will treat them as simple statements or instructions.

Finally, your segments should be declared according to the standard Microsoft format (as described in the *Microsoft Mixed-Language Programming Guide*). This is taken care of for you automatically with each of the Microsoft high-level languages.

1.3.2 CodeView Compile Options

Microsoft compilers accept command-line options preceded by either a forward slash (/) or a dash (-). For brevity, this manual lists only the forward slash when describing options, but you may use either symbol.

The use of uppercase or lowercase letters is significant for options used with the C, FORTRAN, BASIC, and Pascal compilers; you must type the letters exactly as given.

When you compile a source file for a program you want to debug, you must specify the /Zi option on the command line. The /Zi option instructs the compiler to include line-number and symbolic information in the object file.

If you do not need complete symbolic information in some modules, you can compile those modules with the /Zd option instead of /Zi. The /Zd option writes less symbolic information to the object file, so using this option will save disk space and memory. For example, if you are working on a program made up of five modules, but only need to debug one module, you can compile that module with the /Zi option and the other modules with the /Zd option. You will be able to examine global variables and see source lines in modules compiled with the /Zd option, but local variables will be unavailable.

In addition, if you are working with a high-level language, you will probably want to use the /Od option, which turns off optimization. Optimized code may be rearranged for greater efficiency and, as a result, the instructions in your program may not correspond closely to the source lines. After debugging, you can compile a final version of the program with the optimization level you prefer.

NOTE *The /Zd option is not available with QuickBASIC. The /Od option is not available with QuickBASIC or the Macro Assembler.*

You cannot debug a program until you compile it successfully. The CodeView debugger will not help you correct syntax or compiler errors. Once you successfully compile your program, you can use the debugger to locate logical errors in the program.

Compiling examples are given in the sections below on compiling and linking with specific languages.

1.3.3 CodeView Link Options

If you use LINK separately to link an object file or files for debugging, you should specify the /CODEVIEW option (it can be abbreviated as /CO). This instructs the linker to incorporate addresses for symbols and source lines into the executable file.

If you use a Microsoft driver program that automatically invokes the linker (such as CL with C, or FL with FORTRAN), the linker is automatically invoked with the /CO option whenever you specify /Zi on the command line. You do not use /CO unless you are invoking the linker directly, by typing LINK.

Although executable files prepared with the /CODEVIEW option can be executed from the DOS command line like any other executable files, they are larger because of the extra symbolic information in them. To minimize program size, you will probably want to recompile and link your final version without the /Zi option when you finish debugging a program.

Linking examples are given in the sections below on compiling and linking with specific languages.

1.3.4 Preparing C Programs

In order to use the CodeView debugger with a program written in C, you need to compile it with the Microsoft C Compiler, Version 4.0 or later. Earlier versions of the compiler do not support the CodeView compile options. You also need to link with the Microsoft Overlay Linker, Version 3.6 or later.

Writing C Source Code

Microsoft C supports the use of include files through the use of the **#include** directive. However, you will not be able to debug source code put into include files. Therefore, you should reserve the use of include files for **#define** prototypes, macros, and structure definitions.

The C language permits you to put more than one statement on a line. This practice makes it difficult for you to debug such lines of code. For example, the following code is legal in C:

```
code = buffer[count]; if (code == '\n') ++lines;
```

This code is made up of three separate source statements. When placed on the same line, the individual statements cannot be accessed during debugging. You could not, for example, stop program execution at `++lines;`. The same code would be easier to debug in the following form:

```
code = buffer[count];
if (code == '\n')
        ++lines;
```

This makes code easier to read and corresponds with what is generally considered good programming practice.

You cannot easily debug macros with the CodeView debugger. The debugger will not break down the macro for you. Therefore, if you have complex macros with potential side effects, you may need to write them first as regular source statements.

Compiling and Linking C Programs

The /Zi, /Zd, and /Od options are all supported by the Microsoft C Compilers, Versions 4.0 and later. (For a description of these options, see Section 1.3.2, "CodeView Compile Options.") The options are accepted by the CL driver and the MSC driver, which was supplied with Version 4.0. Linking separately with /CO is necessary when you compile with MSC.

The CodeView debugger supports mixed-language programming. For an example of how to link a C module with modules from other languages, see Section 1.3.8, "Preparing Assembly Programs."

Examples

```
CL /Zi /Od EXAMPLE.C

CL /Zi /Od /c EXAMPLE.C
LINK /CO EXAMPLE;

CL /Zi /Od /c MOD1.C
CL /Zd /Od /c MOD2.C
CL /Zi MOD1 MOD2
```

In the first example, CL is used to compile and link EXAMPLE.C, the source file. The CL driver creates an object file, EXAMPLE.OBJ, in the CodeView format, and then automatically invokes the linker with the /CO option. The second example demonstrates how to compile and link source file EXAMPLE.C by using the MSC program provided with Version 4.0 of the compiler. Since MSC does not invoke the linker, you must invoke the linker directly and specify /CO on the command line. Both examples will result in an executable file, EXAMPLE.EXE, which has the line-number information, symbol table, and unoptimized code required by the CodeView debugger.

In the third example, the source module MOD1.C is compiled to produce an object file with full symbolic and line information, while MOD2.C is compiled to produce an object file with limited information. Then, CL is used again to link the resulting object files. (This time, CL does not recompile because the arguments have no .C extension.) Typing /Zi on the command line causes the linker to be invoked with the /CO option. The result is an executable file in which one of the modules, MOD2.C, will be harder to debug. However, the executable file will take up less space on disk than it would if both modules were compiled with full symbolic information.

1.3.5 Preparing FORTRAN Programs

In order to use the CodeView debugger with a program written in FORTRAN, you will need to compile it with the Microsoft FORTRAN Compiler, Version 4.0 or later. Earlier versions of the compiler do not support CodeView compile options. You will also need to link with the Microsoft Overlay Linker, Version 3.6 or later.

Writing FORTRAN Source Code

The Microsoft FORTRAN Compiler supports the use of include files, through use of the **$INCLUDE** directive. However, you will not be able to debug source code in an include file. If you have source code that you wish to put in separate files, then you should use the technique of separately compiled modules. The CodeView debugger does support this technique by allowing you to trace through separate source files in the same session.

Compiling and Linking FORTRAN Programs

The /Zi, /Zd, and /Od options are all supported by the Microsoft FORTRAN Compiler, Version 4.0 or later. For a description of these options, see Section 1.3.2, "CodeView Compile Options." The CodeView debugger supports mixed-language programming. For an example of how to link a FORTRAN module with modules from other languages, see Section 1.3.8, "Preparing Assembly Programs."

Examples

```
FL /Zi /Od EXAMPLE.FOR

FL /Zi /Od /c EXAMPLE.FOR
LINK /CO EXAMPLE;

FL /Zi /Od /c MOD1.FOR
FL /Zd /Od /c MOD2.FOR
FL /Zi MOD1 MOD2
```

In the first example, the FL driver is used to compile and link the source file EXAMPLE.FOR. The FL driver creates an object file in the CodeView format, EXAMPLE.OBJ, and then automatically invokes the linker with the /CO option. The second example demonstrates how to compile and link the source file EXAMPLE.FOR by using separate steps for compiling and linking. In this case, the /CO option must be given explicitly to the linker. Both examples result in an executable file, EXAMPLE.EXE, which has the line-number information, symbol table, and unoptimized code required by the CodeView debugger.

In the third example, the source module MOD1.FOR is compiled to produce an object file with full symbolic and line information, while MOD2.FOR is compiled to produce an object file with limited information. Then FL is used again to link the object files. (Note that this time, FL does not recompile because the arguments have no .FOR extension.) Typing /Zi on the command line causes the linker to be invoked with the /CO option. The result is an executable file in which one of the modules, MOD2.FOR, will be harder to debug. However, the executable file takes up less space on disk than it would if both modules were compiled with full symbolic information.

1.3.6 Preparing BASIC Programs

In order to use the CodeView debugger with a program written in BASIC, you will need to compile it with Microsoft QuickBASIC, Version 4.0 or later. You will also need to link with the Microsoft Overlay Linker, Version 3.6 or later.

Writing BASIC Source Code

Microsoft BASIC supports the use of include files, through the use of the **$INCLUDE** directive. However, you will not be able to debug source code put into include files. The preferred practice for developing source code in separate files is to use separately compiled modules. The CodeView debugger does support this technique by allowing you to trace through separate source files in the same session.

BASIC also permits you to put more than one statement on a line. This practice makes it difficult for you to debug such lines of code. For example, the following code is legal, even common, in BASIC:

```
SUM=0 : FOR I=1 TO N : SUM=SUM+ARRAY(I) : NEXT I
```

This code is actually made up of four separate BASIC statements. When placed on the same line, the individual statements cannot be accessed during debugging. You could not, for example, stop program execution at `SUM=SUM+ ARRAY(I)`. The same code would be easier to debug if it were written in the following form:

```
SUM=0
FOR I=1 TO N
    SUM=SUM+ARRAY(I)
NEXT I
```

Compiling and Linking BASIC Programs

Microsoft QuickBASIC Versions 4.0 and later can prepare BASIC programs for use with the CodeView debugger, through the use of the BC command line.

You cannot prepare programs for use with CodeView when you are in the QuickBASIC editor itself. Instead, compile separately with the BC command-line option /Zi. The /Zi option is described in Section 1.3.2, "CodeView Compile Options." You must also link separately with /CO.

The CodeView debugger supports mixed-language programming. For an example of how to link a BASIC module with modules from other languages, see Section 1.3.8, "Preparing Assembly Programs."

Example

```
BC /Zi EXAMPLE;
LINK /CO EXAMPLE;
```

The example above compiles the source file EXAMPLE.BAS to produce an object file, EXAMPLE.OBJ, which contains the symbol and line-number information required by the CodeView debugger. Then the linker is invoked with the /CO option to create an executable file that can be used with the debugger.

1.3.7 Preparing Pascal Programs

In order to use the CodeView debugger with a program written in Pascal, you need to compile it with the Microsoft Pascal Compiler, Version 4.0 or later. Earlier versions of Pascal do not support the CodeView compile options. You also need to link with the Microsoft Overlay Linker, Version 3.6 or later.

NOTE *If you have a version of Microsoft Pascal earlier than Version 4.0, you can use the CodeView debugger to a limited extent. However, the debugger will not be able to evaluate program symbols in CodeView commands. Compile a program as you would normally, and then link with the /CO option as explained below. You will then be able to use CodeView to step through your program and set breakpoints. The debugger will also be able to display machine-level code and do memory dumps. This version of CodeView does not include a Pascal expression evaluator.*

Writing Pascal Source Code

Microsoft Pascal supports the use of include files by providing the **$include** metacommand. However, you will not be able to debug source code put into include files. You can easily debug code in separately compiled source files. Use this technique, rather than that of include files, to debug a large program.

Pascal allows you to put more than one statement on a line; yet it is difficult to debug programs with multiple statements on a single line. For example, the following code is legal in Pascal:

```
if i = max then begin k := k+1; i = 0 end;
```

This code is actually made up of five separate source statements. When placed on the same line, the individual statements cannot be accessed during debugging. You could not, for example, stop program execution at `k := k+1;`. The same code would be easier to debug if it were written as the following:

```
if i = max then
      begin
         k := k+1;
         i := 0
      end;
```

Writing only one statement on a line makes code easier to read and corresponds with what is generally considered good programming practice.

Compiling and Linking Pascal Programs

Versions 4.0 and later of Microsoft Pascal support the CodeView options /Zi and /Zd when you use the PL driver program. (For a description of these options, see Section 1.3.2, "CodeView Compile Options.") The CodeView compile options are put on the command line when invoking the first pass of the Pascal compiler.

The /CO option is necessary only when you link separately.

Example

```
PL /Zi /c TEST
LINK /CO TEST;
```

The example above compiles the source file TEST.PAS to produce an object file, TEST.OBJ, which contains the symbol and line-number information required by the CodeView debugger. Then the linker is invoked with the /CO option.

The CodeView debugger supports mixed-language programming. For an example of how to link a Pascal module with modules from other languages, see Section 1.3.8 below.

1.3.8 Preparing Assembly Programs

In order to use all the features of the CodeView debugger with assembly programs, you need to assemble with Microsoft Macro Assembler, Version 5.0 or later. (Section 1.7 discusses how to use earlier versions of Microsoft Macro Assembler with the debugger.) No matter what version of the assembler you use, you will need to link with the Microsoft Overlay Linker, Version 3.6 or later.

Writing Assembler Source Code

If you have Version 5.0 or later of the Microsoft Macro Assembler, you can use the simplified segment directives described in the *Microsoft Macro Assembler Programmer's Guide*. Use of these directives ensures that segments are declared in the correct way for use with the CodeView debugger. (These directives also aid mixed-language programming.) If you do not use these directives, make sure that the class name for the code segment ends with the letters `CODE`.

NOTE *The CodeView debugger correctly recognizes floating-point values only when they are in the IEEE (Institute of Electrical and Electronics Engineers, Inc.) format. You should use the IEEE format with any program that you are going to run with the CodeView debugger if that program uses floating-point variables. The IEEE format is the default for Version 5.0 or later of the Microsoft Macro Assembler. You can always specify IEEE format by using the* ***.8087*** *or* ***.287*** *directive, or by assembling with the* /R *option.*

You will not be able to trace through macros while in source mode. Macros will be treated as single instructions unless you are in assembly or mixed mode, so you will not see comments or directives within macros. Therefore, you may want to debug code before putting it into a macro.

The Microsoft Macro Assembler also supports include files, but you will not be able to debug code in an include file. You are better off reserving include files for macro and structure definitions.

Because the assembler does not have its own expression evaluator, you will have to use either the C, FORTRAN, or BASIC expression evaluator. C is the default because it is the closest to assembly language. To make sure the expression evaluator recognizes your symbols and labels, you should observe the following guidelines when writing assembly modules:

1. The assembler has no explicit way to declare real numbers. However, it will pass the correct symbolic information for reals and integers if you initialize each real number with a decimal point and each integer without a decimal point. (The default type is integer.) For example, the following statements correctly initialize `REALSUM` as a real number and `COUNTER` as an integer:

   ```
   REALSUM     DD     0.0
   COUNTER     DD     0
   ```

   You must initialize real number data in data definitions. If you use `?`, the assembler will consider the variable an integer when it generates symbolic information. The CodeView debugger, in turn, will not properly evaluate the value of the variable.

2. Avoid the use of special characters in symbol names. The C, FORTRAN, and BASIC expression evaluators each apply their own standards in detemining what is a legal symbol name. Generally, only alphanumeric characters and the underscore (_) are recognized. BASIC accepts certain type-declaration characters at the end of a name, but C and FORTRAN do not.

3. Assemble with /MX or /ML to avoid conflicts due to case when you do mixed-language programming. By default, the assembler converts all symbols to uppercase when it generates object code. C, however, does not do this conversion. Therefore, the CodeView debugger will not recognize that `var` in a C program and `var` in an assembly program are the same variable unless you leave Case Sense off when using the debugger.

4. If you access command-line data in the Program Segment Prefix (PSP), note that the CodeView debugger changes the PSP; tabs, quote marks, and extra spaces are removed so that exactly one space separates each argument. The debugger retains quote marks (along with any quoted material) for command lines given with the **L** command.

Assembling and Linking

The assembler supports the /Zi and /Zd assemble-time options. The /Od option does not apply, and so is not supported. Assembler options are not case sensitive. You may therefore enter /ZI or /ZD on the assembler command line to produce an object file in the CodeView format.

If you link your assembly program with a module written in C (which is case sensitive), you probably need to assemble with /MX or /ML.

After assembling, link with the /CO option to produce an executable file in the CodeView format.

Examples

```
MASM /ZI EXAMPLE;
LINK /CO EXAMPLE;

MASM /ZI MOD1;
MASM /ZD MOD2;
LINK /CO MOD1 MOD2;

CL /Zi /Od /c /AL prog.c
BC /Zi sub1;
MASM /ZI /MX sub2;
LINK /CO prog sub1 sub2
```

The first example assembles the source file EXAMPLE.ASM and produces the object file EXAMPLE.OBJ, which is in the CodeView format. The linker is then invoked with the /CO option and produces an executable file with the symbol table and line-number information required by the debugger.

The second example produces the object file MOD1.OBJ, which contains symbol and line-number information, and the object file MOD2.OBJ, which contains line-number information but no symbol table. The object files are then linked. The result is an executable file in which the second module will be harder to debug. This executable file, however, will be smaller than it would be if both modules were assembled with the /ZI option.

The last example demonstrates how to create a mixed-language executable file that can be used with the CodeView debugger. The debugger is able to trace through different source files in the same session, regardless of the language.

1.4 Starting the CodeView Debugger

Before starting the debugger, make sure all the files it requires are available. The following files are recommended for source-level debugging:

File	Location
CV.EXE	The CodeView program file can be in the current directory or in any directory accessible with the PATH command. For example, if you are using a hard disk setup, you might put CV.EXE in the \BIN directory. If you have an older version of the debugger, take care to remove any copies of CV.EXE from directories in your PATH. The debugger has an overlay manager that reloads the file CV.EXE from time to time. If it reloads the wrong version of this file, your machine will likely crash.
CV.HLP	If you want to have the on-line help available during your session, you should have this file either in the current directory or in any directory accessible with the PATH command. For example, if you set up your compiler files on a hard disk using the SETUP program provided on the distribution disk, you might put CV.HLP in the \BIN directory. If the CodeView debugger cannot find the help file, you can still use the debugger, but you will see an error message if you use one of the help commands.
program.EXE	The executable file for the program you wish to debug must be in the current directory or in a drive and directory you specify as part of the start-up file specification. The CodeView debugger will display an error message and will not start unless the executable file is found.

source.ext (extension depends on language)	Normally, source files should be in the current directory. However, if you specify a file specification for the source file during compilation, that specification becomes part of the symbolic information stored in the executable file. For example, if you compiled with the command line argument DEMO, the CodeView debugger expects the source file to be in the current directory. However, if you compiled with the command line argument \SOURCE\DEMO, the debugger expects the source file to be in the directory \SOURCE. If the debugger cannot find the source file in the directory specified in the executable file (usually the current directory), the program prompts you for a new directory. You can either enter a new directory, or you can press ENTER to indicate that you do not want a source file to be used for this module. If no source file is specified, you must debug in assembly mode.

If the appropriate files are in the correct directories, you can enter the CodeView command line at the DOS command prompt. The command line has the following form:

CV [[*options*]] *executablefile* [[*arguments*]]

The *options* are one or more of the options described in Section 1.5. The *executablefile* is the name of an executable file to be loaded by the debugger. It must have the extension .EXE or .COM. If you try to load a nonexecutable file, the following message appears:

```
Not an executable file
```

Compiled programs and assembly-language programs containing CodeView symbolic information will always have the extension .EXE. Files with the extension .COM can be debugged in assembly mode, but they can never contain symbolic information.

The optional *arguments* are parameters passed to the *executablefile*. If the program you are debugging does not accept command-line arguments, you do not need to pass any arguments.

If you specify the *executablefile* as a file name with no extension, the CodeView debugger searches for a file with the given base name and the extension .EXE. Therefore, you must specify the .COM extension if you are debugging a .COM file. If the file is not in the CodeView format, the debugger starts in assembly mode and displays the following message:

```
No symbolic information
```

You must specify an executable file when you start the CodeView debugger. If you omit the executable file, the debugger displays a message showing the correct command-line format.

When you give the debugger a valid command line, the executable program and the source file are loaded, the address data are processed, and the CodeView display appears. The initial display will be in window mode or sequential mode, depending on the options you specify and the type of computer you have.

For example, if you wanted to debug the program BENCHMRK.EXE, you could start the debugger with the following command line:

```
CV BENCHMRK
```

If you give this command line on an IBM Personal Computer, window mode is selected automatically. The display will look like Figure 1.1.

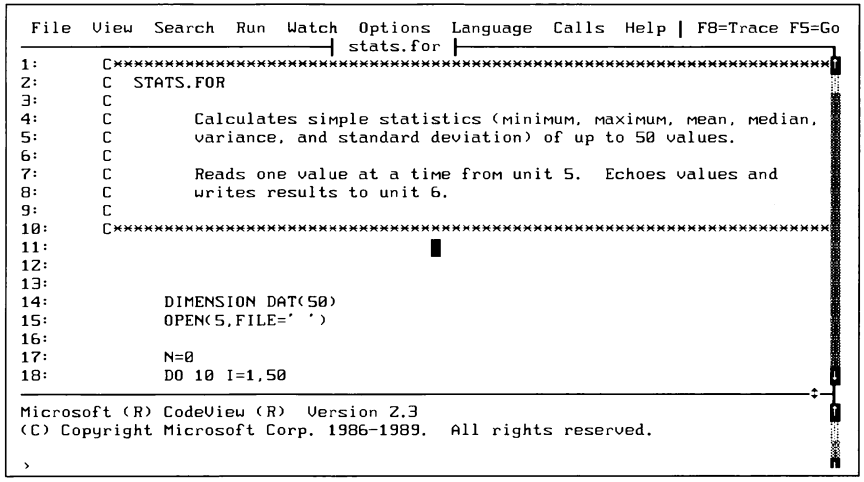


Figure 1.1 CodeView Start-up Screen in Window Mode

If you give the same command line on most non-IBM computers, sequential mode will be selected. The following lines appear:

```
Microsoft (R) CodeView (R)  Version 2.3
(C) Copyright Microsoft Corp. 1986-1989. All rights
reserved.


>
```

You can use CodeView options to override the default start-up mode.

If your program is written in a high-level language, the CodeView debugger is now at the beginning of the start-up code that precedes your program. In source mode, you can enter an execution command (such as Trace or Program Step) to execute automatically through the start-up code to the beginning of your program. At this point, you are ready to start debugging your program, as described in Chapters 3–11.

1.5 Using CodeView Options

You can change the start-up behavior of the debugger by specifying options in the command line. An option is a sequence of characters preceded by either a forward slash (/) or a dash (–).

For brevity, this manual will list only the forward slash when describing options, but you may use either. Unlike compiler command-line options, CodeView command-line options are not case sensitive.

A file whose name begins with a dash must be renamed before you use it with the CodeView debugger so the debugger will not interpret the dash as an option designator. You can use more than one option in a command line, but each option must have its own option designator, and spaces must separate each option from other elements of the line.

NOTE *The CodeView debugger's defaults for IBM Personal Computers are different from the defaults it has for other computers. However, the debugger may not always recognize the difference between computers, and defaults may vary accordingly.*

The following list suggests some situations in which you might want to use an option. If more than one condition applies, you can use more than one option (in any order). If none of the conditions applies, you need not use any options.

Condition	Option
You want to use two monitors with the CodeView debugger.	/2
You want a 43-line display and you have an IBM or IBM-compatible computer with an enhanced graphics adapter (EGA) and an enhanced color display.	/43

You want a 50-line display and you have a graphics adapter card that supports 50-line mode.	/50
You have a two-color monitor, a color graphics adapter, and an IBM or IBM-compatible computer.	/B
You want the CodeView debugger to automatically execute a series of commands when it starts up.	/C*commands*
You are using an IBM-compatible computer that does not support certain IBM-specific interrupt-trapping functions.	/D
You have expanded memory and want the CodeView debugger to take advantage of it.	/E
You are using an IBM-compatible computer to debug a program that does not use graphics or multiple video-display pages, and you want to be able to see the output screen.	/F
You are using a non-IBM-compatible computer and want to enable CTRL+C and CTRL+BREAK.	/I
You run CodeView (CVP) in OS/2 protected mode and want to debug dynamic-link libraries.	/L
You have a mouse installed in your system, but do not want to use it during the debugging session.	/M
You run CodeView (CVP) with OS/2, Version 1.10 or later, and wish to debug multiple processes.	/O
You have a non-IBM EGA and have problems running the debugger.	/P
You have a 386 processor and wish to use the debug registers to speed up the execution of tracepoints.	/R

You are debugging a graphics program or a program that uses multiple video-display pages, and you want to be able to see the output screen.	/S
You are using a non-IBM-compatible computer and want to be able to see the output screen.	/S
You have an IBM computer, but wish to debug in sequential mode (for example, with redirection).	/T
You have an IBM-compatible computer and want to use window mode.	/W

For example, assume you are using an IBM-compatible computer with a color graphics adapter (CGA) and a two-color monitor. The program you are debugging, which you could name GRAPHIX.EXE, plots points in graphics mode. You want to be able to see the output screen during the debugging session. Finally, you want to be able to start the debugger several times without having to remember all the options, and you want to execute the high-level language start-up code automatically each time. You could create a batch file consisting of the following line:

```
CV /W /B /S /CGmain GRAPHIX
```

The CodeView options are described in more detail in Sections 1.5.1–1.5.16 below.

1.5.1 Using Two Video Adapters

Option

/2

The /2 option permits the use of two monitors with the CodeView debugger. The program display will appear on the current default monitor, while the CodeView display appears on the other monitor. You must have two monitors and two adapters to use the /2 option. For instance, if you have both a color graphics adapter and a monochrome adapter, you might want to set the CGA up as the default adapter. You could then debug a graphics program with the graphics display appearing on the graphics monitor and the debugging display appearing on the monochrome monitor.

1.5.2 Using the Enhanced Graphics Adapter's 43-Line Mode

Option

/43

If you have an enhanced graphics adapter (EGA) and a monochrome monitor or an enhanced color display monitor (or a compatible monitor), you can use the /43 option to enable a 43-line-by-80-column text mode. You cannot use this mode with other monitors, such as a CGA or a monochrome adapter (MA). The CodeView debugger will ignore the option if it does not detect an EGA.

The EGA's 43-line mode performs the same as the normal 25-line-by-80-column mode used by default on the EGA, CGA, and MA. The advantage of the 43-line mode is that more text fits on the CodeView display; the disadvantage is that the text is smaller and harder to read. If you have an EGA, you can experiment to see which size you prefer.

The video graphics adapter (VGA) card also supports this option.

Example

```
CV /43 CALC CALC.DAT
```

The example above starts the CodeView debugger in 43-line mode if you have an EGA video adapter and an enhanced color or monochrome monitor. The option will be ignored if you lack the hardware to support it.

1.5.3 Using 50-Line Mode

Option

/50

If you have a graphics adapter card (such as a VGA) and monitor that support 50-line mode, then you can use the /50 option to enable a 50-line-by-80-column text mode. You cannot use this mode with most adapters, such as a CGA or an MA. The CodeView debugger will ignore the option if your hardware does not support 50-line mode.

The 50-line mode performs the same as the normal 25-line-by-80-column mode used by default on the EGA, VGA, CGA, and MA. The advantage of the 43-line mode is that more text fits on the CodeView display; the disadvantage is that the text is smaller and harder to read.

Example

```
CV /50 CALC CALC.DAT
```

The example above starts the CodeView debugger in 50-line mode if this mode is supported by your hardware.

1.5.4 Starting with a Black-and-White Display

Option

/B

The /B option forces the CodeView debugger to display in two colors even if you have a color adapter (CGA, EGA, or compatible). By default, the debugger checks on start-up to see what kind of display adapter is attached to your computer. If the debugger detects an MA, it displays in two colors. If it detects a color adapter, it displays in multiple colors.

If you use a two-color monitor with a CGA or EGA, you may want to disable color. Monitors that display in only two colors (usually green and black, or amber and black) often attempt to show colors with different cross-hatching patterns, or in gray-scale shades of the display color. In either case, you may find the display easier to read if you use the /B option to force black-and-white display. Most two-color monitors still have four color distinctions: background (black), normal text, high-intensity text, and reverse-video text.

Example

```
CV /B CALC CALC.DAT
```

The example above starts the CodeView debugger in black-and-white mode. This is the only mode available if you have an MA. The display is usually easier to read in this mode if you have a CGA and a two-color monitor.

1.5.5 Specifying Start-Up Commands

Option

/C*commands*

The /C option allows you to specify one or more *commands* that will be executed automatically upon start-up. You can use these options to invoke the debugger from a batch or MAKE file. Each command is separated from the previous command by a semicolon.

If one or more of your start-up commands have arguments that require spaces between them, you should enclose the entire option in double quotation marks. Otherwise, the debugger will interpret each argument as a separate CodeView command-line argument rather than as a debugging-command argument.

WARNING *Any start-up option that uses the less-than (<) or greater-than (>) symbol must be enclosed in double quotation marks even if it does not require spaces. This ensures that the redirection command will be interpreted by the CodeView debugger rather than by DOS.*

Examples

```
CV /CGmain CALC CALC.DAT
```

The example above loads the CodeView debugger with `CALC` as the executable file and `CALC.DAT` as the argument. Upon start-up, the debugger executes the high-level-language start-up code with the command `Gmain`. Since no space is required between the CodeView command `(G)` and its argument `(main)`, the option is not enclosed in double quotation marks.

```
CV "/C;S&;G INTEGRAL;DS ARRAYX L 20" CALC CALC.DAT
```

The example above loads the same file with the same argument as the first example, but the command list is more extensive. The debugger starts in mixed source/assembly mode (`S&`). It executes to the routine `INTEGRAL` (`G INTEGRAL`), and then dumps 20 short real numbers, starting at the address of the variable `ARRAYX` (`DS ARRAYX L 20`). Since several of the commands use spaces, the entire option is enclosed in double quotation marks.

```
CV "/C<INPUT.FIL" CALC CALC.DAT
```

The example above loads the same file and argument as the first example, but the start-up command directs the debugger to accept the input from the file `INPUT.FIL` rather than from the keyboard. Although the option does not include any spaces, it must be enclosed in double quotation marks so that the less-than symbol will be read by the CodeView debugger rather than by DOS.

1.5.6 Handling Interrupt Trapping (DOS Only)

Options

/D

The /D option turns off nonmaskable interrupt (NMI) trapping and 8259 interrupt trapping. If you are using an IBM PC Convertible, Tandy® 1000, or the AT&T® 6300 Plus and you are experiencing system crashes with CodeView,

try starting with the /D option. To enable window mode, use /W with /D; otherwise sequential mode is set automatically. Note that because this option turns off interrupt trapping, CTRL+C and CTRL+BREAK will not work, and an external interrupt may occur during a trace operation. If this happens, you may find yourself tracing the interrupt handler instead of your program.

The /I option forces the debugger to handle NMI and 8259 interrupt trapping. Use this option to enable CTRL+C and CTRL+BREAK on computers not recognized as being IBM compatible by the debugger, such as the Eagle® PC. Window mode is set automatically with the /I option; you don't have to specify /W. Using the /I option lets you stop program execution at any point while you are using the CodeView debugger.

1.5.7 Using Expanded Memory (DOS Only)

Option

/E

"Expanded memory" refers to memory made accessible according to the Microsoft/Lotus®/Intel® EMS specification. This access provides your system with memory above the 640K MS-DOS limitation on RAM. However, since MS-DOS will not recognize this additional memory, programs can make use of expanded memory in limited ways.

The /E option enables the use of expanded memory. If expanded memory is present, the CodeView debugger uses it to store the symbolic information of the program. This may be as much as 85% of the size of the executable file for the program, and represents space that would otherwise be taken up in main memory.

NOTE *This option enables only expanded memory, not extended memory. Extended memory makes use of protected-mode instructions, rather than the Microsoft/Lotus/Intel specification for memory paging.*

1.5.8 Setting the Screen-Exchange Mode (DOS Only)

Options

/F
/S

The CodeView debugger allows you to move quickly back and forth between the output screen, which contains the output from your program, and the debugging screen, which contains the debugging display. The debugger can handle this screen exchange in two ways: screen flipping or screen swapping. The /F option (screen flipping) and the /S option (screen swapping) allow you to choose

the method from the command line. If neither method is specified (possible only on non-IBM computers), the Screen Exchange command will not work. No screen exchange is the default for non-IBM computers. Screen flipping is the default for IBM computers with graphics adapters, and screen swapping is the default for IBM computers with monochrome adapters. Screen flipping uses the video-display pages of the graphics adapter to store each screen of text. Video-display pages are a special memory buffer reserved for multiple screens of video output. This method is faster and uses less memory than screen swapping. However, screen flipping cannot be used with an MA, nor to debug programs that produce graphics or use the video-display pages. In addition, the CodeView debugger's screen flipping works only with IBM and IBM-compatible microcomputers.

Screen swapping has none of the limitations of screen flipping, but is significantly slower and requires more memory. In the screen-swapping method, the CodeView debugger creates a buffer in memory and uses it to store the screen that is not being used. When the user requests the other screen, the debugger swaps the screen in the display buffer for the one in the storage buffer.

When you use screen swapping, the buffer size is 16K for all adapters. The amount of memory used by the CodeView debugger is increased by the size of the buffer.

Table 1.1 shows the default exchange mode (swapping or flipping) and the default display mode (sequential or window) for various configurations. Display modes are discussed in Section 1.5.14, "Enabling Window or Sequential Mode."

Table 1.1 Default Exchange and Display Modes

Computer	Display Adapter	Default Modes	Alternate Modes
IBM	CGA or EGA	/F /W	/S if your program uses video-display pages or graphics; /T for sequential mode
IBM compatible	CGA or EGA	/T	/W for window mode; /F for screen flipping with text programs; or /S for screen swapping with programs that use video-display pages or graphics
IBM	MA	/S /W	/T for sequential mode
IBM compatible	MA	/T	/W for window mode; /S for screen swapping
Noncompatible	Any	/T	/S for screen swapping

If you are not sure if your computer is completely IBM compatible, you can experiment. If the basic input/output system (BIOS) of your computer is not compatible enough, the CodeView debugger may not work with the /F option.

If you specify the /F option with an MA, the debugger ignores the option and uses screen swapping. If you try to use screen flipping to debug a program that produces graphics or uses the video-display pages, you may get unexpected results and have to start over with the /S option.

Examples

```
CV /F CALC CALC.DAT
```

The example above starts the CodeView debugger with screen flipping. You might use this command line if you have an IBM-compatible computer, and you want to override the default screen-exchange mode in order to use less memory and switch screens more quickly. The option would not be necessary on an IBM computer, since screen flipping is the default.

```
CV /S GRAFIX
```

The example above starts the debugger with screen swapping. You might use this command line if your program uses graphics mode.

1.5.9 Loading Information from Dynamic-Link Libraries (OS/2 Only)

Option

/L *dynlib*

The /L option directs the protected-mode CodeView debugger (CVP) to search *dynlib* for symbolic information. At least one space must separate /L from *dynlib*.

CVP can debug dynamic-link libraries, but only if it is told what libraries to search at run time. When you place a module in a dynamic-link library, neither code nor symbolic information for that module is stored in an application's executable (.EXE) file. Instead, the code and symbols are stored in the library and are not brought together with the main program until run time.

Thus, the protected-mode debugger needs to search the dynamic-link library for symbolic information. Because the debugger does not automatically know which libraries to look for, use /L to load symbolic information for dynamic-link libraries. You should use this option only with libraries that you have created and wish to debug.

Example

```
CVP /L DLIB1.DLL /L GRAFLIB.DLL PROG
```

In the example above, CVP is invoked to debug the program PROG.EXE. To find symbolic information needed for debugging each module, CVP searches libraries DLIB1.DLL and DLIB2.DLL, as well as the executable file PROG.EXE.

1.5.10 Turning Off the Mouse

Option

/M

If you have a mouse installed on your system, you can tell the CodeView debugger to ignore it by using the /M option. You may need to use this option if you are debugging a program that uses the mouse and your mouse is not a Microsoft Mouse. This is due to a conflict between the program's use of the mouse and the debugger's use of it. Use of /M may possibly disable the program's use of the mouse, as well as CodeView's.

NOTE *The same conflict between program and debugger applies if you are not using the current Microsoft Mouse driver program (MOUSE.SYS), which is included on the distribution disks for certain Microsoft products. You may want to replace your old mouse driver program with the updated version. You will then be able to use the mouse with both the CodeView debugger and the program you are debugging. If you did not install a mouse driver when you set up Version 4.0 of Microsoft FORTRAN, Version 5.0 or later of Microsoft C, or Version 5.0 or later of Macro Assembler, see your user's guide for information on installing MOUSE.SYS. These programs may not work with pointing devices from other manufacturers.*

1.5.11 Debugging Multiple Processes (OS/2 only)

Option

/O

If you are running OS/2, version 1.10 or later, you can use the /O option to enable debugging of multiple processes. See Chapter 12, "Debugging in Protected Mode," for more information on how to debug multiple processes.

NOTE *The /O option is incompatible with the /2 option.*

1.5.12 Extending EGA Compatibility

Option

/P

The use of the /P option may enable the CodeView debugger to run properly in window mode on a non-IBM version of the enhanced graphics adapter (EGA).

Normally, the debugger saves and restores the palette registers of an EGA. However, although this procedure works perfectly well with an IBM EGA, it can create conflicts with other EGAs. The /P option prevents the saving and restoring of palette registers, and so may enhance compatibility.

Symptoms that may indicate the need for using /P include the debugging screen starting in nonstandard colors and the debugger appearing to crash while in window mode.

NOTE *The /P option may cause the program being debugged to lose some colors whenever you switch back and forth between the debugging screen and the output screen. Therefore, do not use the /P option unless necessary.*

1.5.13 Using Debug Registers (386 Only)

Option

/R

If you have a 386 processor, you can enable the four debug registers by giving the /R option. The debug registers can hold up to four tracepoints. Normally, tracepoints slow down execution of the program substantially since CodeView must interrupt the program after each instruction and test all tracepoints and watchpoints. Use of debug registers lets CodeView implement tracepoints through the processor itself. CodeView can therefore execute the program at normal speed even though areas of memory are being monitored.

If you specify more than four watchpoints or specify any watch expression, CodeView does not use the debug registers.

1.5.14 Enabling Window or Sequential Mode

Options

/T

The CodeView debugger can operate in window mode or sequential mode. Window mode displays up to four windows, enabling you to see different aspects of the debugging-session program simultaneously. You can also use a mouse in window mode. Window mode requires an IBM or IBM-compatible microcomputer. Sequential mode works with any computer and is useful with redirection commands. Debugging information is displayed sequentially on the screen.

The behavior of each mode is discussed in detail in Chapter 2, "The CodeView Display." Refer back to Table 1.1 for the default and alternate modes for your computer. If you are not sure if your computer is completely IBM compatible,

you can experiment with the options. If the BIOS of your computer is not compatible enough, you may not be able to use window mode (the /W option).

NOTE *Although window mode is more convenient, any debugging operation that can be done in window mode can also be done in sequential mode.*

Examples

```
CV /W SIEVE
```

The example above starts the CodeView debugger in window mode. You will probably want to use the /W option if you have an IBM-compatible computer since the default sequential mode is less convenient for most debugging tasks.

```
CV /T SIEVE
```

The example above starts the debugger in sequential mode. You might want to use this option if you have an IBM computer and have a specific reason for using sequential mode. For instance, sequential mode usually works better if you are redirecting your debugging output to a remote terminal.

1.6 Debugging Large Programs

Because the CodeView debugger must reside in memory along with the program you are debugging, there may not be enough room to debug some large programs that could otherwise run in memory alone. However, there are at least three ways to get around memory limitations:

1. If you have expanded memory, use the /E option described earlier. This will enable CodeView to put the symbol table in expanded memory, thus freeing up a good deal of main memory.
2. Since CodeView now supports the debugging of overlaid programs, you can substantially reduce the amount of memory required to run your program by using overlays when you link your program.
3. Save space by using /Zi with modules you plan to focus on in the debugging session only, using /Zd with other modules.

1.7 Working with Older Versions of the Assembler

You can run the CodeView debugger with files developed using older versions of the Microsoft or IBM assemblers (prior to 5.0). Since older versions do not write line numbers to object files, some of the CodeView debugger's features will not be available when you debug programs developed with the older assemblers. The following considerations apply in addition to the considerations mentioned in Section 1.3.8, "Preparing Assembly Programs."

The procedure for assembling and debugging .EXE files by using older versions of the assembler is summarized below. The debugger can be used on either .EXE or .COM files, but you can only view symbolic information in .EXE files.

1. In your source file, declare public any symbols, such as labels and variables, that you want to reference in the debugger. If the file is small, you may want to declare all symbols public.
2. As mentioned earlier, make sure that the code segment class name ends with the letters `CODE`. (For example: `'MYCODE`.)
3. Assemble as usual. No special options are required, and all assembly options are allowed.
4. Use LINK, Version 3.6 or later. Do not use the linker provided with older assembler versions. Use the /CODEVIEW option when linking.
5. Debug in assembly mode (this is the start-up default if the debugger fails to find line-number information). You cannot use source mode for debugging, but you can load the source file into the display window and view it in source mode. Any labels or variables that you declared public in the source file can be displayed and referenced by name instead of by address. However, they cannot be used in expressions because type information is not written to the object file.

CHAPTER 2

The CodeView Display

2

The Microsoft CodeView debugger screen display can appear in two different modes—window and sequential. Either mode provides a useful debugging environment, but window mode is more powerful and convenient. The CodeView debugger accepts either window commands or dialog commands. Dialog commands are entered as command lines following the CodeView prompt (>) in sequential mode. They are discussed in Chapter 3, "Using Dialog Commands."

You will probably want to use window mode if you have the hardware to support it. In window mode, the pull-down menus, function keys, and mouse support offer fast access to the most common commands. Different aspects of the program and debugging environment can be seen in different windows simultaneously. Window mode is described in Section 2.1 below.

Sequential mode is similar to the display mode of the CodeView debugger's forerunner, the Microsoft Symbolic Debug Utility (SYMDEB) and the DOS DEBUG utility. This mode is required if you do not have an IBM-compatible computer, and it is sometimes useful when redirecting command input or output. Sequential mode is described in Section 2.2.

2.1 Using Window Mode

The elements of the CodeView display marked in Figure 2.1 below include the following:

1. The display window shows the program being debugged. It can contain source code (as in the example), assembly-language instructions, or any specified text file.

2. The current location line (the next line the program will execute) is displayed in reverse video or in a different color. This line may not always be visible because you can scroll to earlier or later parts of the program.

3. Lines containing previously set breakpoints are shown in high-intensity text.

4. The dialog window is where you enter dialog commands. These are the commands with optional arguments that you can enter at the CodeView prompt (>). You can scroll up or down in this window to view previous dialog commands and command output.

5. The cursor is a thin, blinking line that shows the location at which you can enter commands from the keyboard. You can move the cursor up and down, and place it in either the dialog or display window.

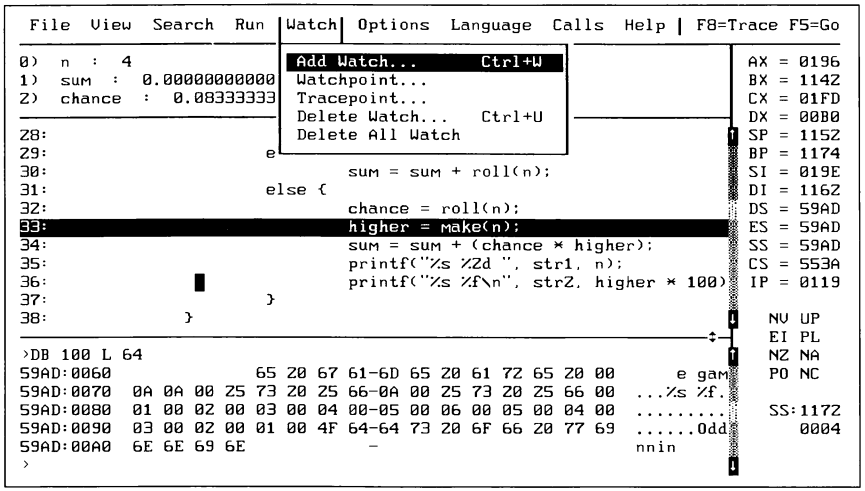


Figure 2.1 Elements of the CodeView Debugging Screen

6. The display/dialog separator line divides the dialog window from the display window.

7. The register window shows the current status of processor registers and flags. It is an optional window that can be opened or closed with one keystroke or with the mouse. If the 386 option is on, a much wider register window is displayed, with 32-bit registers. The register window also displays the effective address at the bottom; the effective address shows the actual location of an operand in physical memory. It is useful when debugging in assembly mode.

8. The scroll bars are the vertical bars on the right side of the screen. Each scroll bar has an up arrow and a down arrow you can use to scroll through the display with a mouse.

9. The optional watch window shows the current status of specified variables or expressions. The watch window appears automatically whenever you create watch statements.

10. The menu bar shows titles of menus and commands that you can activate with the keyboard or the mouse. "Trace" and "Go" represent commands; the other titles are all menus.

11. Menus can be opened by specifying the appropriate title on the menu bar. On the sample screen, the Watch menu has been opened.

12. The menu "highlight" is a reverse-video or colored strip indicating the current selection in a menu. You can move the highlight up or down to change the current selection.

13. The mouse pointer indicates the current position of the mouse. It is shown only if you have a mouse installed on your system.

14. Dialog boxes (not shown) appear in the center of the screen when you choose a menu selection that requires a response. The dialog box prompts you for a response and then it disappears when you enter your answer.

15. Message boxes (not shown) appear in the center of the screen to display errors or other messages.

The Microsoft CodeView debugger screen elements are described in greater detail in the rest of this chapter.

2.1.1 Executing Window Commands with the Keyboard

The most common CodeView debugging commands, and all the commands for managing the CodeView display, are available with window commands. Window commands are one-keystroke commands that can be entered with function keys, CTRL-key combinations, ALT-key combinations, or the direction keys on the numeric keypad.

Most window commands can also be entered with a mouse, as described in Section 2.1.2.1, "Changing the Screen with the Mouse." The window commands available from the keyboard are described by category in Sections 2.1.1.1–2.1.1.4 below.

2.1.1.1 Moving the Cursor with Keyboard Commands

The following keys move the cursor or scroll text up or down in the display or dialog window.

Key	Function (Switch Cursor)
F6	Moves the cursor between the display and dialog windows. If the cursor is in the dialog window when you press F6, it will move to its previous position in the display window. If the cursor is in the display window, it will move to its previous position in the dialog window.
CTRL+G	Makes the size of the dialog window or display window grow. This works for whichever window the cursor is in. If the cursor is in the display window, the display/dialog separator line will move down one line. If the cursor is in the dialog window, the separator line will move up one line.
CTRL+T	Makes the size of the dialog or display window smaller. This works for whichever window the cursor is in. If the cursor is in the display window, the display/dialog separator line will move up one line. If the cursor is in the dialog window, the separator line will move down one line.
UP ARROW	Moves the cursor up one line in either the display or dialog window.
DOWN ARROW	Moves the cursor down one line in either the display or dialog window.
PGUP	Scrolls up one page. If the cursor is in the display window, the source lines or assembly-language instructions scroll up. If the cursor is in the dialog window, the buffer of commands entered during the session scrolls up. The cursor remains at its current position in the window. The length of a page is the current number of lines in the window.

PGDN	Scrolls down one page. If the cursor is in the display window, the source lines or assembly-language instructions scroll down. If the cursor is in the dialog window, the buffer of commands entered during the session scrolls down. The cursor remains at its current position in the window. The length of a page is the current number of lines in the window.
HOME	Scrolls to the top of the file, first local variable, or beginning of the current command. If the cursor is in the display or locals window, the text scrolls to the start of the source file, program instructions, or local variables. If the cursor is in the dialog window and you are currently entering a command, the cursor moves to the beginning of the line, right after the prompt.
CTRL+HOME	Scrolls to the top of the file, first local variable, or beginning of the command buffer. This key produces the same effect that the HOME key does, except that in the dialog window it moves the cursor to the beginning of the command buffer. The top of the command buffer may be blank if you have not yet entered enough commands to fill the buffer. The cursor remains at its current position in the window.
END	Scrolls to the bottom of the file, last local variable, or the end of the current command. If the cursor is in the display or locals window, the text scrolls to the end of the source file, program instructions, or local variables. If the cursor is in the dialog window and you are entering a command, the cursor moves to the end of the command.
CTRL+END	Scrolls to the bottom of the file, to the last local variables, or to the end of the command buffer. This key produces the same effect that the END key does, except that in the dialog window, this key moves the cursor to the end of the command buffer.

2.1.1.2 Changing the Screen with Keyboard Commands

The following keys change the screen or switch to a different screen.

Key	Function
F1	Displays initial on-line Help screen. The Help system is discussed in Section 2.1.4. You can also take advantage of the Help system by using the Help menu, as mentioned in Section 2.1.3.9.
F2	Toggles the register window. The window disappears if present, or appears if absent. You can also toggle the register window with the Register selection from the View menu, as described in Section 2.1.3.2.
F3	Switches between source, mixed, and assembly modes. Source mode shows source code in the display window, whereas assembly mode shows assembly-language instructions. Mixed mode shows both. You can also change modes with the Source, Mixed, and Assembly selections from the View menu, as described in Section 2.1.3.2.
F4	Switches to the output screen. The output screen shows the output, if any, from your program. Press any key to return to the CodeView screen.

2.1.1.3 Controlling Program Execution with Keyboard Commands

The following keys set and clear breakpoints, trace through your program, or execute to a breakpoint.

Key	Function
F5	Executes to the next breakpoint or to the end of the program if no breakpoint is encountered. This keyboard command corresponds to the Go dialog command when it is given without a destination breakpoint argument.

F7

Sets a temporary breakpoint on the line with the cursor, and executes to that line (or to a previously set breakpoint or the end of the program if either is encountered before the temporary breakpoint).

In source mode, if the line does not correspond to code (for example, data declaration or comment lines), the CodeView debugger sounds a warning and ignores the command. This window command corresponds to the Go dialog command when it is given with a destination breakpoint.

F8

Executes a Trace command.

The CodeView debugger executes the next source line in source mode or the next instruction in assembly mode. If the source line or instruction contains a call to a routine or interrupt, the debugger starts tracing through the call (enters the call and is ready to execute the first source line or instruction). This command will not trace into DOS function calls.

F9

Sets a breakpoint or clears a breakpoint on the line with the cursor.

If the line does not currently have a breakpoint, one is set on that line. If the line already has a breakpoint, the breakpoint is cleared. If the cursor is in the dialog window, the CodeView debugger sounds a warning and ignores the command. This window command corresponds to the Breakpoint Set and Breakpoint Clear dialog commands.

F10

Executes the Program Step command.

The debugger executes the next source line in source mode or the next instruction in assembly mode. If the source line or instruction contains a call to a routine or interrupt, the debugger steps over the entire call (executes it to the return) and is ready to execute the line or instruction after the call.

NOTE *You can usually interrupt program execution by pressing either CTRL+BREAK or CTRL+C. These key combinations can be used to exit endless loops or to interrupt loops slowed by the Watchpoint or Tracepoint commands (see Chapter 8, "Managing Watch Statements"). CTRL+BREAK or CTRL+C may not work if your program has a special use for one or both of these key combinations. If you have an IBM Personal Computer AT (or an AT-compatible), you can use the SYSTEM-REQUEST key to interrupt execution regardless of your program's use of CTRL+BREAK and CTRL+C.*

2.1.1.4 Selecting from Menus with the Keyboard

This section discusses how to make selections from menus with the keyboard. The effects of the selections are discussed in Section 2.1.3, "Using Menu Selections," below.

The menu bar at the top of the screen has eleven titles: File, View, Search, Run, Watch, Options, Language, Calls, Help, Trace, and Go. The first nine titles are menus, and the last two are commands. The Trace and Go titles are provided primarily for mouse users.

The four steps for opening a menu and making a selection are described below.

1. To open a menu, press ALT and the mnemonic (the first letter) of the menu title. This can be accomplished either by pressing ALT first, releasing the key, and pressing the letter; or you can hold down ALT and then press the letter. For example, press ALT+S to open the Search menu. The menu title is highlighted, and a menu box listing the selections pops up below the title.

   The mnemonic is a single letter that represents the selection. In color displays, this letter is in red; in black-and-white displays, it is in bold. In most cases, but not all, the letter is simply the first letter of the name of the selection. You can type an uppercase or a lowercase letter for the same selection.

2. There are two ways to make a selection from an open menu:

   a. Press the DOWN ARROW key on the numeric keypad to move down the menu. The highlight will follow your movement. When the item you want is highlighted, press ENTER to execute the command.

      You can also press the UP ARROW key to move up the menu. If you move off the top or bottom of the menu, the highlight wraps around to the other end of the menu.

   b. Press the key corresponding to the menu-selection mnemonic.

3. After a selection is made from the menu, one of three things will happen:

   a. For most menu selections, the choice is executed immediately.

   b. The items on the View, Options, and Language menus have small double arrows next to them if the option is on, or no arrows if the option is off. Choosing the item toggles the option. The status of the arrows will be reversed the next time an option is chosen.

   c. Some items require a response. In this case, there is another step in the menu-selection process.

4. If the item you select requires a response, a dialog box opens when you select a menu item. Type your response to the prompt in the box and press ENTER. For example, the Find dialog box asks you to enter a regular expression (see Appendix A for a complete explanation of regular expressions).

   If your response is valid, the command will be executed. If you enter an invalid response, a message box will appear, telling you the problem and asking you to press a key. Press any key to make the message box disappear.

At any point during the process of selecting a menu item, you can press ESCAPE to cancel the menu. While a menu is open, you can press the LEFT ARROW or RIGHT ARROW key to move from one menu to an adjacent menu, or to one of the command titles on the menu bar. Pressing ENTER without entering any characters in response to a message box will also cancel the menu.

2.1.2 Executing Window Commands with the Mouse

The CodeView debugger is designed to work with the Microsoft Mouse (it also works with some compatible pointing devices). By moving the mouse on a flat surface, you can move the mouse pointer in a corresponding direction on the screen. The following terms refer to the way you select items or execute commands with the mouse.

Term	Definition
Point	Move the mouse until the mouse pointer rests on the item you want to select.
Click	Quickly press and release a mouse button while pointing at an item you want to select.
Drag	Press a mouse button while on a selected item, then hold the button down while moving the mouse. The item moves in the direction of the mouse movement. When the item you are moving is where you want it, release the button; the item will stay at that place.

The CodeView debugger uses two mouse buttons. The terms "click Right," "click Left," "click both," and "click either" are sometimes used to designate which buttons to use. When dragging, either button can be used.

2.1.2.1 Changing the Screen with the Mouse

You can change various aspects of the screen display by pointing to one of the following elements and then either clicking or dragging.

Item	Action
Single line separating display and dialog windows	Drag the separator line up to increase the size of the dialog window while decreasing the size of the display window, or drag the line down to increase the size of the display window while decreasing the size of the dialog window. You can eliminate either window completely by dragging the line all the way up or down (providing the cursor is not in the window you want to eliminate).
UP ARROW or DOWN ARROW on the scroll bar	Point and click Left on one of the four arrows on the scroll bars to scroll up or down. If you are in the display window, source code scrolls up or down. If you are in the dialog window, the buffer containing dialog commands entered during the session scrolls up or down. Click Left to scroll up or down just one line at a time. Press Left and hold it down in order to scroll continuously. Continuous scrolling is easier to use when you want to scroll more than a couple of lines. The scrolling stops as soon as you release the mouse button.
Scroll bar elevator	Each scroll bar has an "elevator," which is a highlighted rectangle on the bar that can be moved up or down with the mouse. In the display window, the elevator indicates your relative position in the source file; if you are in mixed or assembly mode, the elevator indicates your position in the executable file relative to the instructions that correspond to the source file. You can move quickly through the source file by dragging the display window elevator up or down. In the dialog window, the position of the elevator does not have any significance. To move up one page (either in the display or dialog window), click the scroll bar anywhere above the elevator. To move down a page, click the scroll bar anywhere below the elevator.

2.1.2.2 Controlling Program Execution with the Mouse

By clicking the following mouse items, you can set and clear breakpoints, trace through your program, execute to a breakpoint, or change flag bits.

Item

Action

Source line or instruction

Point and click on a source line in source mode or on an instruction in assembly mode to take one of the following actions:

Button	Result
Click Left	If the line under the mouse cursor does not have a breakpoint, one is set there. If the line already has a breakpoint, the breakpoint is removed. Lines with breakpoints are shown in high-intensity text.
Click Right	A temporary breakpoint is set on the line, and the CodeView debugger executes until it reaches the line (or until it reaches a previously set breakpoint or the end of the program if either is encountered before the temporary breakpoint).

If you click on a line that does not correspond to code (for example, a declaration or comment), the CodeView debugger will sound a warning and ignore the command.

"Trace" on menu bar

Point and click to trace the next instruction. The kind of trace is determined by which button is clicked:

Button	Result
Click Left	The Trace command is executed. The CodeView debugger executes the next source line in source mode or the next instruction in assembly mode. If the source line or instruction contains a call to a routine or interrupt, the debugger

		starts tracing through the call (it enters the call and is ready to execute the first source line or instruction). This command will not trace into DOS function calls.
	Click Right	The Program Step command is executed. The debugger executes the next source line in source mode, or the next instruction in assembly mode. If the source line or instruction contains a call to a routine or interrupt, the CodeView debugger steps over the entire call (it executes the call to the return) and is ready to execute the line or instruction after the call.
	These two commands are different only if the current location is the start of a procedure, an interrupt, or a call.	
"Go" on menu bar	Point and click either button to execute to the next breakpoint, or to the end of the program if no breakpoints are encountered.	
Flag in register window	Point to a flag name and click either button to reverse the flag. If the flag bit is set, it will be cleared; if the flag bit is cleared, it will be set. The flag name is changed on the screen to match the new status. If you are using color mode, the color of the flag mnemonic will also change. This command can only be used when the register window is open. Use the command with caution since changing flag bits can change program execution at the lowest level.	

NOTE *You can usually interrupt program execution by pressing either CTRL+BREAK or CTRL+C. See the note in Section 2.1.1.3, "Controlling Program Execution with Keyboard Commands," for more information.*

2.1.2.3 Selecting from Menus with the Mouse

This section discusses how to make selections from menus with the mouse. The effect of each selection is discussed in Section 2.1.3, "Using Menu Selections."

The menu bar at the top of the screen has eleven titles: File, View, Search, Run, Watch, Options, Language, Calls, Help, Trace, and Go. The first nine are menus, and the last two are commands that you can execute by clicking with the mouse. The five steps for opening a menu and making a selection are described below:

1. To open a menu, point to the title of the menu you want to select.
2. With the mouse pointer on the title, press and hold down either mouse button. The selected title is highlighted and a menu box with a list of selections pops up below the title.
3. Press and drag the mouse toward you. The highlight follows the mouse movement. You can move the highlight up or down in the menu box.

   If you move off the box, the highlight will disappear. However, as long as you do not release the button, you can move the pointer back onto the menu to make the highlight reappear.
4. When the selection you want is highlighted, release the mouse button.

   When you release the button, the menu selection is executed. One of three things will happen:

   a. For most menu selections, the choice is executed immediately.

   b. The items on the View, Options, and Language menus have small double arrows next to them if the option is on, or no arrows if the option is off. Choosing the item toggles the option. The status of the arrows on a chosen item will appear reversed the next time you open the menu.

   c. Some items require a response. In this case, there is another step in the menu-selection process.
5. If the item you select requires a response, a dialog box with a prompt appears. Type your response and press ENTER or a mouse button. For example, if you select Find, the prompt will ask you to enter a regular expression (see Section 2.1.3.3, "The Search Menu," or Appendix A for an explanation of regular expressions).

   If your response is valid, the command will be executed. If you enter an invalid response in the dialog box, a message box will appear telling you the problem and asking you to press a key. Press any key or click a mouse button to make the message box disappear.

   Also, if you press ENTER without entering any characters, the message box will disappear.

There are several shortcuts you can take when selecting menu items with the mouse. If you change your mind and decide not to select an item from a menu, just move off the menu and release the mouse button—the menu will disappear. You can move from one menu to another by dragging the pointer directly from the current menu to the title of the new menu.

2.1.3 Using Menu Selections

This section describes the selections on each of the CodeView menus. These selections can be made with the keyboard, as described in Section 2.1.1, or with the mouse, as described in Section 2.1.2.

Note that although the Trace and Go commands appear on the menu bar, they are not menus. These titles are provided primarily for mouse users.

2.1.3.1 The File Menu

The File menu includes selections for working on the current source or program file. The File menu is shown in Figure 2.2, and the selections are explained below.

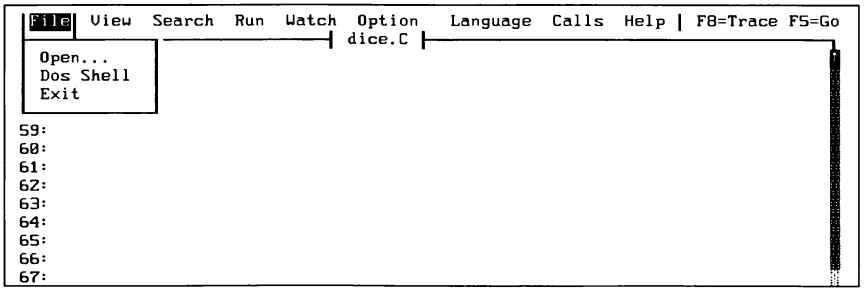


Figure 2.2 The File Menu

Selection	Action
Open	Opens a new file. When you make this selection, a dialog box appears asking for the name of the new file you want to open. Type the name of a source file, an include file, or any other text file. The text of the new file replaces the current contents of the display window

(if you are in assembly mode, the CodeView debugger will switch to source mode). When you finish viewing the file, you can reopen the original file. The last location and breakpoints will still be marked when you return.

You may not need to open a new file to see source files for a different module of your program. The CodeView debugger automatically switches to the source file of a module when program execution enters that module. Although switching source files is never necessary, it may be desirable if you want to set breakpoints or execute to a line in a module not currently being executed.

Note that if the debugger cannot find the source file when it switches modules, a dialog box appears asking for a file specification for the source file. You can either enter a new file specification if the file is in another directory, or press ENTER if no source file exists. If you press ENTER, the module can only be debugged in assembly mode.

DOS Shell

Exits to a DOS shell. This brings up the DOS screen, where you can execute DOS commands or executable files. To return to the CodeView debugger, type `exit` at the DOS command prompt. The CodeView screen reappears with the same status it had when you left it.

The Shell Escape command works by saving the current processes in memory and then executing a second copy of COMMAND.COM. This requires more than 200K of free memory, since the debugger, COMMAND.COM, symbol tables, and the debugged program must all be saved in memory. If you do not have enough memory to execute the Shell Escape command, an error message appears. Even if you have enough memory to execute the command, you may not have enough memory left to execute large programs from the shell.

The Shell Escape command does not work under certain conditions. See Section 11.7 for additional information.

Exit

Terminates the debugger and returns to DOS.

2.1.3.2 The View Menu

The View menu includes selections for switching between source and assembly modes, and for switching between the debugging screen and the output screen. The corresponding function keys for menu selection are shown on the right side of the menu where appropriate. The View menu is shown in Figure 2.3, and the selections are explained below.

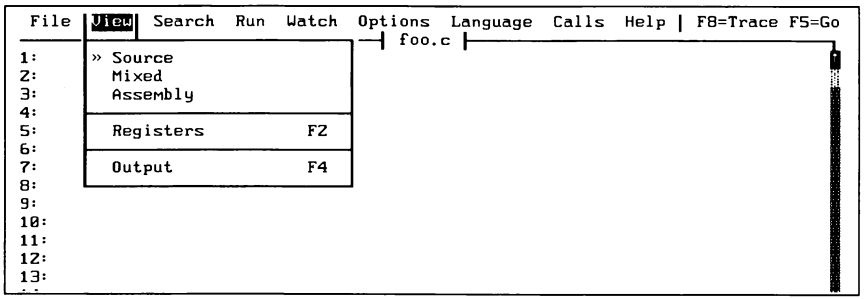


Figure 2.3 The View Menu

NOTE *The terms "source mode" and "assembly mode" apply to Microsoft Macro Assembler programs as well as to high-level-language programs. Source mode used with assembler programs shows the source code as originally written, including comments and directives. Assembly mode displays unassembled machine code, without symbolic information.*

The use of one mode or another affects Trace and Program Step commands, as explained in Chapter 5, "Executing Code."

At all times exactly one of the following selections will have a small double arrow to the left of the name: Source, Mixed, and Assembly. This arrow indicates which of the three display modes is in use. If you select a mode when you are already in that mode, the selection will be ignored. The Registers selection may or may not have a double arrow to the left, depending on whether or not the register window is being displayed.

Selection	Action
Source	Changes to source mode (showing source lines only).
Mixed	Changes to mixed mode (showing both unassembled machine code and source lines).
Assembly	Changes to assembly mode (showing only unassembled machine code).

Registers	Selecting this option will toggle the register window on and off. You can also turn the register on and off by pressing F2.
Output	Selecting this option will display the output screen. The entire CodeView display will temporarily disappear, but come back as soon as you press any key. The Output command can also be selected with F4.

2.1.3.3 The Search Menu

The Search menu includes selections for searching through text files for text strings and searching executable code for labels. The Search menu is shown in Figure 2.4, and the selections are explained below.

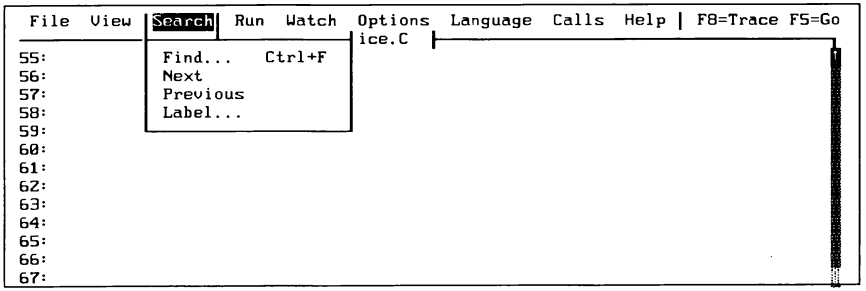


Figure 2.4 The Search Menu

Selection	Action
Find	Searches the current source file or other text file for a specified regular expression. (This selection can also be made without pulling down a menu by pressing CTRL+F.) When you make this selection, a dialog box opens asking you to enter a regular expression. Type the expression you want to search for and press ENTER. The CodeView debugger starts at the current or most recent cursor position in the display window and searches for the expression. If your entry is found, the cursor moves to the first source line containing the expression. If you are in assembly mode, the debugger automatically switches to source mode when the expression is found. If the entry is not found, a message box opens, telling you the problem and asking you to press a key (you can also click a mouse button) to continue.

Regular expressions are a method of specifying variable text strings. This method is similar to the DOS method of using wild cards in file names. Regular expressions are explained in detail in Appendix A of this manual.

You can use the Search selections without understanding regular expressions. Since text strings are the simplest form of regular expressions, you can simply enter a string of characters as the expression you want to find. For example, you could enter `count` if you wanted to search for the word "count."

The following characters have a special meaning in regular expressions: backslash (\), asterisk (*), left bracket ([), period (.), dollar sign ($), and caret (^). In order to find strings containing these characters, you must precede the characters with a backslash; this cancels their special meanings.

For example, the periods in FORTRAN relational and logical operators must be preceded by backslashes. You would use `\.EQ` to find the `.EQ` operator. With C, you would use `\*ptr` to find `*ptr`; with BASIC, you would use `NAME\$` to find `NAME$`.

The Case Sense selection from the Options menu has no effect on searching for regular expressions.

Next

Searches for the next match of the current regular expression.

This selection is meaningful only after you have used the Search command to specify the current regular expression. If the CodeView debugger searches to the end of the file without finding another match for the expression, it wraps around and starts searching at the beginning of the file.

Previous

Searches for the previous match of the current regular expression.

This selection is meaningful only after you have used the Search command to specify the current regular expression. If the debugger searches to the beginning of the file without finding another match for the expression, it wraps around and starts searching at the end of the file.

Label	Searches the executable code for an assembly-language label. If the label is found, the cursor moves to the instruction containing the label. If you start the search in source mode, the debugger will switch to assembly mode to show a label in a library routine or an assembly-language module.

2.1.3.4 The Run Menu

The Run menu includes selections for running your program. The Run menu is shown in Figure 2.5, and the selections are explained below.

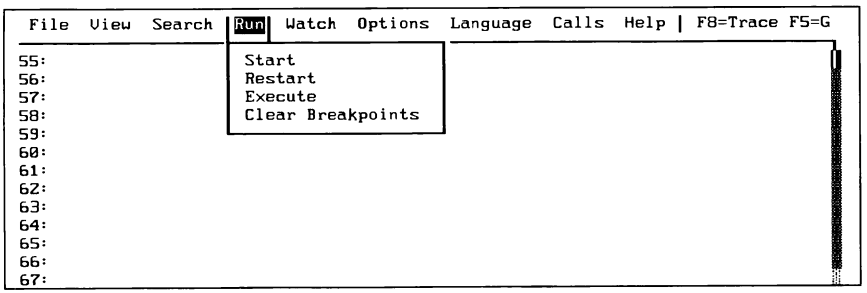


Figure 2.5 The Run Menu

Selection	Action
Start	Starts the program from the beginning and runs it. Any previously set breakpoints or watch statements will still be in effect. The CodeView debugger will run your program from the beginning to the first breakpoint, or to the end of the program if no breakpoint is encountered. This has the same effect as selecting Restart (see the next selection), then entering the Go command.
Restart	Restarts the current program, but does not begin executing it. You can debug the program again from the beginning. Any previously set breakpoints or watch statements will still be in effect.
Execute	Executes from the current instruction. This is the same as the Execute dialog command (**E**). To stop execution, press any key or a mouse button.

Clear Breakpoints	Clears all breakpoints. This selection may be convenient after selecting Restart if you don't want to use previously set breakpoints. Note that watch statements are not cleared by this command.

NOTE *Although the Start and Restart selections retain breakpoints along with pass count and arguments, any instructions entered with the Assemble command will be overwritten by the original program.*

2.1.3.5 The Watch Menu

The Watch menu includes selections for managing the watch window. Selections on this menu are also available with dialog commands. The Watch menu is shown in Figure 2.6, and the selections are explained below.

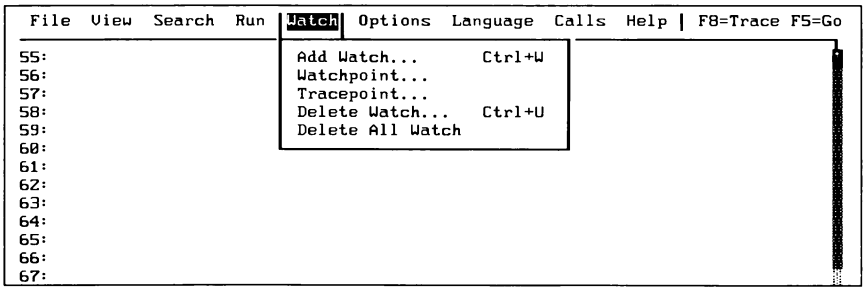


Figure 2.6 The Watch Menu

Selection	Action
Add Watch	Adds a watch-expression statement to the watch window. (This selection can also be made directly, by pressing CTRL+W.) A dialog window opens, asking for the source-level expression (which may be simply a variable) whose value you want to see displayed in the watch window. Type the expression and press ENTER or a mouse button. The statement appears in the watch window in normal text. You cannot specify a memory range to be displayed with the Add Watch selection as with the Watch dialog command.

You can specify the format in which the value will be displayed. Type the expression, followed by a comma and a CodeView format specifier. If you do not give a format specifier, the CodeView debugger displays the value in a default format. See Chapter 6, "Examining Data and Expressions," for more information about format specifiers and the default format. See Section 8.2, "Setting Watch-Expression and Watch-Memory Statements," for more information about the Watch command.

Watchpoint

Adds a watchpoint statement to the window.

A dialog window opens, asking for the source-level expression whose value you want to test. The watch-point statement appears in the watch window in high-intensity text when you enter the expression. A watchpoint is a conditional breakpoint that causes execution to stop when the expression becomes non-zero (true). See Section 8.3, "Setting Watchpoints," for more information.

Tracepoint

Adds a tracepoint statement to the watch window.

A dialog window opens, asking for the source-level expression or memory range whose value you want to test. The tracepoint statement appears in the watch window in high-intensity text when you enter the expression. A tracepoint is a conditional break-point that causes execution to stop when the value of a given expression changes. You cannot specify a memory range to be tested with the Tracepoint selection as you can with the Tracepoint dialog command.

When setting a tracepoint expression, you can specify the format in which the value will be dis-played. After the expression type a comma and a format specifier. If you do not give a format speci-fier, the CodeView debugger displays the value in a default format. See Chapter 6, "Examining Data and Expressions," for more information about format specifiers and default. See Section 8.4, "Setting Tracepoints," for more information about trace-points.

Delete Watch

Deletes a statement from the watch window. (This selection can also be made directly, by pressing CTRL+U.)

	A dialog window opens, showing the current watch statements. If you are using a mouse, move the pointer to the statement you want to delete and click either button. If you are using the keyboard, press the UP ARROW or DOWN ARROW key to move the highlight to the statement you want to delete, then press ENTER.
Delete All Watch	Deletes all statements in the watch window. All watch, watchpoint, and tracepoint statements are deleted, the watch window disappears, and the display window is redrawn to take advantage of the freed space on screen.

2.1.3.6 The Options Menu

The Options menu allows you to set options that affect various aspects of the behavior of the CodeView debugger. The Options menu is shown in Figure 2.7, and the selections are explained below.

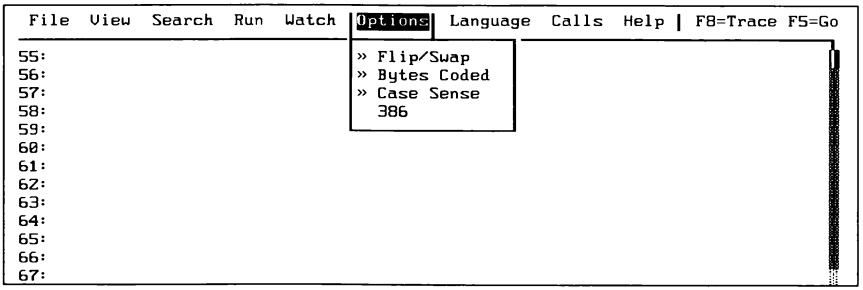


Figure 2.7 The Options Menu

Selections on the Options menu have small double arrows to the left of the selection name when the option is on. The status of the option (and the presence of the double arrows) is reversed each time you select the option. By default, the Flip/Swap and Bytes Coded options are on and the 386 option is off when you start the CodeView debugger. Depending on which language your main program is in, the debugger will automatically turn Case Sense on (if your program is in C) or off (if your program is in another language) when you start debugging.

The selections from the Options menu are discussed below.

Selection	Action
Flip/Swap	When on (the default), screen swapping or screen flipping (whichever the debugger was started with) is active; when off, swapping or flipping is disabled.

Turning off swapping or flipping makes the screen scroll more smoothly. You will not see the program flip or swap each time you execute part of the program. This option has no effect if neither swapping nor flipping was selected during start-up.

WARNING *Any time your program writes to the screen, make sure flipping or swapping is on. If swapping and flipping are off, your program writes the output at the location of the cursor. The CodeView debugger detects the screen has changed and redraws the screen, thus destroying the program output. An error message is also displayed:*

```
Flip/Swap option off -application output lost.
```

Bytes Coded

When on (the default), the instructions, instruction addresses, and the bytes for each instruction are shown; when off, only the instructions are shown.

This option affects only assembly mode. The following display shows the appearance of sample code when the option is off:

```
27:              name = gets(namebuf);
          LEA        AX,Word Ptr [namebuf]
          PUSH       AX
          CALL       _gets (03E1)
          ADD        SP,02
          MOV        Word Ptr [name],AX
```

The following display shows the appearance of the same code when the option is on:

```
27:              name = gets(namebuf);
32AF:003E 8D46DE          LEA        AX,Word Ptr [namebuf]
32AF:0041 50              PUSH       AX
32AF:0042 E89C03          CALL       _gets (03E1)
32AF:0045 83C402          ADD        SP,02
32AF:0048 8946DA          MOV        Word Ptr [name],AX
```

Case Sense

When the selection is turned on, the CodeView debugger assumes that symbol names are case sensitive (each lowercase letter is different from the corresponding uppercase letter); when off, symbol names are not case sensitive.

This option is on by default for C programs and off by default for FORTRAN, BASIC, and assembly programs. You will probably want to leave the option in its default setting.

386 (DOS Only)

When on, the register window will display the registers in the wider, 386 format. Furthermore, this option will enable you to assemble and execute instructions that reference 32-bit registers. If the 386 option is not on, any data stored in the high-order word of a 32-bit register will be lost.

To use this option, you should have a 386 processor running in 386 mode. If you do not have a 386 processor, the debugger will respond with the message, `CPU is not an 80386`, and leave the option turned off.

2.1.3.7 The Language Menu

The Language menu allows you to either select the expression evaluator or instruct the CodeView debugger to select it for you automatically. The Language menu is shown in Figure 2.8, and the selections are explained below.

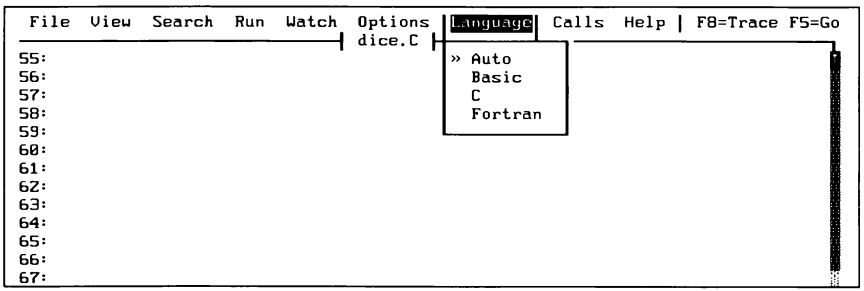


Figure 2.8 The Language Menu

As with the Options menu, the selection on is marked by double arrows. Unlike the Options menu, however, exactly one item (and no more) on the Language menu is selected at any given time.

The Auto selection causes the debugger to select automatically the expression evaluator each time a new source file is loaded. The debugger will examine the extension of the source file in order to determine which expression evaluator to select. The Auto selection will use the C expression evaluator if the current source file does not have a .BAS, .F, .FOR, or .PAS extension.

If you change to a source module with an .ASM extension, Auto will cause the debugger to select the C expression evaluator, but not all of the C defaults will be used; system radix will be hexadecimal, case sensitivity will be turned off, and the register window will be displayed.

When a language expression evaluator is selected, the debugger uses that evaluator, regardless of what kind of program is being debugged.

2.1.3.8 The Calls Menu

The Calls menu is different from other menus in that its contents and size change depending on the status of your program. The Calls menu is shown in Figure 2.9.

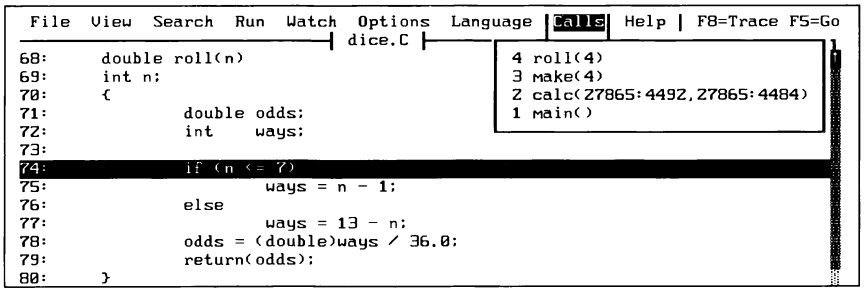


Figure 2.9 The Calls Menu

The mnemonic for each item in the Calls menu is a number. Type the number displayed immediately to the left of a routine in order to select it. You can also use the UP ARROW or DOWN ARROW key to move to your selection, and then press ENTER. You can use the mouse to select from the Calls menu as well.

The effect of making a selection from the Calls menu is to view a routine. The cursor will go to the line at which the selected routine was last executing. For example, selecting `main` in the example above will cause CodeView to display `main`, at the point at which `main` made a call to `calc` (the function immediately above it). Note that selecting a routine from the Calls menu does not (by itself) affect program execution. It simply provides a convenient way to view previously called routines.

It is not required that one of the routines be selected. The Calls menu is useful for viewing the list of previously called routines.

The Calls menu shows the current routine and the trail of routines from which it was called. The current routine is always at the top. The routine from which the current routine was called is directly below. Other active routines are shown in the reverse order in which they were called. With C and FORTRAN programs, the bottom routine should always be **main**. (The only time **main** will not be the bottom routine is when you are tracing through the standard library's start-up or termination routines.)

The current value of each argument, if any, is shown in parentheses following the routine. The menu expands to accommodate the arguments of the widest routine. Arguments are shown in the current radix (the default is decimal). If there are more active routines than will fit on the screen, or if the routine arguments are too wide, the display will expand to both the left and right. The Stack Trace dialog command (**K**) also shows all the routines and arguments.

NOTE *If you are using the CodeView debugger to debug assembly-language programs, routines will be shown in the Calls menu only if they use one of the Microsoft calling conventions. These calling conventions are explained in the Microsoft Mixed-Language Programming Guide.*

2.1.3.9 The Help Menu

The Help menu lists the major topics in the on-line Help. For Help, open the Help menu and select the topic you want to view.

Each topic may have any number of subtopics. You must go to the major topic first. Information on how to move around within the help system is provided in the next section.

The bottom selection on the Help menu is the About command. When you make this selection, the debugger displays a small box at the center of the screen that gives the time, the name of the product, and the version number.

2.1.4 Using On-Line Help

The CodeView on-line Help system uses tree-structured menus to give you quick access to help screens on a variety of subjects. Help uses a combination of menu access and sequentially linked screens, as explained below.

The Help file is called CV.HLP. It must be present in the current directory or in one of the directories specified with the DOS PATH command. If the Help file is not found, the CodeView debugger will still operate, but you will not be able to use Help. An error message will appear if you try to use a Help command.

When you request help, either by pressing F1, by using the **H** dialog command, or by selecting the Help menu, the first help screen appears. You can select Next and Previous buttons to page through the screens. The screens are arranged in a circular fashion, so that selecting Next on the last screen gets you to the first screen. Select the Cancel button to return to the CodeView screen. Pressing the PGDN, PGUP, and ESC keys achieves the same results as selecting Next, Previous, and Cancel, respectively, with the mouse.

You can enter Help at a particular topic by selecting the topic from the Help menu. Once in Help, use Next and Previous to page to other screens.

2.2 Using Sequential Mode

Sequential mode is required if you have neither an IBM Personal Computer nor a closely compatible computer. In sequential mode, the CodeView debugger works much like its forerunner, the Microsoft Symbolic Debug Utility (SYMDEB) and the DOS DEBUG utility. Sequential mode is also useful when you are using redirected CodeView input and output.

In sequential mode, the CodeView debugger's input and output always move down the screen from the current location. When the screen is full, the old output scrolls off the top of the screen to make room for new output appearing at the bottom. You can never return to examine previous commands once they scroll off, but in many cases, you can reenter the command to put the same information on the screen again.

Most window commands cannot be used in sequential mode. However, the following function keys, which are used as commands in window mode, are also available in sequential mode.

Command	Action
F1	Displays a command-syntax summary.
F2	Displays the registers. This is equivalent to the Register (**R**) dialog command.
F3	Toggles between source, mixed, and assembly modes. Pressing this key will rotate the mode between source, mixed, and assembly. You can achieve the same effect by using the Set Assembly (**S-**), Set Mixed (**S&**), and Set Source (**S+**) dialog commands.
F4	Switches to the output screen, which shows the output of your program. Press any key to return to the CodeView debugging screen. This is equivalent to the Screen Exchange (\) dialog command.
F5	Executes from the current instruction until a breakpoint or the end of the program is encountered. This is equivalent to the Go dialog command (**G**) with no argument.
F8	Executes the next source line in source mode, or the next instruction in assembly mode. If the source line or instruction contains a function, procedure, or interrupt call, the CodeView debugger executes the first source line or instruction of the call and is ready to execute the next source line or instruction within the call. This is equivalent to the Trace (**T**) dialog command.
F9	Sets or clears a breakpoint at the current program location.

If the current program location has no breakpoint, one is set. If the current location has a breakpoint, it is removed. This is equivalent to the Breakpoint Set (**BP**) dialog command with no argument.

F10

Executes the next source line in source mode or the next instruction in assembly mode.

If the source line or instruction contains a function, procedure, or interrupt call, the call is executed to the end, and the CodeView debugger is ready to execute the line or instruction after the call. This is equivalent to the Program Step (**P**) dialog command.

The CodeView Watch (**W**), Watchpoint (**WP**), and Tracepoint (**TP**) commands work in sequential mode, but since there is no watch window, the watch statements are not shown. You must use the Watch List command (**W**) to examine watch statements and watch values. See Chapter 8 for information on watch statement commands.

All the CodeView commands that affect program operation (such as Trace, Go, and Breakpoint Set) are available in sequential mode. Any debugging operation done in window mode can also be done in sequential mode.

CHAPTER 3

Using Dialog Commands

CodeView dialog commands can be used in sequential mode or from the dialog window. In sequential mode, they are the primary method of entering commands. In window mode, dialog commands are used to enter commands that require arguments or do not have corresponding window commands.

Many window commands have duplicate dialog commands. Generally, the window version of a command is more convenient, but the dialog version is more powerful. For example, to set a breakpoint on a source line in window mode, put the cursor on the source line and press F9, or point to the line and click the left mouse button. The dialog version of the Breakpoint command (**BP**) requires more keystrokes, but it allows you to specify an address, a pass count, and a string of commands to be taken whenever the breakpoint is encountered.

The rest of this chapter explains how to enter dialog commands and how to select text on the screen for use with commands.

3.1 Entering Commands and Arguments

Dialog commands are entered at the CodeView prompt (>). Type the command and arguments and press ENTER.

In window mode, you can enter commands whether or not the cursor is at the CodeView prompt. If the cursor is not in the dialog window, it moves to the dialog window as soon as you type text.

3.1.1 Using Special Keys

When entering dialog commands or viewing output from commands, you can use the following special keys:

Key	Action
CTRL+C	Stops the current output or cancels the current command line. For example, if you are watching a long display from a Dump command, you can press CTRL+C to interrupt the output and return to the CodeView prompt. If you make a mistake while entering a command, you can press CTRL+C to cancel the command without executing it. A new prompt appears, and you can reenter the command.
CTRL+S	Pauses during the output of a command. You can press any key to continue output. For example, if you are watching a long display from a Dump command, you can press CTRL+S when a part of the display appears that you want to examine more closely. Then press any key when you are ready for the output to continue scrolling.
BACKSPACE	Deletes the previous character on the command line and moves the cursor back one space. For example, if you make an error while typing a command, you can use the BACKSPACE key to delete the characters back to the error—then retype the remainder of the command.

3.1.2 Using the Command Buffer

In window mode, the CodeView debugger has a command buffer where the last 2 – 4 screens of commands and command output are stored. The command buffer is not available in sequential mode.

When the cursor is in the dialog window, you can scroll up or down to view the commands you have entered earlier in the session. The commands for moving the cursor and scrolling through the buffer are explained in sections 2.1.1.1 and 2.1.2.1.

Scrolling through the buffer is particularly useful for viewing the output from commands, such as Dump or Examine Symbols, whose output may scroll off the top of the dialog window.

If you have scrolled through the dialog buffer to look at previous commands and output, you can still enter new commands. When you type a command, the cursor moves to the end of the command buffer and the text appears on a new line.

When you start the debugger, the buffer is empty except for the copyright message. As you enter commands during the session, the buffer is gradually filled from the bottom to the top. If you have not filled the entire buffer and you press HOME to go to the top of the buffer, you will not see the first commands of the session. Instead you will see blank lines, since there is nothing at the top of the buffer.

3.2 Format for CodeView Commands and Arguments

The CodeView command format is similar to the format of the earlier Microsoft debuggers, SYMDEB and DEBUG. However, some features, particularly operators and expressions, are different. The general format for CodeView commands is shown below:

command [[*arguments*]] [[;*command2*]]

The *command* is a one-, two-, or three-character command name, and *arguments* are expressions that represent values or addresses to be used by the command. The *command* is not case sensitive; any combination of uppercase and lowercase letters can be used. However, *arguments* consisting of source-level expressions may or may not be case sensitive. (For example, C expressions are normally case sensitive; FORTRAN expressions are not. Case sensitivity can be affected by the language selected for expression evaluation.) Usually, the first *argument* can be placed immediately after *command* with no space separating the two fields.

The number of arguments required or allowed with each command varies. If a command takes two or more arguments, you must separate the arguments with spaces. A semicolon (;) can be used as a command separator if you want to specify more than one command on a line.

Examples

```
DB 100 200        ;* Example 1

>U Label1         ;* Example 2, C variable as argument

>U SUM            ;* Example 3, FORTRAN variable as argument

>U SUM; DB        ;* Example 4, multiple commands
```

In Example 1, `DB` is the first command (for the Dump Bytes command). The arguments to the command are `100` and `200`. The second command on this line is the Comment command (`*`). A semicolon is used to separate the two commands. The Comment command is used throughout the rest of the manual to number examples.

In Example 2, `U` is the first command (for the Unassemble command), and the C-language variable `Label1` is a command argument.

In Example 3, `U` is again the first command (for the Unassemble command), and the FORTRAN variable `SUM` is a command argument.

Example 4 consists of three commands, separated by semicolons. The first is the Unassemble command (`U`) with the FORTRAN variable `SUM` as an argument. The second is the Dump Bytes command (`DB`) with no arguments. The third is the Comment command (`*`).

3.3 Selecting Text for Use with Commands

If you run CodeView in window mode, you can select text on the screen and use this same text as a command. This technique is useful for reusing a dialog command that is not among the last 20. Any text that appears in any window can be selected.

To select and use text onscreen, follow these steps:

1. Select text with either the mouse or keyboard.

   To select text with the mouse, move the mouse cursor to the beginning of the desired text, hold the left mouse button down and drag the mouse to the left. When you have dragged the mouse to the end of the desired text, release the button.

   To select text with the keyboard, move the cursor to the beginning of the desired text, hold the SHIFT key down, and move the cursor to the right. When the cursor is at the end of the desired text, release the SHIFT key and press ENTER.

2. To use the selected text, press INS.

3. Edit the command if desired, and press ENTER to execute.

CHAPTER 4

CodeView Expressions

4

CodeView command arguments are expressions that can include symbols, constant numbers, operators, and registers. Arguments can be simple machine-level expressions that directly specify an address or range in memory, or they can be source-level expressions that correspond to operators and symbols used in Microsoft C, FORTRAN, BASIC, or the Macro Assembler. For each high-level language (C, FORTRAN, and BASIC), CodeView has an expression evaluator that computes the value of source-level expressions.

Each of the three expression evaluators has a different set of operators and rules of precedence. However, the basic syntax for registers, addresses, and line numbers is the same regardless of the language. You can always change the expression evaluator. If you specify a language other than the one used in the source file, the expression evaluator will still recognize your program symbols—if possible. (C and FORTRAN, however, will not accept BASIC type tags.) If you are debugging an assembly routine called from BASIC or FORTRAN, you may want to choose the language of the main program rather than C, the default for assembly programs.

If the Auto option is on, the debugger examines the file extension of each new source file you trace through. Both C and assembly modules cause the debugger to select C as the expression evaluator.

This chapter first deals with the expressions specific to each language. Line-number expressions are presented next; they work the same way regardless of the language. Then, register and address expressions are described. Generally, these do not have to be mastered unless you are doing assembly-level debugging. Finally, the chapter describes how to switch the expression evaluator.

4.1 C Expressions

The C expression evaluator uses a subset of the most commonly used C operators. It also supports the colon operator (**:**), which is described in Section 4.6.2, "Addresses," and the three memory operators (**BY**, **WO**, and **DW**), which are discussed in Section 4.7. The memory operators are primarily useful for debugging assembly source code. The CodeView C expression operators are listed in Table 4.1 in order of precedence. The superscripts a, b, and c refer to explanatory footnotes.

Table 4.1 CodeView C Expression Operators

Precedence	Operators
(Highest)	
1	( ) [ ] -> .
2	! ~ –[a] (*type*) ++ —*[b] &[c] **sizeof**
3	*[b] / % :
4	+ –[a]
5	< > <= >=
6	== !=
7	&&
8	||
9	= += –= *= /= %=
10	**BY WO DW**
(Lowest)	

[a] The minus sign with precedence 2 is the *unary minus* indicating the sign of a number; the minus sign with precedence 4 is a *binary minus* indicating subtraction.

[b] The asterisk with precedence 2 is the pointer operator; the asterisk with precedence 3 is the multiplication operator.

[c] The ampersand with precedence 2 is the address-of operator. The ampersand as a bitwise **AND** operator is not supported by the CodeView debugger.

See the *Microsoft C Compiler Language Reference* for a description of how C operators can be combined with identifiers and constants to form expressions. With the C expression evaluator, the period (.) has its normal use as a member selection operator, but it also has an extended use as a specifier of local variables in parent functions. The syntax is shown below:

function.variable

The *function* must be a high-level-language function, and the *variable* must be a local variable within the specified function. The *variable* cannot be a register

variable. For example, you can use the expression `main.argc` to refer to the local variable `argc` when you are in a function that has been called by `main`.

The *type* operator (used in type casting) can be any of the predefined C types. The CodeView debugger limits casts of pointer types to one level of indirection. For example, `(char *)sym` is accepted, but `(char **)sym` is not.

When a C expression is used as an argument with a command that takes multiple arguments, the expression should not have any internal spaces. For example, `count+6` is allowed, but `count + 6` may be interpreted as three separate arguments. Some commands (such as the Display Expression command) do permit spaces in expressions.

4.1.1 C Symbols

Syntax

name

A symbol is a name that represents a register, a segment address, an offset address, or a full 32-bit address. At the C source level, a symbol is a variable name or the name of a function. Symbols (also called identifiers) follow the naming rules of the C compiler. Although CodeView command letters are not case sensitive, symbols given as arguments are case sensitive (unless you have turned off case sensitivity with the Case Sense selection from the Options menu).

In assembly-language output or in output from the Examine Symbols command, the CodeView debugger displays some symbol names in the object-code format produced by the Microsoft C Compiler. This format includes a leading underscore. For example, the function `main` is displayed as `_main`. Only global labels (such as procedure names) are shown in this format. You do not need to include the underscore when specifying such a symbol in CodeView commands. Labels within library routines are sometimes displayed with a double underscore (`__chkstk`). You must use two leading underscores when accessing these labels with CodeView commands.

4.1.2 C Constants

Syntax

digits	Default radix
0*digits*	Octal radix
0x*digits*	Hexadecimal radix
0n*digits*	Decimal radix

Numbers used in CodeView commands represent integer constants. They are made up of octal, hexadecimal, or decimal digits, and are entered in the current

input radix. The C-language format for entering numbers of different radixes can be used to override the current input radix.

The default radix for the C expression evaluator is decimal. However, you can use the Radix command (**N**) to specify an octal or hexadecimal radix, as explained in Section 11.3, "Radix Command."

If the current radix is 16 (hexadecimal) or 8 (octal), you can enter decimal numbers in the special CodeView format **0n***digits*. For example, you would enter 21 decimal as `0n21`.

With radix 16, it is possible to enter a value or argument that could be interpreted either as a symbol or as a hexadecimal number. The CodeView debugger resolves the ambiguity by searching first for a symbol (identifier) with that name. If no symbol is found, the debugger interprets the value as a hexadecimal number. If you want to enter a number that overrides an existing symbol, use the hexadecimal format (**0x***digits*).

For example, if you enter `abc` as an argument when the program contains a variable or function named `abc`, the CodeView debugger interprets the argument as the symbol. If you want to enter `abc` as a number, enter it as `0xabc`.

Table 4.2 shows how a sample number (63 decimal) would be represented in each radix.

Table 4.2 C Radix Examples

Input Radix	Octal	Decimal	Hexadecimal
8	`77`	`0n63`	`0x3F`
10	`077`	`63`	`0x3F`
16	`077`	`0n63`	`3F`

4.1.3 C Strings

Syntax

"*null-terminated-string*"

Strings can be specified as expressions in the C format. You can use C escape characters within strings. For example, double quotation marks within a string are specified with the escape character `\"`.

Example

```
EA message "This \"string\" is okay."
```

The example uses the Enter ASCII command (**EA**) to enter the given string into memory starting at the address of the variable `message`.

4.2 FORTRAN Expressions

The FORTRAN expression evaluator uses a subset of the most commonly used FORTRAN operators. It also supports two additional operators, the period (.) and the colon (:). A number of FORTRAN intrinsic functions, which are listed in Section 4.2.4, are also supported. FORTRAN function calls are permitted, but statement function names and COMMON block names are not. (Note that these limitations only apply to the arguments of CodeView commands. They do not apply to the source program, which can contain any valid FORTRAN expression.)

The CodeView FORTRAN operators are listed in Table 4.3 in order of precedence.

Table 4.3 CodeView FORTRAN Operators

Precedence	Operators
(Highest)	
1	()
2	. :
3	Unary + -
4	* /
5	Binary + -
6	.LT. .LE. .EQ. .NE. .GT. .GE.
7	.NOT.
8	.AND.
9	.OR.
10	.EQV. .NEQV.
11	=
(Lowest)	

The FORTRAN expression evaluator does not support the character concatenation operator (//) or the exponentiation operator (**). Relational operators are not supported for string variables or constants.

The order and precedence with which the CodeView debugger evaluates FORTRAN expressions are the same as in the Microsoft FORTRAN language. See Chapter 1, "Elements of FORTRAN" of the *Microsoft FORTRAN Reference* for a description of how FORTRAN operators can be combined with symbols and constants to form expressions.

The colon operator (:) may be used when specifying a memory address. It acts as a *segment*:*offset* separator, as described in Section 4.6.2, "Addresses."

In the CodeView debugger, the period (.) has an extended use as a specifier of local variables in parent routines. The syntax is shown below:

routine.variable

The *routine* must be a high-level-language routine and the *variable* must be a local variable within the specified routine. For example, you can use the expression `main.X` to refer to the local variable `X` in the procedure `main` if you are in a routine called by `main`. Note that in this example, `main` refers to the main routine of a FORTRAN or C program. It does not appear in FORTRAN source code.

4.2.1 FORTRAN Symbols

Syntax

name

A symbol is a name that represents a register, a segment address, an offset address, or a full 32-bit address. At the FORTRAN source level, a symbol is simply a variable name or the name of a routine; you do not necessarily need to know what kind of address it represents. When given as arguments, symbols are never case sensitive with the FORTRAN expression evaluator. If you have turned on case sensitivity with the Case Sense selection from the Options menu, it is turned off automatically when a symbol is used.

In assembly-language output or in output from the Examine Symbols command, the CodeView debugger displays some symbol names in the object-code format produced by the Microsoft FORTRAN Optimizing Compiler. This format includes a leading underscore. For example, the main routine in your program is displayed as `_main`. Only global labels (such as procedure names) are shown in this format. You do not need to include the underscore when specifying such a symbol in CodeView commands. Labels within library routines are sometimes displayed with a double underscore (`__chkstk`). You must use leading underscores when accessing these labels with CodeView commands.

4.2.2 FORTRAN Constants

Syntax

digits	Default radix
radix#digits	Specified radix
#digits	Hexadecimal radix

Numbers used in CodeView commands represent integer constants. These constants are entered in the current input radix (base). When you are using the FORTRAN expression evaluator, the debugger will recognize any explicitly specified radix between 2 and 36 inclusive, as in `20#2G`. The FORTRAN

radix specifiers can be used to override the current radix. Note that a hexadecimal number may be entered in two ways. For example, 3F hex could be entered as either `#3F` or `16#3F`. In this manual, the number sign alone is used to indicate hexadecimal numbers.

The default radix for the FORTRAN version of the CodeView debugger is decimal. However, you can use the Radix command (**N**) to specify an octal or hexadecimal radix, as explained in Section 11.3, "Radix Command."

With radix 16, it is possible to enter a value or argument that could be interpreted either as an identifier or as a hexadecimal number. The CodeView debugger resolves the ambiguity by searching first for a symbol (identifier) with that name. If no symbol is found, the debugger interprets the value as a hexadecimal number. If you want to enter a number that overrides an existing symbol, use the hexadecimal format (*#digits*).

For example, if you enter `ABC` as an argument when the program contains a variable or function named `ABC`, the CodeView debugger interprets the argument as the symbol. If you want to enter `ABC` as a number, enter it as `#ABC`.

Table 4.4 shows how a sample number (63 decimal) would be represented in the octal, decimal, and hexadecimal radixes.

Table 4.4 FORTRAN Radix Examples

Input Radix	Octal	Decimal	Hexadecimal
8	`77`	`10#63`	`#3F`
10	`8#77`	`63`	`#3F`
16	`8#77`	`10#63`	`3F`

4.2.3 FORTRAN Strings

Syntax

' string '

Strings can be specified as character expressions in the FORTRAN format. Single quotation marks within a string must be specified by two single quotation marks.

Example

```
EA MESSAGE 'This ''string'' is okay.  '
```

The example above uses the Enter ASCII command (**EA**) to enter the given string into memory, starting at the address of the variable `MESSAGE`. Notice that the string includes embedded single quotation marks and trailing blanks.

4.2.4 FORTRAN Intrinsic Functions

When entering a FORTRAN expression, you may use a limited number of FORTRAN intrinsic functions. The primary use of these functions is to convert a FORTRAN variable or value from one type to another for purposes of calculation. The intrinsic functions recognized by the expression evaluator of the CodeView debugger are listed in Table 4.5. See Chapter 5, "Intrinsic Functions and Additional Procedures" of the *Microsoft FORTRAN Reference* for a complete description of the FORTRAN intrinsic functions.

Table 4.5 FORTRAN Intrinsic Functions Supported by the CodeView Debugger

Name	Definition	Argument Type	Function Type
CHAR(*int*)	Data-type conversion	int	char*
CMPLX(*genA*[,*genB*])	Data-type conversion	int, real, or cmp	cmp8
DBLE(*gen*)	Data-type conversion	int, real, or cmp	dbl
DCMPLX(*genA*[,*genB*])	Data-type conversion	int, real, or cmp	cmp16
DIMAG(*cmp16*)	Imaginary part of cmp16 number	cmp16	dbl
DREAL(*cmp16*)	Data-type conversion	cmp16	dbl
ICHAR(*char*)	Data-type conversion	char	int
IMAG(*cmp*)	Imaginary part of cmp number	cmp	real†
INT(*gen*)	Data-type conversion	int, real, or cmp	int
INT1(*gen*)	Data-type conversion	int, real, or cmp	int1
INT4(*gen*)	Data-type conversion	int, real, or cmp	int4
INTC(*gen*)	Data-type conversion	int, real, or cmp	**INTEGER[C]**
LOCFAR(*gen*)	Segmented address	int, real, or cmp	int4

Table 4.5 *(continued)*

Name	Definition	Argument Type	Function Type
LOCNEAR(*gen*)	Unsegmented address	int, real, or cmp	int2
REAL(*gen*)	Data-type conversion	int, real, or cmp	real4

* The abbreviations used for the different data types in this table are listed in Chapter 5, "Intrinsic Functions and Additional Procedures" of the *Microsoft FORTRAN Reference*.

† If argument is **COMPLEX*8**, function is **REAL*4**. If argument is **COMPLEX*16**, function is **DOUBLE PRECISION**

4.3 BASIC Expressions

The BASIC expression evaluator uses a subset of the most commonly used BASIC operators. It also supports one important BASIC command—the **LET** command—and one operator in addition to the BASIC operators—the colon (**:**). The CodeView BASIC operators are listed in Table 4.6 in order of precedence.

Table 4.6 CodeView BASIC Operators

Precedence	Operators
(Highest)	
1	()
2	. :
3	* /
4	\ **MOD**
5	+ -
6	= <> < > <= >=
7	**NOT**
8	**AND**
9	**OR**
10	**XOR**
11	**EQV**
12	**IMP**
13	**LET** *variable* =
(Lowest)	

The BASIC expression evaluator does not support the exponentiation operator (^). Nor does it support string assignment, the string concatenation operator (+), or any of the relational operators (=, <, >,etc.), when used with strings. However, arrays, records, and user-defined types are all supported.

The order and precedence with which the CodeView debugger evaluates BASIC expressions are the same as in the Microsoft BASIC language. See your BASIC documentation for a description of how BASIC operators can be combined with symbols and constants to form expressions.

The assignment operator **LET** is supported for numerical operations only. When you use **LET** in a BASIC expression, the return value will not be useful. However, an assignment will take place whenever the expression is evaluated. This gives you a convenient way of manipulating data. For example, after the expression `LET A = 5` is evaluated, the variable `A` will contain the value 5. You must use the keyword **LET** to specify assignment; otherwise, the BASIC expression evaluator will interpret the equal sign (=) as a test for equality.

The colon operator (**:**) may be used to specify a memory address. It acts as a *segment*:*offset* separator, as described in Section 4.6.2, "Addresses."

In the CodeView debugger, the period (.) has an extended use as a specifier of local variables in parent routines. The syntax is shown below:

```
routine.variable
```

The `routine` must be a high-level-language routine and the `variable` must be a local variable within the routine.

When a BASIC expression is used as an argument with a command that takes multiple arguments, the expression should not have any internal spaces. For example, `COUNT+6` is allowed, but `COUNT + 6` may be interpreted as three arguments. Some commands (such as the Display Expression command) only take one argument; these commands do permit spaces in expressions.

4.3.1 BASIC Symbols

Syntax

name

A symbol is a name that represents a register, a segment address, an offset address, or a full 32-bit address. At the BASIC source level, a symbol is simply a variable name or the name of a routine; you do not necessarily need to know what kind of address it represents. With the BASIC expression evaluator,

symbols follow the naming rules of the BASIC compiler. In particular, all the type specifiers used in BASIC ($, %, &, !, and #) are accepted by the BASIC expression evaluator. Symbols are never case sensitive to BASIC, whether the Case Sense option is on or not.

4.3.2 BASIC Constants

Syntax

fixed-point-string[[#|!]]	Single or double, fixed-point format
floating-point-string[[#| !]]	Single or double, floating-point format

digits	Integer, default radix
&O*digits*	Octal radix
&*digits*	Alternative octal radix
&H*digits*	Hexadecimal radix

With the BASIC expression evaluator, numbers can be entered as integer, long, single-precision, or double-precision data objects. Constants are formed according to the rules of the Microsoft BASIC Compiler. A single- or double-precision constant must be entered in decimal radix, regardless of the current system radix. To enter a single or double, use the Microsoft BASIC rules for forming fixed and floating point strings.

Integer constants are entered in the system radix and are made up of octal, decimal, or hexadecimal digits. You may override the system radix by using the octal, or hexadecimal prefix. In addition, you can use the & suffix on any integer constant to indicate that the integer is to be stored as a long (four-byte) integer, rather than as a short (two-byte) integer. To enter integers in the decimal format, the system radix must be 10, and you use the default radix. There is no way to enter decimal integers when the system radix is other than 10, unless you switch to another expression evaluator.

The default radix for the BASIC expression evaluator is decimal. However, you can use the Radix command (**N**) to specify an octal or hexadecimal radix, as explained in Section 11.3, "Radix Command." With radix 16, it is possible to enter a value or argument that could be interpreted either as an identifier or as a hexadecimal number. The CodeView debugger resolves the ambiguity by searching first for a symbol (identifier) with that name. If no symbol is found, the debugger interprets the value as a hexadecimal number. If you want to enter a number that overrides an existing symbol, use the hexadecimal format (**&H***digits*).

For example, if you enter `ABC` as an argument when the program contains a variable or function named `ABC`, the CodeView debugger interprets the argument as the symbol. If you want to enter `ABC` as a number, enter it as `&HABC`.

Table 4.7 shows how a sample number (63 decimal) would be represented in the octal, decimal, and hexadecimal radixes.

Table 4.7 BASIC Radix Examples

Input Radix	Octal	Decimal	Hexadecimal
8	77	-	&H3F
10	&O77	63	&H3F
16	&O77	-	3F

4.3.3 BASIC Strings

The BASIC expression evaluator does not allow you to input strings while debugging. However, it does recognize both fixed and variable-length string variables, as defined by the BASIC compiler. (This includes arrays and records of strings.) Expressions that refer to strings will probably be quite simple, because string operators (concatenation and relational operators) are not supported by the BASIC expression evaluator.

By using the Enter Address command, you can enter a string literal at a given address. To use this technique effectively, however, you will need to understand how BASIC handles string variables. For more information, see Chapter 6, "Examining Data and Expressions."

4.3.4 BASIC Intrinsic Functions

When entering a BASIC expression, you may use a limited number of BASIC intrinsic functions. The primary use of these functions is to convert a BASIC variable or value from one type to another for purposes of calculation. The intrinsic functions recognized by the expression evaluator of the CodeView debugger are listed in Table 4.8. See your BASIC compiler manual for a complete description of the BASIC intrinsic functions.

Table 4.8 BASIC Intrinsic Functions Supported by the CodeView Debugger

Name	Definition	Argument Type	Function Type
ASC[1]	ASCII value of first character	string	integer
CDBL	Data-type conversion	numerical expression	double
CINT	Conversion, with rounding	numerical expression	integer
CSGN	Data-type conversion	numerical expression	single

Table 4.8 *(continued)*

Name	Definition	Argument Type	Function Type
CVI	Data-type conversion	two-byte string	integer
CVL	Data-type conversion	four-byte string	long
CVS	Data-type conversion	four-byte string	short
CVD	Data-type conversion	eight-byte string	double
FIX	Conversion, with truncation	numerical expression	integer
INT	Conversion, with truncation	numerical expression	integer
LBOUND(*arr*[,*dim*]**)**	Lowest index of array	array, dimension	integer
UBOUND(*arr*[,*dim*]**)**	Highest index of array	array, dimension	integer
VAL	Numerical value of string	string	integer, long, single, or double
VARPTR	Offset of variable	variable name	integer
VARSEG	Segment of variable	variable name	integer

[1] Except where noted, each of the functions in this table takes exactly one argument of the type indicated in the third column

4.4 Assembly Expressions

The /ZI option, available with Version 5.0 and later of the Microsoft Macro Assembler, provides variable size information for the CodeView debugger. This makes for correct evaluation of expressions derived from assembly code (except with arrays, which are discussed later in this section). If you have an earlier version of the Macro Assembler, you will need to use C type casts to get correct evaluation.

When a program assembles or when the Auto switch is on, source files with an .ASM extension will cause CodeView to select the C expression evaluator. However, the following options will be set differently from the C default options:

- System radix is hexadecimal (not decimal).
- Register window is on.
- Case sense is off.

The C expression evaluator supports the memory operators described in Section 4.7, and generally is the appropriate expression evaluator with which to debug assembly because of its flexibility.

However, you cannot always use the C expression evaluator to specify an expression exactly as it would appear in assembly code. The list below describes the principal differences between assembler syntax and syntax used with the C expression evaluator.

NOTE *The examples below present expressions, not CodeView commands. You can see the results of these expressions by using them as operands for the Display Expression command (?), described in Chapter 6, "Examining Data and Expressions."*

In the following list, examples of assembly source code are shown in the left-hand column. Corresponding CodeView expressions (with the C expression evaluator) are shown in the right-hand column.

1. Register indirection.

   The C expression evaluator does not extend the use of brackets to registers. To refer to the byte, word, or double word pointed to by a register, use the **BY**, **WO**, or **DW** operator.

```
BYTE PTR [bx]                    BY bx
WORD PTR [bp]                    WO bp
DWORD PTR [bp]                   DW bp
```

2. Register indirection with displacement.

   To perform based, indexed, or based-index indirection with a displacement, use either the **BY**, **WO**, or **DW** operator along with addition in a complex expression:

```
BYTE PTR [di+6]                  BY di+6
BYTE PTR [si][bp+6]              BY si+bp+6
WORD PTR [bx][si]                WO bx+si
```

3. Taking the address of a variable.

   Use the ampersand (**&**) to get the address of a variable with the C expression evaluator.

```
OFFSET var                       &var
```

4. The **PTR** operator.

   With the CodeView debugger, C type casts perform the same function as the assembler **PTR** operator.

```
BYTE PTR var                      (char) var
WORD PTR var                      (int) var
DWORD PTR var                     (long) var
```

5. Accessing array elements.

   Accessing arrays declared in assembly code is problematic because the Macro Assembler emits no type information to indicate which variables are arrays. Therefore the CodeView debugger treats an array name like any other variable.

   In C, an array name is equated with the address of the first element. Therefore, if you prefix an array with the address operator (&), the C expression evaluator gives correct results for array operations.

```
string[12]                    (&string)[12]
warray[bx+di]                 (&warray)(bx+di)/2
darray[4]                     (&darray)[1]
```

In the second and third examples above, notice that the indexes used in the assembly source-code expressions differ from the indexes used in the CodeView expressions. This difference is necessary because C arrays are automatically scaled according to the size of elements. In assembly, the program must do the scaling.

4.5 Line Numbers

Line numbers are useful for source-level debugging. They correspond to the lines in source-code files (BASIC, C, FORTRAN, or Macro Assembler). In source mode, you see a program displayed with each line numbered sequentially. The CodeView debugger allows you to use these same numbers to access parts of a program.

Syntax

.[[*filename*:]]*linenumber*

The address corresponding to a source-line number can be specified as *linenumber* prefixed with a period (.). The CodeView debugger assumes that the source line is in the current source file, unless you specify the optional *filename* followed by a colon and the line number.

The CodeView debugger displays an error message if *filename* does not exist, or if no source line exists for the specified number.

Examples

```
V .100
```

The example above uses the View command (V) to display code starting at the source line `100`. Since no file is indicated, the current source file is assumed.

```
V .SAMPLE.FOR:10

>V .EXAMPLE.BAS:22

>V .DEMO.C:301
```

The examples above use **V** to display source code starting at line `10` of `SAMPLE.FOR`, line `22` of `EXAMPLE.BAS`, and line `301` of `DEMO.C`, respectively.

4.6 Registers and Addresses

This section presents alternative ways to refer to objects in memory, including values stored in the processor's registers. Addresses are basic to each of the expression evaluators. A data symbol represents an address in a data segment; a procedure name represents an address in a code segment. All of the syntax in this section can be considered as an extension to the BASIC, C, or FORTRAN expression evaluator.

4.6.1 Registers

Syntax

[[**@**]]*register*

You can specify a register name if you want to use the current value stored in the register. Registers are rarely needed in source-level debugging, but they are used frequently for assembly-language debugging.

When you specify an identifier, the CodeView debugger first checks the symbol table for a symbol with that name. If the debugger does not find a symbol, it checks to see if the identifier is a valid register name. If you want the identifier to be considered a register, regardless of any name in the symbol table, use the "at" sign (**@**) as a prefix to the register name. For example, if your program has a symbol called `AX`, you could specify `@AX` to refer to the AX register. You can avoid this problem entirely by making sure that identifier names in your program do not conflict with register names.

The register names known to the CodeView debugger are shown in Table 4.9. Note that the 32-bit registers are available only if the 386 option is on and if the computer is a 386 machine running in 386 mode.

Table 4.9 Registers

Type	Names			
8-bit high byte	AH	BH	CH	DH
8-bit low byte	AL	BL	CL	DL
16-bit general purpose	AX	BX	CX	DX
16-bit segment	CS	DS	SS	ES
16-bit pointer	SP	BP	IP	
16-bit index	SI	DI		
32-bit general purpose	EAX	EBX	ECX	EDX
32-bit pointer	ESP	EBP		
32-bit index	ESI	EDI		

4.6.2 Addresses

Syntax

[[*segment*:]]*offset*

Addresses can be specified in the CodeView debugger through the use of the colon operator as a *segment*:*offset* connector. Both the *segment* and the *offset* are made up of expressions.

A full address has a *segment* and an *offset*, separated by a colon. A partial address has just an *offset*; a default segment is assumed. The default segment varies, depending on the command with which the address is used. Commands that refer to data (Dump, Enter, Watch, and Tracepoint) use the contents of the DS register. Commands that refer to code (Assemble, Breakpoint Set, Go, Unassemble, and View) use the contents of the CS register.

Full addresses are seldom necessary in source-level debugging. Occasionally they may be convenient for referring to addresses outside the program, such as BIOS (basic input/output system) or DOS addresses.

Examples

```
DB 100
```

In the example above, the Dump Bytes command (**DB**) is used to dump memory starting at offset address `100`. Since no segment is given, the data segment (the default for Dump commands) is assumed.

```
DB ARRAY(COUNT)      ;* FORTRAN/BASIC example
```

In the example above, the Dump Bytes command is used to dump memory starting at the address of the variable `ARRAY(COUNT)`. In C, a similar variable might be denoted as `array[count]`.

```
DB label+10
```

In the example above, the Dump Bytes command is used to dump memory starting at a point 10 bytes beyond the symbol `label`.

```
DB ES:200
```

In the example above, the Dump Bytes command is used to dump memory at the address having the segment value stored in ES and the offset address `200`.

4.6.3 Address Ranges

Syntax

startaddress endaddress
startaddress **L** *count*

A "range" is a pair of memory addresses that bound a sequence of contiguous memory locations.

You can specify a range in two ways. One way is to give the start and end points. In this case, the range covers *startaddress* to *endaddress*, inclusively. If a command takes a range, but if you do not supply a second address, the CodeView debugger usually assumes the default range. Each command has its own default range. (The most common default range is 128 bytes.)

You can also specify a range by giving its starting point and the number of objects you want included in the range. This type of range is called an object range. In specifying an object range, *startaddress* is the address of the first object in the list, **L** indicates that this is an object range rather than an ordinary range, and *count* specifies the number of objects in the range.

The size of the objects is the size taken by the command. For example, the Dump Bytes command (**DB**) has byte objects, the Dump Words command (**DW**) has words, the Unassemble command (**U**) has instructions, and so on.

Examples

```
DB buffer
```

The example above dumps a range of memory starting at the symbol `buffer`. Since the end of the range is not given, the default size (128 bytes for the Dump Bytes command) is assumed.

```
DB buffer buffer+20
```

The example above dumps a range of memory starting at `buffer` and ending at `buffer+20` (the point 20 bytes beyond `buffer`).

```
DB buffer L 20
```

The example above uses an object range to dump the same range as in the previous example. The `L` indicates that the range is an object range, and `20` is the number of objects in the range. Each object has a size of 1 byte, since that is the command size.

```
U funcname-30 funcname
```

The example above uses the Unassemble command (`U`) to list the assembly-language statements starting 30 instructions before `funcname` and continuing to `funcname`.

4.7 Memory Operators

Memory operators return the content of specific locations in memory. They are unary operators that work in the same way regardless of the language selected and return the result of a direct memory operation. They are chiefly of interest to programmers who debug in assembly mode, and are not necessary for high-level debugging.

All of the operators listed in this section are part of the CodeView C expression evaluator and should not be confused with CodeView commands. As operators, they can only build expressions, which in turn are used as arguments in commands.

NOTE *The memory operators discussed in this section are only available with the C expression evaluator and have the lowest precedence of any C operators.*

4.7.1 Accessing Bytes (BY)

You can access the byte at an address by using the **BY** operator. This operator is useful for simulating the **BYTE PTR** operation of the Microsoft Macro Assembler. It is useful for watching the byte pointed to by a particular register.

NOTE *The examples that follow in this section make use of the Display Expression (**?**) Command, which is described in Section 6.1. The **x** format specifier causes the debugger to produce output in hexadecimal.*

Syntax

BY *address*

The result is a short integer that contains the value of the first byte stored at *address*.

Examples

```
? BY sum
101
```

The example above returns the first byte at the address of `sum`.

```
? BY bp+6
42
```

This example returns the byte pointed to by the BP register, with a displacement of 6.

4.7.2 Accessing Words (WO)

You can access the word at an address by using the **WO** operator. This operator is useful for simulating the **WORD PTR** operation of the assembler. It is particularly useful for watching the word pointed to by a particular register, such as the stack pointer.

Syntax

WO *address*

The result is a short integer that contains the value of the first two bytes stored at *address*.

Examples

```
? WO sum
>13120
```

The example above returns the first word at the address of `sum`.

```
? WO sp,x
>2F38
```

This example returns the word pointed to by the stack pointer; the word therefore represents the last word pushed (the "top" of the stack).

4.7.3 Accessing Double Words (DW)

You can access the word at an address by using the **DW** operator. This operator is useful for simulating the **DWORD PTR** operation of the Microsoft Macro Assembler. It is particularly useful for watching the word pointed to by a particular register.

Syntax

DW *address*

The result is a long integer that contains the value of the first four bytes stored at *address*.

NOTE *Be careful not to confuse the **DW** operator with the **DW** command. The operator is only useful for building expressions; it occurs within a CodeView command line, but never at the beginning. The second use of **DW** mentioned above, the Dump Words Command, occurs only at the beginning of a CodeView command line. It displays an entire range of memory (in words, not double words) rather than returning a single result.*

Examples

```
? DW sum
>132120365
```

The example above returns the first double word at the address of `sum`.

```
? DW si,x
>3F880000
```

This example returns the double word pointed to by the SI register.

4.8 Switching Expression Evaluators

The CodeView debugger allows you to specify a particular expression evaluator: BASIC, C, or FORTRAN. You may want to specify the expression evaluator if you are debugging a source module that does not use the standard extension of the source language (such as .C for C, .BAS for BASIC, etc.), or if you want to use a feature of a different language. For example, you might be debugging a C program and want to evaluate a string of binary digits. The FORTRAN expression evaluator accepts base 2, so you might want to switch temporarily to the FORTRAN expression evaluator.

It is normally not necessary to specify the evaluator, even if you are debugging a mixed-language program; the Auto selection changes the expression evaluator for you.

Mouse

To switch expression evaluators with the mouse, open the Language menu and click the appropriate language selection.

Keyboard

To switch expression evaluators with a keyboard command, press ALT+L to open up the Language menu, use the direction keys (or mnemonic letter) to move to the appropriate language, then press RETURN.

Dialog

To switch expression evaluators using a dialog command, enter a command line with the syntax

USE [[*language*]]

where *language* is C, FORTRAN, BASIC, or Auto. The command is not case sensitive, and you can enter the language name in any combination of uppercase and lowercase letters. Entered on a line by itself, **USE** displays the name of the current expression evaluator. The **USE** command always displays the name of the current expression evaluator or the new expression evaluator (if specified).

Examples

```
USE fortran
FORTRAN
```

The example above switches to the FORTRAN expression evaluator.

```
USE
BASIC
```

The example above displays the name of the current expression evaluator, which in this case happens to be BASIC.

CHAPTER 5

Executing Code

Several commands execute code within a program. Among the differences between the commands is the size of step executed by each. The commands and their step sizes are listed below.

- Trace (**T**)

  Executes the current source line in source mode, or the current instruction in assembly mode; traces into routines, procedures, or interrupts

- Program Step (**P**)

  Executes the current source line in source mode, or the current instruction in assembly mode; steps over routines, procedures, or interrupts

- Go (**G**)

  Executes the current program

- Execute (**E**)

  Executes the current program in slow motion

- Restart (**L**)

  Restarts the current program

5.1 Window and Sequential Mode Commands

In window mode, the screen is updated to reflect changes that occur when you execute a Trace, Program Step, or Go command. The highlight marking the current location is moved to the new instruction in the display window. When appropriate, values are changed in the register and watch windows.

In sequential mode, the current source line or instruction is displayed after each Trace, Program Step, or Go command. The format of the display depends on the display mode. The three display modes in sequential mode (source, assembly, and mixed) are discussed in Chapter 9, "Examining Code."

If the display mode is source (**S+**) in sequential mode, the current source line is shown. If the display mode is assembly (**S–**), the status of the registers and the flags and the new instruction are shown in the format of the Register command (see Chapter 6, "Examining Data and Expressions"). If the display mode is mixed (**S&**), then the registers, the new source line, and the new instruction are all shown.

The commands that execute code are explained in Sections 5.2–5.6.

NOTE *If you are executing a section of code with the Go or Program Step command, you can usually interrupt program execution by pressing CTRL+BREAK or CTRL+C. This can terminate endless loops, or it can interrupt loops that are delayed by the Watchpoint or Tracepoint command (see Chapter 8, "Managing Watch Statements").*

CTRL+BREAK or CTRL+C may not work if your program has a special use for either of these key combinations. If you have an IBM Personal Computer AT (or a compatible computer), you can use the SYSTEM-REQUEST key to interrupt execution regardless of your program's use of CTRL+BREAK and CTRL+C.

5.2 Trace Command

The Trace command executes the current source line in source mode, or the current instruction in assembly mode. The current source line or instruction is the one pointed to by the CS and IP registers. In window mode, the current instruction is shown in reverse video or in a contrasting color.

In source mode, if the current source line contains a call, the CodeView debugger executes the first source line of the called routine. In this mode, the debugger will only trace into functions and routines that have source code. For example, if the current line contains a call to an intrinsic function or a standard C library function, the debugger will simply execute the function if you are in source mode, since the source code for Microsoft standard libraries is not available.

If you are in assembly or mixed mode, the debugger will trace into the function. In this mode, if the current instruction is **CALL**, **INT** or **REP**, the debugger executes the first instruction of the procedure, interrupt, or repeated string sequence.

NOTE *When you debug Microsoft Macro Assembler programs in source mode, the paragraph above still applies. The debugger will not trace into an **INT** or **REP** sequence when you are in source mode.*

Use the Trace command if you want to trace into calls. To execute calls without tracing into them, you should use the Program Step command (**P**) instead. Both commands execute DOS function calls without tracing into them. There is no direct way to trace into DOS function calls. However, you can trace through BIOS calls in assembly or mixed mode.

NOTE *The Trace command (**T**) uses the hardware trace mode of the 8086 family of processors. Consequently, you can also trace instructions stored in ROM (read-only memory). However, the Program Step command (**P**) will not work in ROM. Using the Program Step command has the same effect as using the Go command (**G**).*

Mouse

To execute the Trace command with the mouse, point to Trace on the menu bar and click Left.

Keyboard

To execute the Trace command with a keyboard command, press F8. This works in both window and sequential modes.

Dialog

To execute the Trace command using a dialog command, enter a command line with the following syntax:

T [[*count*]]

If the optional *count* is specified, the command executes *count* times before stopping.

Example

The following example shows the Trace command in sequential mode. (In window mode, there would be no output from the commands, but the display would be updated to show changes caused by the command.)

```
S+        ;* FORTRAN example
source
>.
9:                CALL INPUT (DATA,N,INPFMT)
>T 3
34:               OPEN (1,FILE='EXAMPLE.DAT',STATUS='OLD')
35:               DO 100 I=1,N
36:               READ (1,'(BN,I10)',END=999) DATA(I)

>
```

The FORTRAN example above sets the display mode to source, and then uses the Source Line command to display the current source line. (See Chapter 9, "Examining Code," for a further explanation of the Set Source and Source Line

commands.) Note that the current source line calls the subroutine `INPUT`. The Trace command is then used to execute the next three source lines. These lines will be the first three lines of the subroutine `INPUT`.

Debugging C and BASIC source code is very similar. If you execute the Trace command when the current source line contains a C function call or a BASIC subprogram call, the debugger will execute the first line of the called routine.

```
S-
assembly
>T
AX=0058  BX=3050  CX=000B  DX=3FB0  SP=304C  BP=3056
SI=00CC  DI=40E0
DS=49B7  ES=49B7  SS=49B7  CS=3FB0  IP=0013  NV UP EI PL NZ
AC PO NC
3FB0:0013 50             PUSH      AX
>
```

The example above sets the display mode to assembly and traces the current instruction. (This example and the next example are the same as the examples of the Program Step command in Section 5.3.) The Trace and Program Step commands behave differently only when the current instruction is a **CALL**, **INT**, or **REP** instruction.

```
S&
mixed
>T
AX=0000  BX=319C  CX=0028  DX=0000  SP=304C  BP=3056
SI=00CC  DI=40E0
DS=49B7  ES=49B7  SS=49B7  CS=3FB0  IP=003C  NV UP EI PL NZ
NA PO NC
8:              IF (N.LT.1 .OR. N.GT.1000) GO TO 100
3FB0:003C 833ECE2101   CMP      Word Ptr [21CE],+01
DS:21CE=0028
>
```

The example above sets the display mode to mixed and traces the current instruction.

5.3 Program Step Command

The Program Step command executes the current source line in source mode, or the current instruction in assembly mode. The current source line or instruction is the one pointed to by the CS and IP registers. In window mode, the current instruction is shown in reverse video or in a contrasting color.

In source mode, if the current source line contains a call, the CodeView debugger executes the entire routine and is ready to execute the line after the call. In assembly mode, if the current instruction is **CALL**, **INT**, or **REP**, the debugger executes the entire procedure, interrupt, or repeated string sequence. Use the Program Step command if you want to execute over routine, function, procedure, and interrupt calls. If you want to trace into calls, you should use the Trace command (**T**) instead. Both commands execute DOS function calls without tracing into them. There is no direct way to trace into DOS function calls.

Mouse

To execute the Program Step command with the mouse, point to Trace on the menu bar and click Right.

Keyboard

To execute the Program Step command with a keyboard command, press F10. This works in both window and sequential modes.

Dialog

To execute the Program Step command with a dialog command, enter a command line with the following syntax:

P [[*count*]]

If the optional *count* is specified, the command executes *count* times before stopping.

Example

This example shows the Program Step command in sequential mode. In window mode, there would be no output from the commands, but the display would be updated to show changes.

```
S+        ;* FORTRAN/BASIC example
source
>.
9:                CALL INPUT (DATA,N,INPFMT)
>P 3
10:               CALL BUBBLE (DATA,N)
11:               CALL STATS (DATA,N)
12:               END
>
```

The example above (in FORTRAN or BASIC) sets the display mode to source, and then uses the Source Line command to display the current source line. (See Chapter 9, "Examining Code," for an explanation of the Set Source and Source Line commands.) Notice that the current source line calls the subprogram `INPUT`. The Program Step command is then used to execute the next three

source lines. The first program step executes the entire subprogram `INPUT`. The next two steps execute the subprograms `BUBBLE` and `STATS` in their entirety.

The same program, written in C, would behave exactly the same way with the Program Step command. The Program Step command will not trace into a C function call.

```
S-
assembly
>P
AX=0058  BX=3050  CX=000B  DX=3FB0  SP=304C  BP=3056
SI=00CC  DI=40E0
DS=49B7  ES=49B7  SS=49B7  CS=3FB0  IP=0013  NV UP EI PL NZ
AC PO NC
3FB0:0013 50              PUSH      AX
>
```

The example above sets the display mode to assembly and steps through the current instruction. (This example and the next example are the same as the examples of the Trace command in Section 5.2.) The Trace and Program Step commands behave differently only when the current instruction is a **CALL**, **INT**, or **REP** instruction.

```
S&
mixed
>P
AX=0000  BX=319C  CX=0028  DX=0000  SP=304C  BP=3056
SI=00CC  DI=40E0
DS=49B7  ES=49B7  SS=49B7  CS=3FB0  IP=003C  NV UP EI PL NZ
NA PO NC
8:                IF (N.LT.1 .OR. N.GT.1000) GO TO 100
3FB0:003C 833ECE2101   CMP     Word Ptr [21CE],+01
DS:21CE=0028
>
```

The example above sets the display mode to mixed and steps through the current instruction.

5.4 Go Command

The Go command starts execution at the current address. There are two variations of the Go command—Go and Goto. The Go variation simply starts execution and continues to the end of the program or until a breakpoint set earlier with the Breakpoint Set (**BP**), Watchpoint (**WP**), or Tracepoint (**TP**) command is encountered. The other variation is a Goto command, in which a destination is given with the command.

If a destination address is given but never encountered (for example, if the destination is on a program branch that is never taken), the CodeView debugger executes to the end of the program.

If you enter the Go command and the debugger does not encounter a breakpoint, the entire program is executed and the following message is displayed:

```
Program terminated normally   (number)
```

The *number* in parentheses is the value returned by the program (sometimes called the exit or "errorlevel" code).

Mouse

To execute the Go command with no destination, point to Go on the menu bar and press either button.

To execute the Goto variation of the Go command, point to the source line or instruction you wish to go to, then press the right button. The highlight marking the current location will move to the source line or instruction you pointed to (unless a breakpoint is encountered first). The CodeView debugger will sound a warning and take no action if you try to go to a comment line or other source line that does not correspond to code.

If the line you wish to go to is in another module, you can use the Load command from the Files menu to load the source file for the other module. Then point to the destination line and press the right button.

Keyboard

To use a keyboard command to execute the Go command with no destination, press F5. This works in both window and sequential modes.

To execute the Goto variation of the Go command, point to the source line or instruction you wish to go to then press the right button. The highlight marking the current location will move to the source line or instruction to which you wish to go. If the cursor is in the dialog window, first press F6 to move the cursor to the display window. When the cursor is at the appropriate line in the display window, press F7. The highlight marking the current location will move to the source line or instruction you pointed to (unless a breakpoint is encountered first). If you try to go to a comment line or to another source line that does not correspond to code, the CodeView debugger will sound a warning and take no action.

If the line you wish to go to is in another module, you can use the Load command from the Files menu to load the source file for the other module. Then move the cursor to the destination line and press F7.

Dialog

To execute the Go command with a dialog command, enter a command line with the following syntax:

G [[*breakaddress*]]

If the command is given with no argument, execution continues until a breakpoint or the end of the program is encountered.

The Goto form of the command can be given by specifying *breakaddress*. The *breakaddress* can be given as a symbol, a line number, or an address in the *segment*:*offset* format. If the offset address is given without a segment, the address in the CS register is used as the default segment. If *breakaddress* is given as a line number, but the corresponding source line is a comment, declaration, or blank line, the following message appears:

```
No code at this line number
```

Examples

The following examples show the Go command in sequential mode. In window mode there would be no output from the commands, but the display would be updated to show changes caused by the command.

```
G

Program terminated normally (0)
>
```

The example above passes control to the instruction pointed to by the current values of the CS and IP registers. No breakpoint is encountered, so that the CodeView debugger executes to the end of the program, where it prints a termination message and the exit code returned by the program (`0` in the example).

```
S+      ;* FORTRAN/BASIC example (source mode)
source
>G BUBBLE
17:             A = B + C
>
```

In the example above, the display mode is first set to source (`S+`). (See Chapter 9, "Examining Code," for information on setting the display mode.) When the Go command is entered, the CodeView debugger starts program execution at the current address and continues until it reaches the start of the subprogram `BUBBLE`.

```
S&      ;* C example (mixed mode)
mixed
>G .22
AX=02F4  BX=0002  CX=00A8  DX=0000  SP=3036  BP=3042
SI=0070  DI=40E0
DS=49B7  ES=49B7  SS=49B7  CS=3FB0  IP=0141  NV UP EI PL NZ
NA PO NC
22:           x[i] = x[j];
3FB0:0141 8B7608       MOV     SI,Word Ptr [BP+08]
SS:304A=0070
>
```

The example above passes execution control to the program at the current address and executes to the address of source line 22. If the address with the breakpoint is never encountered (for example, if the program has less than 22 lines, or if the breakpoint is on a program branch that is never taken), the CodeView debugger executes to the end of the program.

```
S-
assembly
>G #2A8
AX=0049  BX=0049  CX=028F  DX=0000  SP=12F2  BP=12F6
SI=04BA  DI=1344
DS=5DAF  ES=5DAF  SS=5DAF  CS=58BB  IP=02A8  NV UP EI PL NZ
NA PE NC
58BB:02A8 98             CBW
>
```

The example above executes to the hexadecimal address CS:2A8. Since no segment address is given, the CS register is assumed.

NOTE *Mixed and source mode can be used equally well with all three languages. The examples alternate languages in this chapter simply to be accessible to more users.*

5.5 Execute Command

The Execute command is similar to the Go command with no arguments except that it executes in slow motion (several source lines per second). Execution starts at the current address and continues to the end of the program or until a breakpoint, tracepoint, or watchpoint is reached. You can also stop automatic program execution by pressing any key or a mouse button.

Mouse

To execute code in slow motion with the mouse, point to Run on the menu bar, press a mouse button and drag the highlight down to the Execute selection, and then release the button.

Keyboard

To execute code in slow motion with a keyboard command, press ALT+R to open the Run menu, and then press ALT+E to select Execute.

Dialog

To execute code in slow motion with a dialog command, enter a command line with the following syntax:

E

You cannot set a destination for the Execute command as you can for the Go command.

In sequential mode, the output from the Execute command depends on the display mode (source, assembly, or mixed). In assembly or mixed mode, the command executes one instruction at a time. The command displays the current status of the registers and the instruction. In mixed mode, it will also show a source line if there is one at the instruction. In source mode, the command executes one source line at a time, displaying the lines as it executes them.

NOTE *The Execute command has the same command letter (**E**) as the Enter command. If the command has at least one argument, it is interpreted as Enter; if not, it is interpreted as Execute.*

5.6 Restart Command

The Restart command restarts the current program. The program is ready to be executed just as if you had restarted the CodeView debugger. Program variables are reinitialized, but any existing breakpoints or watch statements are retained. The pass count for all breakpoints is reset to 1. Any program arguments are also retained, though they can be changed with the dialog version of the command.

The Restart command can only be used to restart the current program. If you wish to load a new program, you must exit and restart the CodeView debugger with the new program name.

Mouse

To restart the program with the mouse, point to Run on the menu bar, press a mouse button and drag the highlight down to the Restart or Start selection, and then release the button. The program will be restarted. If the Restart selection is chosen, the program will be ready to start executing from the beginning (but not actually running). If the Start selection is chosen, the program starts executing from the beginning and continues until a breakpoint or the end of the program is encountered.

Keyboard

To restart the program with a keyboard command, press ALT+R to open the Run menu, and then press either ALT+R to select Restart or ALT+S to select Start. The program will be restarted. If the Restart selection is chosen, the program will be ready to start executing from the beginning (but not actually running). If the Start selection is chosen, the program starts executing from the beginning and continues until a breakpoint or the end of the program is encountered.

Dialog

To restart the program with a dialog command, enter a command line with the following syntax:

L [[*arguments*]]

When you restart using the dialog version of the command, the program will be ready to start executing from the beginning. If you want to restart with new program arguments, you can give optional *arguments*. You cannot specify new arguments with the mouse or keyboard version of the command.

NOTE *The command letter* ***L*** *is a mnemonic for Load, but the command should not be confused with the Load selection from the File menu, since that selection only loads a source file without re-starting the program.*

Examples

```
L
>
```

The example above starts the current executable file, retaining any breakpoints, watchpoints, tracepoints, and arguments.

```
L 6
>
```

The example above restarts the current executable file, but with `6` as the new program argument.

CHAPTER 6

6

Examining Data and Expressions

The CodeView debugger provides several commands for examining different kinds of data such as expressions, symbols, memory, and registers. The data-evaluation commands discussed in this chapter are summarized below.

Command	Action
Display Expression (**?**)	Evaluates and displays locals, the value of symbols, or expressions
Graphic Display (**??**)	Displays local variables and complete data structures in a scrollable dialog box and traces pointer, structure, and array references
Examine Symbol (**X?**)	Displays the addresses of symbols
Dump (**D**)	Displays sections of memory containing data (with variations for examining different kinds of data)
Compare Memory (**C**)	Compares two blocks of memory, byte by byte
Search Memory (**S**)	Scans memory for specified byte values
Port Input (**I**)	Reads a byte from a hardware port
Register (**R**)	Shows the current value of each register and each flag (and optionally changes them)
8087 (**7**)	Shows the current value in the 8087 or 80287 register

6.1 Display Expression Command

The Display Expression command displays the value of a CodeView expression.

Each of the expression evaluators (C, FORTRAN, and BASIC) accepts a different set of symbols, operators, functions, and constants, as explained in Chapter 4, "CodeView Expressions." The resulting expressions can contain the intrinsic functions listed for the FORTRAN and BASIC expression evaluators. They may also contain functions that are part of the executable file. The simplest form of expression is a symbol representing a single variable or routine.

NOTE *FORTRAN subroutines and BASIC subprograms do not return values as functions do. They can be used in expressions, and may be useful for observing side effects. However, the value returned by the expression will be meaningless.*

In addition to displaying values, the Display Expression command can also set values as a side effect. For example, with the C expression evaluator you can increment the variable `n` by using the expression `++n` with the Display Expression command. With the FORTRAN expression evaluator you would use `N=N+1`, and with the BASIC expression evaluator you would use `LET N=N+1`. After being incremented, the new value will be displayed.

You can specify the format in which the values of expressions are displayed by the Display Expression command. Type a comma after the expression, followed by a CodeView format specifier. The format specifiers used in the CodeView debugger are a subset of those used by the C **printf** function. They are listed in Table 6.1.

Table 6.1 CodeView Format Specifiers

Character	Output Format	Sample Expression	Sample Output
d	Signed decimal integer	`?40000,d`	`40000`
i	Signed decimal integer	`?40000,i`	`40000`
u[1]	Unsigned decimal integer	`?40000,u`	`40000`
o	Unsigned octal integer	`?40000,o`	`116100D`
x or **X**	Hexadecimal integer	`?40000,x`	`9c40`
f	Signed value in floating-point decimal format with six decimal places	`?3./2.,f`	`1.500000`
e or **E**[2]	Signed value in scientific-notation format with up to six decimal places (trailing zeros and decimal point are truncated)	`?3./2.,e`	`1.500000e+00 0`

Table 6.1 *(continued)*

Character	Output Format	Sample Expression	Sample Output
g or **G**[3]	Signed value with floating-point decimal format (**f**) or scientific-notation format (**g** or **G**), whichever is more compact	`?3./2.,g`	`1.5`
c	Single character	`?65,c`	`A`
s[3]	Characters printed up to the first null character	`?"String",s`	`String`

[1] FORTRAN and BASIC have no unsigned data types. Using an unsigned format specifier has no effect on the output of positive numbers, but causes negative numbers to be output as positive values.

[2] The "E" is uppercse if the type is **E** or **G**; and lowercase if the type is **e** or **g**.

[3] The s string format is used only with the C expression evaluator; it prints characters up to the first null.

If no format specifier is given, single- and double-precision real numbers are displayed as if the format specifier had been given as **g**. (If you are familiar with the C language, you should note that the **n** and **p** format specifiers and the **F** and **H** prefixes are not supported by the CodeView debugger, even though they are supported by the C **printf** function.)

The prefix **h** can be used with the integer format specifiers (**d**, **o**, **u**, **x**, and **X**) to specify a two-byte integer. The prefix **l** can be used with the same types to specify a four-byte integer. For example, the command `?100000,ld` produces the output `100000`. However, the command `?100000,hd` evaluates only the low-order two bytes, producing the output `-31072`.

You can specify individual members of a C structure or BASIC user-defined type, or display the entire structure. Each member of a structure or BASIC user-defined type is displayed, within the limits of the dialog box. (See Section 6.2, "The Graphic Display Command," for information on how to see all the fields of a large structure.)

The Display Expression command does not work for programs assembled with Microsoft Macro Assembler Versions 4.0 and earlier because the assembler does not write information to the object file about the type size of each variable. Use the Dump command instead.

When calling a FORTRAN subroutine that uses alternate returns, the value of the return labels in the actual parameter list must be 0. For example, the subroutine call `CALL PROCESS (I,*10,J,*20,*30)` must be called from the debugger as `?PROCESS(IARG1,0,IARG2,0,0)`. Using other values as return labels will cause the error `Type clash in function argument` or `Unknown symbol`.

NOTE *Do not use a type specifier when evaluating strings in FORTRAN or BASIC. Simply leave off the type specifier, and the expression evaluator will display the string correctly. The* **s** *type specifier assumes the C language string format, with which other languages conflict; if you use* **s**, *then the debugger will simply display characters at the given address until a null is encountered.*

Mouse

The Display Expression command cannot be executed with the mouse.

Keyboard

The Display Expression command cannot be executed by using a keyboard command.

Dialog

To display the value of an expression using a dialog command, enter a command line with the following syntax:

? *expression*[[, *format*]]

The *expression* is any valid CodeView expression, and the optional *format* is a CodeView format specifier.

The remainder of this section first gives examples that are relevant to all languages and then gives examples specific to C, FORTRAN, and BASIC.

If you are debugging code written with the assembler, you will use the C expression evaluator by default. See Section 4.4 for guidelines on how to use the C expression evaluator with assembly code.

Examples

```
? amount
500
>? amount,x
1f4
>? amount,o
764
>
```

The example above displays the value stored in the variable `amount`, an integer. This value is first displayed in the system radix (in this case, decimal), then in hexadecimal, and then in octal.

```
? mystruct
{c1=123, c2={a=1, b=2}, c3=0x1000:2d34}
```

The example above shows how the CodeView debugger displays a C structure or BASIC user-defined type. Note that nested structures are displayed within an extra set of braces.

```
? 92,x
5c
>? 109*(35+2),o
7701
>? 118,c
v
>
```

The example above shows how the CodeView debugger can be used as a calculator. You can convert between radixes, calculate the value of constant expressions, or check ASCII equivalences.

```
? chance,f
0.083333
>? chance,e
8.333333e-002
>? chance,E
8.333333E-002
```

The example above shows a double-precision real number, `chance`, displayed in three formats. The **f** format always displays six digits of precision. The **e** format uses scientific notation. Note that the **E** format yields essentially the same display as **e** does.

The rest of the examples in this section are specific to particular languages.

C Examples

The following examples assume that a C source file is being debugged and it contains the following declarations:

```
char *text = "Here is a string."
int  amount;
struct {
        char    name[20];
        int     id;
        long    class;
} student, *pstudent;

int square(int);
```

Assume also that the program has been executed where the above variables have been assigned values, and that the C expression evaluator is in use.

```
? text, X
13f3
>DA 0x13F3
3D83:13F0  Here is a string.
>? text,s
Here is a string.
>
```

The example above shows how to examine strings. One method is to evaluate the variable that points to the string, and then dump the values at that address (the Dump commands are explained in Section 6.4). A more direct method is to use the **s** type specifier.

```
? student.id
19643
>? pstudent->id
19643
>
```

The example above illustrates how to display the values of members of a structure. The same syntax applies to unions.

```
? amount
500
>? ++amount
501
>? amount=600
600
>
```

The example above shows how the Display Expression command can be used with the C expression evaluator to change the values of variables.

```
? square(9)
81
>
```

The example above shows how functions can be evaluated in expressions. The CodeView debugger executes the function `square` with an argument of `9`, and displays the value returned by the function. Note that you can use symbols as well as constants as function arguments. However, you can only display function values after you have executed into the function `main`.

The C expression evaluator also supports type casts. The equivalent of a type cast in another language is a type-conversion function.

FORTRAN Examples

The examples below assume that the FORTRAN source file contains the following variable declarations, in which `SQUARE` is a function:

```
INTEGER*2 SQUARE
INTEGER*2 AMOUNT
CHARACTER*16 STR
STR = 'Here is a string'
```

Assume also that the program has executed to the point where these variables have been assigned values, and that the FORTRAN expression evaluator has been selected.

```
? STR
'Here is a string'
```

The example above shows how to examine strings with the FORTRAN expression evaluator. The s format specifier is not required.

```
? AMOUNT
500
>? AMOUNT=AMOUNT+1
501
>? AMOUNT=600
600
>? AMOUNT
600
>
```

The example above shows how the Display Expression command can be used to change values with the FORTRAN expression evaluator.

```
? SQUARE(9)
81
>
```

The example above shows how functions can be evaluated in expressions. The CodeView debugger executes the function `SQUARE` with an argument of `9`, and displays the value returned by the function. You can only display the values of functions after you have executed into the main program level.

BASIC Examples

These examples assume the BASIC source file contains the following statements:

```
amount% = 500
str$ = "Here is a string"
```

Assume also that the program has been executed up to these statements and that the BASIC expression evaluator is in use.

```
? str$
Here is a string
```

The first example above shows how to examine strings with the BASIC expression evaluator. The s format specifier should not be used.

```
? ASC(str$)
72
```

The second example demonstrates one of the BASIC intrinsic functions supported by the CodeView debugger, **ASC**, which returns the ASCII value of the first character in a string.

```
? amount%
500
>? LET amount%=amount%+1
501
>? LET amount%=600
600
>? amount%
600
>
```

The example above shows how the Display Expression command can be used to change values with the BASIC expression evaluator. With BASIC, the **LET** command can only be applied to numeric data, not strings.

NOTE *The BASIC expression evaluator cannot evaluate functions defined in the program, as the C and FORTRAN expression evaluators can.*

Assembly Examples

By default, the C expression evaluator is used for debugging assembly modules. However, some C expressions are particularly helpful for debugging assembly code. Some typical examples are presented below.

```
? BY bx
12
>
```

The example above displays the first byte at the location pointed to by BX, and is equivalent to the assembly expression `BYTE PTR [bx]`.

```
? WO bp+8
9359
>
```

The example above displays the first word at the location pointed to by `[bp+8]`.

```
? DW si+12
12555324
>
```

The example above displays the first double word at the location pointed to by `[si+12]`.

```
? (char) var
5
>? (int) var
1005
>
```

The last two examples use type casts, which are similar to the assembler **PTR** operator. The expression `(char) var` displays the byte at the address of `var`, in signed format. The expression `(int) var` displays the word at the same address, also in signed format. You can alter either of these commands to display results in unsigned format simply by using the **u** format specifier.

```
? (char) var,u

>? (int) var,u
```

6.2 The Graphic Display Command

The Graphic Display command (**??**) is similar to the Examine Symbols command. The Graphic Display command shows the value of any symbol you specify. However, the Graphic Display command is the more efficient means for viewing a multiple-field object such as a structure or a linked list of data.

The Graphic Display command lets you browse through related data. For example, both C and BASIC let you define structures inside of other structures. (In BASIC, structures are called "user-defined types.") The Graphic Display command lets you quickly move up and down through layers of structures. The command also works with C pointer variables; with a single mouse click or a few keystrokes, you see the entire structure that a pointer addresses. When you examine a list of structures linked through pointers, the Graphic Display command lets you quickly move back and forth through the list.

To resume debugging, you must remove the Graphic Display dialog box. Pressing ESC terminates the dialog box.

NOTE *Throughout the rest of this section, the term "structure" is used to refer to any of the following: a C structure, Pascal record, or BASIC user-defined type.*

This section discusses how to invoke the Graphic Display command and how to browse through data once the Graphic Display dialog box appears. Regardless of how you invoke the command, the same rules apply for browsing through the data.

6.2.1 Invoking the Graphic Display Command

The Graphic Display command is useful for evaluating a structure or pointer, although you can legally use the command with any variable. To use this command to display the contents of a variable, enter the following:

?? *symbol*, c

In the syntax display above, *symbol* is the name of any recognized variable, the second field is either blank or contains the character c.

The second field may contain the character `c`. This character is a C **printf** format specifier that causes CodeView to display each byte of a character array in its ASCII form, rather than display its numerical value.

As shown in the Figure 6.1 below, structures are represented as three dots enclosed in braces: {...}. Pointers are represented in the standard *segment:offset* format. The Graphic Display dialog box also displays a title; the title is the name of the variable or member currently displayed.

Example

```
?? graduate, c
```

The example above displays the members of a structure, as shown in Figure 6.1:

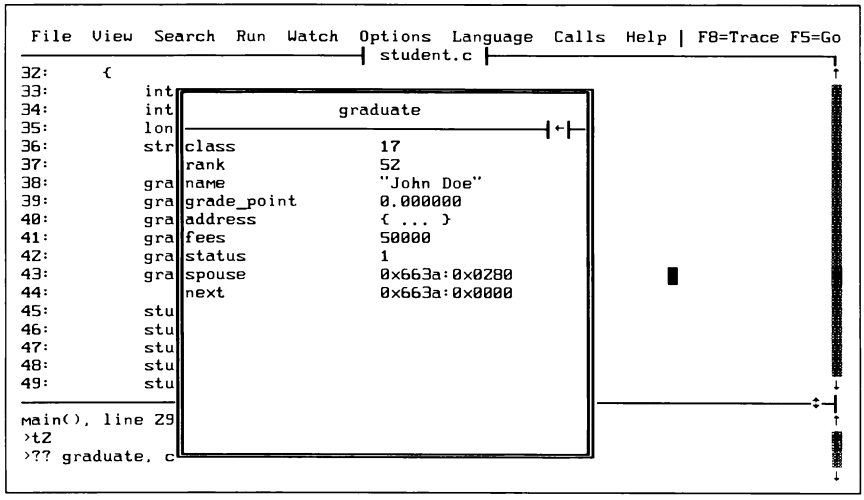


Figure 6.1 Viewing Structures with Graphic Display

As is the case with the display of local variables, nested structures are represented as three dots enclosed in braces, and pointers are represented in the

standard *segment:offset* format. Section 6.2.2, "Changing the Display," explains how to select nested structures and pointers for display.

```
?? Kount
```

Since `Kount` is neither a structure nor an array, CodeView responds by displaying a single field as shown in Figure 6.2:

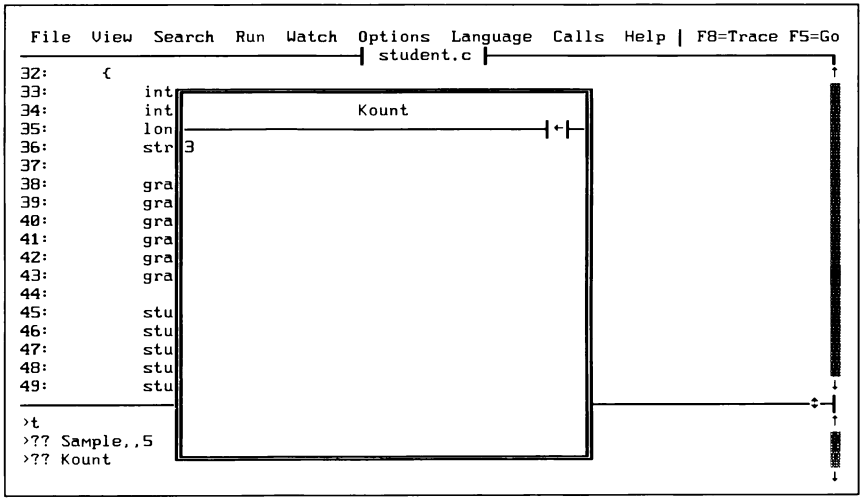


Figure 6.2 Viewing a Simple Variable with Graphic Display

To close the Graphic display dialog box and continue debugging, click left outside the dialog box or press ESC.

6.2.2 Changing the Display

Once the Graphic Display dialog box appears, you change what information is displayed by selecting an individual variable, member, or array element. (However, the command displays array elements only when the current module is a C module.) Making such a selection changes the subject matter of the dialog box; for example, selecting a nested structure moves you one level deeper within the structure. You can use either the mouse or the keyboard to select an item.

Changing the Display with the Mouse

To select an item with the mouse, simply click the left mouse button on the line where the item appears.

CodeView allows you to move backward through displays as well as forward. After you select an item and move to a new display, CodeView remembers the previous state of the dialog box. To move back to the previous display, click the backward arrow just below the dialog box title, or click the right mouse button.

To close the dialog box and continue debugging, click the left mouse button while outside the dialog box.

Changing the Display with the Keyboard

To select an item with the keyboard, move the cursor to the desired item and press ENTER.

CodeView allows you to move backward through displays as well as forward. After you select an item and move to a new display, CodeView remembers the previous state of the dialog box. To move back to the previous display, press BACKSPACE.

To close the dialog box and continue debugging, press ESC.

Effect of Selecting an Item

Depending on the item you select, CodeView executes a specific action:

Item	Action
Nested structure	The structure is "expanded"; the nested structure becomes the new subject of the dialog box. The dialog box displays each member of the nested structure.
Pointer	The pointer is "dereferenced"; in other words, CodeView locates the data that the pointer addresses. This data becomes the new subject of the dialog box. The pointer's type determines how the debugger displays the dereferenced data. The debugger uses this type information even if the pointer does not currently address any meaningful data. If the pointer addresses a structure, CodeView displays each element.
Other items	CodeView takes no action.

No matter how many times you change the display, and no matter what the previous display looked like, all the rules above apply. You can repeat these operations any number of times. For example, given a sufficiently complex structure, you can move down several levels of nested structures, then follow a pointer reference to another variable.

6.3 Examine Symbols Command

The Examine Symbols command displays the names and addresses and the names of modules defined within a program. You can specify the group you want to examine by module, procedure, or name.

Mouse

The Examine Symbols command cannot be executed with the mouse.

Keyboard

The Examine Symbols command cannot be executed with a keyboard command.

Dialog

To view the addresses of symbols with a dialog command, enter a command line in one of the following formats:

XL
X*
X
X? [[*module*!]] [[*routine*.]] [[*symbol*]] [[*]]

in which *routine* is in a program unit, such as a C function or a BASIC subprogram, capable of having its own local variables.

The syntax combinations are listed in more detail below.

Syntax	Display
X?*module*!*routine*.*symbol*	The specified *symbol* in the specified *routine* in the specified *module*.
X?*module*!*routine*.*	All symbols in the specified *routine* in the specified *module*.
X?*module*!*symbol*	The specified *symbol* in the specified *module* (symbols within routines are not found).
X?*module*!*	All symbols in the specified *module*.
X?*routine*.*symbol*	The specified *symbol* in the specified *routine* (looks for *routine* first in the current module, and then in other modules from first to last).

X?*routine*.*	All symbols in the specified *routine* (looks for *routine* first in the current module, and then in other modules from first to last).
X?*symbol*	Looks for the specified *symbol* in this order: 1. In the current routine. 2. In the current module. 3. In other modules, from first to last.
X?*	All symbols in the current routine.
XL	All local variables of the currently executing routine. This variation of the command uses a special format as explained below.
X*	All module names (file extensions are added to these names).
X	All symbolic names in the program, including all modules and symbols.

When you debug an assembly module, you cannot use the *routine* field; you must use the *module* field. Therefore, the only versions of this command that work with assembly modules are the following:

X?*module***!***
X?*module***!***symbol*

XL is a special variation of the Examine Symbol command. It lists local variables for the currently executing routine and provides more information than other variations of the command.

Whereas most forms of the command display the address, type, and name of each symbol, the **XL** variation displays the value of each local variable as well. The value of a local variable is displayed in the same format that the Display Expression command would use, assuming no type specifier.

The following example shows the use of the **XL** command when the currently executing routine has many local variables.

```
XL
[BP+0004]    int          argc = 1
[BP+0006]    char * *     argv = 0x2f:0x1510
[BP-0002]    int          i = 20
SI register  int          k = 7
[BP-0078]    struct cat   item0 = {item1=0, item2=0,
dog=0x2f:0x1476}
[BP-0070]    struct cow   moo = {c1=11, c2=22, c3=36, c4=16}
[BP-0008]    char *       wiz = 0x2f:0x1514
[BP-0080]    int          duck = 0
DI register  int          j = 83
```

In the example above, variables `i` and `j` are register variables assigned to the registers `SI` and `DI`, respectively. The other variables are located on the stack; `XL` shows the displacement of each variable from the `BP` register, which holds the value of the stack pointer (SP) at the time of entry into the procedure.

If you have a parameter that is declared as a register in a C program, it will appear twice: on the stack (as an offset from BP) and in the SI or DI register.

Note that if you program in assembly, local variables are not recognized by CodeView unless you use the **PROC** and **LOCALS** directives provided with MASM Version 5.1 and later.

The rest of this section shows examples using the other variations of the Examine Symbols command.

C Examples

For the following examples, assume that the program being examined is called `pi.exe`, and that it consists of two modules: `pi.c` and `math.c`. The `pi.c` module is a skeleton consisting only of the `main` function, whereas the `math.c` module has several functions. Assume that the current function is **div** within the `math` module.

```
X*                ;*Example 1
PI.OBJ
MATH.OBJ
chkstk.asm
crt0.asm
.
.
.
C:\LIB\SLIBC.LIB(xtoa.asm)
>
```

Example 1 lists the two user-created modules of the program, as well as the library modules used in the program.

```
X?*             ;*Example 2
            DI          int             b
            [BP-0006]   int             quotient
            SI          int             i
            [BP-0002]   int             remainder
            [BP+0004]   int             divisor
>
```

Example 2 lists the symbols in the current function (**div**). Local variables are shown as being stored either in a register (`b` in register `DI`) or at a memory location specified as an offset from a register (`divisor` at location `[BP+0004]`).

```
X?pi!*          ;* Example 3
3D37:19B2 int               _scratch0   3D37:0A10 char
_p[]
3D37:2954 int               _scratch1   3D37:19B4 char
_t[]
3D37:2956 int               _scratch2   3D37:19B0 int
_q
3A79:0010 int               _main()     3A79:0010 int
main()
3D37:19B2 int                scratch0
3D37:0A10 char               p[]
3D37:2954 int                scratch1
3D37:19B4 char               t[]
3D37:2956 int                scratch2
3D37:19B0 int                q
>
```

Example 3 shows all the symbols in the `pi.c` module.

```
X?math!div.*    ;*Example 4
3A79:0264 int               div()
            DI          int             b
            [BP-0006]   int             quotient
            SI          int             i
            [BP-0002]   int             remainder
            [BP+0004]   int             divisor
>
```

Example 4 shows the symbols in the `div` function in module `math.c`. You wouldn't need to specify the module if `math.c` were the current module, but you would if the current module were `pi.c`.

Variables local to a function are indented under that function.

```
X?math!arctan.s ;* Example 5
3A79:00FA int               arctan()
            [BP+0004]   int             s
>
```

Example 5 shows one specific variable (`s`) within the `arctan` function.

FORTRAN Examples

For the following examples, assume that the program being examined is called FRUST.EXE, and it consists of four modules: FRUST.FOR, FRUST1.FOR, FRUST2.FOR, and FRUST3.FOR. Assume that the current routine is main within the FRUST.FOR module.

```
X*
FRUST.OBJ
FRUST1.OBJ
FRUST2.OBJ
FRUST3.OBJ
fixups.asm
crt0.asm
.
.
.
txtmode.asm
_creat.asm
```

The example above lists the four modules called by the program. The library files called by the program are also listed.

```
X?T
        520D:0DE4 REAL*4         T
```

The example above shows the address of the variable T in the current module.

```
X?FRUST3!MULTPI.*
4B28:0005 INTEGER*4         MULTPI()
                [BP+000A]                   V
                [BP+0006]                   X
                [BP-0004] INTEGER*4         MULTPI
```

The example above lists the symbols in the function MULTPI, located in module FRUST3. Variables local to the function are indented under the function. You wouldn't need to specify the module if FRUST3 were the current module.

```
X?FRUST2!SAREA.*
4B15:000E void              SAREA()
                [BP+0012]                   R1
                [BP+000E]                   R2
                [BP+000A]                   H
                [BP+0006]                   T
                520D:0DEC REAL*4            S12
                520D:0DE8 REAL*4            U
```

The example above shows all the symbols in the routine SAREA in the module FRUST2. Because SAREA is a subroutine instead of a function, the word void appears where function return-value types are shown.

BASIC Examples

For the following examples, assume that the program being examined is called PROG.EXE, and it consists of the following modules: PROG.BAS and SORT.BAS. Assume that the current routine is the main program (which, unlike subprograms, has no name in a BASIC program) and the module SORT.BAS contains two subprograms, SORT and SWITCH.

```
X*
PROG.OBJ
SORT.OBJ
ftmdata.asm
crt0.asm
crt0dat.asm
.
.
.
xtoa.asm
```

The example above lists the two modules of the program, including PROG.OBJ, which is the main module. The BASIC library files called by the program are also listed.

```
X?*
        5825:17BE integer          A%[array]
        5825:1780 single           HOURS!
        5825:1784 integer          I%
```

The example above lists the symbols in the current routine, which happens to be the main program. Although the main program has no label and therefore will not show up in a stack trace, it is still an independent routine and has its own local variables. In BASIC, local variables are not put on the stack unless they are subprogram parameters.

```
X?*SORT!*
        572F:0033 integer          SORT()
        572F:00E1 integer          SWITCH()
```

The example above lists the routines in the module SORT.OBJ. This form of the Display Symbols command lists routines only, not variables. Note that SORT() and SWITCH() are given with the addresses of the two subprograms by that name.

```
X?SORT!SWITCH.*
        [BP+0008] integer          B%
        [BP+0006] integer          C%
        5824:1798 integer          TEMP%
```

The example above shows all the symbols in the routine SWITCH, which is in the SORT.OBJ module. Each represents an integer. However, B% and C% represent subprogram parameters that were passed on the stack, whereas

TEMP% is a true subprogram variable. Therefore, TEMP% has an absolute address in memory, whereas B% and C% are addressed relative to the stack. (BP points to the value of the stack at the time the routine SWITCH was called.)

6.4 Dump Commands

The CodeView debugger has several commands for dumping data from memory to the screen (or other output device). The Dump commands are listed below.

Command	Command Name
D	Dump (size is the default type)
DB	Dump Bytes
DA	Dump ASCII
DI	Dump Integers
DU	Dump Unsigned Integers
DW	Dump Words
DD	Dump Double Words
DS	Dump Short Reals
DL	Dump Long Reals
DT	Dump 10-Byte Reals

Mouse

The Dump commands cannot be executed with the mouse.

Keyboard

The Dump commands cannot be executed with keyboard commands.

Dialog

To execute a Dump command with a dialog command, enter a command line with the following syntax:

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

The *type* is a one-letter specifier that indicates the type of the data to be dumped. The Dump commands expect either a starting *address* or a *range* of memory. If the starting *address* is given, the commands assume a default range (usually

determined by the size of the dialog window) starting at *address*. If *range* is given, the commands dump from the start to the end of *range*. The maximum size of *range* is 32K.

If neither *address* nor *range* is given, the commands assume the current dump address as the start of the range and the default size associated with the size of the object as the length of the range. The Dump Real commands have a default range size of one real number. The other Dump commands have a default size determined by the size of the dialog window (if you are in window mode), or a default size of 128 bytes otherwise.

The current dump address is the byte following the last byte specified in the previous Dump command. If no Dump command has been used during the session, the dump address is the start of the data segment (DS). For example, if you enter the Dump Words command with no argument as the first command of a session, the CodeView debugger displays the first 64 words (128 bytes) of data declared in the data segment. If you repeat the same command, the debugger displays the next 64 words following the ones dumped by the first command.

NOTE *If the value in memory cannot be evaluated as a real number, the Dump commands that display real numbers (Dump Short Reals, Dump Long Reals, or Dump 10-Byte Reals) will display a number containing one of the following character sequences:* `#NAN`, `#INF`, *or* `#IND`. *NAN (not a number) indicates that the data cannot be evaluated as a real number. INF (infinity) indicates that the data evaluates to infinity. IND (indefinite) indicates that the data evaluates to an indefinite number.*

Sections 6.4.1–6.4.10 discuss the variations of the Dump commands in order of the size of data they display.

6.4.1 Dump

Syntax

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

The Dump command displays the contents of memory at the specified *address* or in the specified *range* of addresses. The command dumps data in the format of the default type. The default type is the last type specified with a Dump, Enter, Watch Memory, or Tracepoint Memory command. If none of these commands has been entered during the session, the default type is bytes.

The Dump command displays one or more lines, depending on the address or range specified. Each line displays the address of the first item displayed. The Dump command must be separated by at least one space from any *address* or *range* value. For example, to dump memory starting at symbol `a`, use the command `D a`, not `Da`. The second syntax would be interpreted as the Dump ASCII command.

6.4.2 Dump Bytes

Syntax

DB [[*address* | *range*]]

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

Each line displays the address of the first byte in the line, followed by up to 16 hexadecimal byte values. The byte values are immediately followed by the corresponding ASCII values. The hexadecimal values are separated by spaces, except the eighth and ninth values, which are separated by a dash (–). ASCII values are printed without separation. Unprintable ASCII values (less than 32 or greater than 126) are displayed as dots. No more than 16 hexadecimal values are displayed in a line. The command displays values and characters until the end of the *range* or, if no *range* is given, until the first 128 bytes have been displayed.

Example

```
DB 0 36
3D5E:0000 53 6F 6D 65 20 6C 65 74-74 65 72 73 20 61 6E 64
Some letters and
3D5E:0010 20 6E 75 6D 62 65 72 73-3A 00 10 EA 89 FC FF EF
numbers:.......
3D5E:0020 00 F0 00 CA E4          -
.....
>
```

The example above displays the byte values from `DS:0` to `DS:36` (36 decimal is equivalent to 24 hexadecimal). The data segment is assumed if no segment is given. ASCII characters are shown on the right.

6.4.3 Dump ASCII

Syntax

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

The Dump ASCII command displays the ASCII characters at a specified *address* or in a specified *range* of addresses. The command displays one or more lines of characters, depending on the *address* or *range* specified.

If no ending address is specified, the command dumps either 128 bytes or all bytes preceding the first null byte, whichever comes first. Up to 64 characters per line are displayed. Unprintable characters, such as carriage returns and line feeds, are displayed as dots. ASCII characters less than 32 and greater than 126 in number are unprintable.

Examples

```
DA 0
3D7C:0000  Some letters and numbers:
>
```

The example above displays the ASCII values of the bytes starting at `DS:0`. Since no ending address is given, values are displayed up to the first null byte.

```
DA 0 36
3D7C:0000  Some letters and numbers:.....
>
```

In the example above, an ending address is given, so that the characters from `DS:0` to `DS:36` (24 hexadecimal) are shown. Unprintable characters are shown as dots.

6.4.4 Dump Integers

Syntax

DI [[*address* | *range*]]

The Dump Integers command displays the signed decimal values of the words (two-byte values) starting at *address* or in the specified *range* of addresses. The command displays one or more lines, depending on the address or range specified. Each line displays the address of the first integer in the line, followed by up to eight signed decimal words. The values are separated by spaces. The command displays values until the end of the *range* or until the first 64 two-byte integers have been displayed, whichever comes first.

NOTE *In this manual an integer is considered a two-byte value since the CodeView debugger assumes that integer size. Note that a default FORTRAN integer is a four-byte value.*

Example

```
DI 0 36
3D5E:0000   28499  25965  27680  29797  25972  29554  24864
25710
3D5E:0010   28192  28021  25954  29554     58  -5616   -887
-4097
3D5E:0020   -4096 -13824   2532
>
```

The example above displays the byte values from `DS:0` to `DS:36` (24 hexadecimal). Compare the signed decimal numbers at the end of this dump with the same values shown as unsigned integers in Section 6.4.5 below.

6.4.5 Dump Unsigned Integers

Syntax

DU [[*address* | *range*]]

The Dump Unsigned Integers command displays the unsigned decimal values of the words (two-byte values) starting at *address* or in the specified *range* of addresses. The command displays one or more lines, depending on the address or range specified. Each line displays the address of the first unsigned integer in the line, followed by up to eight decimal words. The values are separated by spaces. The command displays values until the end of the *range* or until the first 64 unsigned integers have been displayed, whichever comes first.

Example

```
DU 0 36
3D5E:0000   28499  25965  27680  29797  25972  29554  24864
25710
3D5E:0010   28192  28021  25954  29554     58  59920  64649
61439
3D5E:0020   61440  51712   2532
>
```

The example above displays the byte values from `DS:0` to `DS:36` (24 hexadecimal). Compare the unsigned decimal numbers at the end of this dump with the same values shown as signed integers in Section 6.4.4 above.

6.4.6 Dump Words

Syntax

DW [[*address* | *range*]]

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

Example

```
DW 0 36
3D5E:0000  6F53 656D 6C20 7465 6574 7372 6120 646E
3D5E:0010  6E20 6D75 6562 7372 003A EA10 FC89 EFFF
3D5E:0020  F000 CA00 09E4
>
```

The example above displays the word values from `DS:0` to `DS:36` (24 hexadecimal). No more than eight values per line are displayed.

6.4.7 Dump Double Words

Syntax

DD [[*address* | *range*]]

The Dump Double Words command displays the hexadecimal values of the double words (four-byte values) starting at *address* or in the specified *range* of addresses.

The command displays one or more lines, depending on the address or range specified. Each line displays the address of the first double word in the line, followed by up to four hexadecimal double-word values. The words of each double word are separated by a colon. The values are separated by spaces. The command displays values until the end of the *range* or until the first 32 double words have been displayed, whichever comes first.

Example

```
DD 0 36
3D5E:0000  656D:6F53 7465:6C20 7372:6574 646E:6120
3D5E:0010  6D75:6E20 7372:6562 EA10:003A EFFF:FC89
3D5E:0020  CA00:F000 6F73:09E4
>
```

The example above displays the double-word values from `DS:0` to `DS:36` (24 hexadecimal). No more than four double-word values per line are displayed.

6.4.8 Dump Short Reals

Syntax

DS [[*address* | *range*]]

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

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

The decimal value has the following form:

[[–]]*digit.digits*E{+ | –}*exponent*

If the number is negative, it will have a minus sign; positive numbers have no sign. The first digit of the number is followed by a decimal point. Six decimal places are shown following the decimal point. The letter E follows the decimal digits and marks the start of a three-digit signed *exponent.*

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

Example

```
DS SPI
5E68:0100  DB 0F 49 40  3.141593E+000
>
```

The example above displays the short-real floating-point number at the address of the variable `SPI`. Only one value is displayed per line.

6.4.9 Dump Long Reals

Syntax

DL [[*address* | *range*]]

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

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

The decimal value has the following form:

[[–]]*digit.digits*E{+ | –}*exponent*

If the number is negative, it will have a minus sign; positive numbers have no sign. The first digit of the number is followed by a decimal point. Six decimal places are shown following the decimal point. The letter E follows the decimal digits and marks the start of a three-digit signed *exponent.*

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

Example

```
DL LPI
5E68:0200  11 2D 44 54 FB 21 09 40  3.141593E+000
>
```

The example above displays the long-real floating-point number at the address of the variable `LPI`. Only one value per line is displayed.

6.4.10 Dump 10-Byte Reals

Syntax

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

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

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

The decimal value has the following form:

[[−]]*digit.digits***E**{+ | −}*exponent*

If the number is negative, it will have a minus sign; positive numbers have no sign. The first digit of the number is followed by a decimal point. Six decimal places are shown following the decimal point. The letter E follows the decimal digits and marks the start of a three-digit signed *exponent.*

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

Example

```
DT TPI
5E68:0300  DE 87 68 21 A2 DA 0F C9 00 40  3.141593E+000
>
```

The example above displays the 10-byte floating-point number at the address of the variable `TPI`. Only one number per line is displayed.

6.5 Compare Memory Command

The Compare Memory command provides a convenient way for comparing two blocks of memory, specified by absolute addresses. This command is primarily of interest to programmers using assembly mode; however, it can be useful to anyone who wants to compare two large areas of data, such as arrays.

Mouse

The Compare Memory command cannot be executed with the mouse.

Keyboard

The Compare Memory command cannot be executed with a keyboard command.

Dialog

To compare two blocks of memory, enter a command line with the following syntax:

C *range address*

The bytes in the memory locations specified by *range* are compared with the corresponding bytes in the memory locations beginning at *address*. If one or more pairs of corresponding bytes do not match, each pair of mismatched bytes is displayed.

Examples

```
C 100 01FF 300      ;* hexadecimal radix assumed
39BB:0102 0A 00 39BB:0302
39BB:0108 0A 01 39BB:0308
>
```

The first example (in which hexadecimal is assumed to be the default radix) compares the block of memory from 100 to 1FF with the block of memory from 300 to 3FF. It indicates that the third and ninth bytes differ in the two areas of memory.

```
C arr1(1) L 100  arr2(1)  ;* BASIC/FORTRAN notation used
>
```

The second example compares the 100 bytes starting at the address of `arr1(1)`, with the 100 bytes starting at address of `arr2(1)`. The CodeView debugger produces no output in response, so this indicates that the first 100 bytes of each array are identical. (In C language, this example would be entered as `Carr1[0] L 100 arr2[0]`.)

NOTE *You can enter the Compare Memory command using any radix you like; however, any output will still be in hexadecimal format.*

6.6 Search Memory Command

The Search Memory command (not to be confused with the Search command discussed in Section 11.5) scans a specified area of memory, looking for specific byte values. It is primarily of interest to programmers using assembly mode and to users testing for the presence of specific values within a range of data.

Mouse

The Search Memory command cannot be executed with the mouse.

Keyboard

The Search Memory command cannot be executed with a keyboard command.

Dialog

To search a block of memory, enter the Search Memory command with the following syntax:

S *range list*

The debugger will search the specified *range* of memory locations for the byte values specified in the *list*. If bytes with the specified values are found, the debugger displays the addresses of each occurrence of bytes in the list.

The *list* can have any number of bytes. Each byte value must be separated by a space or comma, unless the list is an ASCII string. If the list contains more than one byte, the Search Memory command looks for a series of bytes that precisely match the order and value of bytes in *list*. If found, the beginning address of each such series is displayed.

Examples

```
S buffer L 1500 "error"
2BBA:0404
2BBA:05E3
2BBA:0604
>
```

The first example displays the address of each memory location containing the string `error`. The command searches the first 1500 bytes at the address specified by `buffer`. The string was found at the three addresses displayed by the CodeView debugger.

```
S DS:100 200 0A   ;* hexadecimal radix assumed
3CBA:0132
3CBA:01C2
>
```

The second example displays the address of each memory location that contains the byte value 0A in the range DS:0100 to DS:0200 (hexadecimal). The value was found at two addresses.

6.7 Port Input Command

The Port Input command reads and displays a byte from a specified hardware port. It is primarily of interest to assembly-language programmers writing hardware-specific programs.

Mouse

The Port Input command cannot be executed with the mouse.

Keyboard

The Port Input command cannot be executed with a keyboard command.

Dialog

The Port Input command is executed with the following syntax:

I *port*

The byte is read and displayed from the specified *port*, which can be any 16-bit address.

Examples

```
I 2f8   ;* hexadecimal radix assumed
E8
>
```

The preceding example reads input port, number `2F8`, and displays the result, `E8`. You may enter the port address using any radix you want, but the result will always be displayed in current radix.

The Port Input command is often used in conjunction with the Port Output command, which is described in Section 10.5.

6.8 Register Command

The Register command has two functions. It displays the contents of the central processing unit (CPU) registers. It can also change the values of the registers. The display features of the Register command are explained here. The modification features of the command are explained in Chapter 10, "Modifying Code or Data."

Mouse

To display the registers with the mouse, point to View on the menu bar, press a mouse button and drag the highlight down to the Registers selection, then release the button. The register window will appear on the right side of the screen. If the register window is already on the screen, the same command removes it.

Keyboard

To display the registers using a keyboard command in window mode, press F2. The register window will appear on the right side of the screen. If the register window is already on the screen, the same command will remove it.

In sequential mode, the F2 key will display the current status of the registers. (This produces the same effect as entering the Register dialog command with no argument.)

Dialog

To display the registers in the dialog window (or sequentially in sequential mode), enter a command line with the following syntax:

R

The current values of all registers and flags are displayed. The instruction at the address pointed to by the current CS and IP register values is also shown. (The Register command can also be given with arguments, but only when used to modify registers, as explained in Chapter 10, "Modifying Code or Data.")

If the display mode is source (**S+**) or mixed (**S&**) (see Section 9.1, "Set Mode Command," for more information), the current source line is also displayed by the Register command. If an operand of the instruction contains memory expressions or immediate data, the CodeView debugger will evaluate operands and show the value to the right of the instruction. This value is referred to as the "effective address," and is also displayed at the bottom of the register window. If the CS and IP registers are currently at a breakpoint location, the register display will indicate the breakpoint number.

In sequential mode, the Trace (**T**), Program Step (**P**), and Go (**G**) commands show registers in the same format as the Register command.

Examples

```
S&
mixed
>R
AX=0005  BX=299E  CX=0000  DX=0000  SP=3800  BP=380E
SI=0070  DI=40D1
DS=5067  ES=5067  SS=5067  CS=4684  IP=014F  NV UP EI PL NZ
NA PO NC
35:                    VARIAN = (N*SUMXSQ-SUMX**2)/(N-1)
4684:014F 8B5E06       MOV     BX,Word Ptr [BP+06]     ;BR1
SS:3814=299E
>
```

The example above displays all register and flag values, as well as the instruction at the address pointed to by the CS and IP registers. Because the mode has been set to mixed (**S&**), the current source line is also shown. The example is from a FORTRAN program, but applies equally well to BASIC and C programs.

```
S-
assembly
>R
AX=0005  BX=299E  CX=0000  DX=0000  SP=3800  BP=380E
SI=0070  DI=40D1
DS=5067  ES=5067  SS=5067  CS=4684  IP=014F  NV UP EI PL NZ
NA PO NC
4684:014F 8B5E06       MOV     BX,Word Ptr [BP+06]     ;BR1
SS:3814=299E
>
```

In the example above, the display mode is set to assembly (**S–**), so no source line is shown. Note the breakpoint number at the right of the last line indicating that the current address is at Breakpoint 1.

6.9 8087 Command

The 8087 command dumps the contents of the 8087 registers. If you do not have an 8087, 80287, or 80387 coprocessor chip on your system, this command will dump the contents of the pseudoregisters created by the compiler's emulator routines. This command is useful only if you have an 8087, 80287, or 80387 chip installed or if your executable file includes math routines from a Microsoft 8087-emulator library.

NOTE *This section does not attempt to explain how the registers of the Intel 8087 and 80287 processors are organized or how they work. In order to interpret the command output, you must learn about the chip from an Intel reference manual or other book on the subject. Since the Microsoft emulator routines mimic the behavior of the 8087 coprocessor, these references will apply to emulator routines as well as to the chips themselves.*

Mouse

The 8087 command cannot be executed with the mouse.

Keyboard

The 8087 command cannot be executed with a keyboard command.

Dialog

To display the status of the 8087 or 80287 chip (or floating-point emulator routines) with a dialog command, enter a command line with the following syntax:

```
7
```

The current status of the chip is displayed when you enter the command. In window mode, the output is to the dialog window. If you do not have an 8087 or 80287 chip and are not linking to an emulator library, the debugger will report the error message `Floating point not loaded`. CodeView reports this message each time you give the **7** command, unless a floating-point instruction has been executed.

The following example shows a display for a machine that actually has an 8087 or 80287 chip. The example at the end of this section shows the same display for a machine using an emulator library instead of an actual math coprocessor.

8087 Example

```
7
cControl 037F  (Projective closure, Round nearest, 64-bit
precision)
                          iem=0 pm=1 um=1 om=1 zm=1 dm=1 im=1
cStatus  6004  cond=1000 top=4 pe=0 ue=0 oe=0 ze=1 de=0 ie=0
Tag     A1FF  instruction=59380  operand=59360  opcode=D9EE
Stack         Exp  Mantissa           Value
cST(3) special 7FFF 8000000000000000 = + Infinity
cST(2) special 7FFF 0101010101010101 = + Not a Number
cST(1) valid   4000 C90FDAA22168C235 =
+3.141592265110390E+000
cST(0) zero    0000 0000000000000000 =
+0.000000000000000E+000
>
```

In the example above, the first line of the dump shows the current closure method, rounding method, and the precision. The number `037F` is the hexadecimal value in the control register. The rest of the line interprets the bits of the number. The closure method can be either projective (as in the example) or affine. The rounding method can be either rounding to the nearest even number (as in the example), rounding down, rounding up, or using the chop method of rounding (truncating toward zero). The precision may be 64 bits (as in the example), 53 bits, or 24 bits.

The second line of the display indicates whether each exception mask bit is set or cleared. The masks are interrupt-enable mask (`iem`), precision mask (`pm`), underflow mask (`um`), overflow mask (`om`), zero-divide mask (`zm`), denormalized-operand mask (`dm`), and invalid-operation mask (`im`).

The third line of the display shows the hexadecimal value of the status register (`6004` in the example), and then interprets the bits of the register. The condition code (`cond`) in the example is the binary number `1000`. The top of the stack (`top`) is register 4 (shown in decimal). The other bits shown are precision exception (`pe`), underflow exception (`ue`), overflow exception (`oe`), zero-divide exception (`ze`), denormalized-operand exception (`de`), and invalid-operation exception (`ie`).

The fourth line of the display first shows the 20-bit hexadecimal address value of the tag register (`A1FF` in the example). It then gives the hexadecimal offsets of the instruction (`59380`), the operand (`59360`), and the operation code, or opcode, (`D9EE`).

The fifth line is a heading for the subsequent lines that contain the contents of each 8087 or 80287 stack register. The registers in the example contain four types of numbers that may be held in these registers. Starting from the bottom, register 0 contains zero. Register 1 contains a valid real number. Its exponent (in hexadecimal) is `4000` and its mantissa is `C90FDAA22168C235`. The number is shown in scientific notation in the rightmost column. Register 2 contains a value that cannot be interpreted as a number, and register 3 contains infinity.

The `c` that precedes `Control`, `Status`, and each of the `ST` listings indicates that an actual math-coprocessor chip is in use. If emulator routines were in use instead of a chip, each `c` prefix would be replaced by `e`, as in the next example.

Floating-Point Emulator Example

```
7
eControl 037F  (Projective closure, Round nearest, 64-bit
precision)
                          iem=0 pm=1 um=1 om=1 zm=1 dm=1 im=1
eStatus  6004  cond=1000 top=4 pe=0 ue=0 oe=0 ze=1 de=0 ie=0
Tag      A1FF  instruction=59380  operand=59360  opcode=D9EE
Stack          Exp  Mantissa           Value
eST(3) special 7FFF 8000000000000000 = + Infinity
eST(2) special 7FFF 0101010101010101 = + Not a Number
eST(1) valid   4000 C90FDAA22168C235 =
+3.141592265110390E+000
eST(0) zero    0000 0000000000000000 =
+0.000000000000000E+000
>
```

Note the `e` at the beginning of the first, third, sixth, seventh, eighth, and ninth lines. Aside from this replacement of the `c` prefix by `e`, the emulator display is the same as the corresponding display for an 8087 chip.

CHAPTER 7

Managing Breakpoints

7

The CodeView debugger enables you to control program execution by setting breakpoints. A breakpoint is an address that stops program execution each time the address is encountered. By setting breakpoints at key addresses in your program, you can "freeze" program execution and examine the status of memory or expressions at that point.

The commands listed below control breakpoints:

Command	Action
Breakpoint Set (**BP**)	Sets a breakpoint and, optionally, a pass count and break commands
Breakpoint Clear (**BC**)	Clears one or more breakpoints
Breakpoint Disable (**BD**)	Disables one or more breakpoints
Breakpoint Enable (**BE**)	Enables one or more breakpoints
Breakpoint List (**BL**)	Lists all breakpoints

In addition to these commands, the Watchpoint (**WP**) and Tracepoint (**TP**) commands can be used to set conditional breakpoints (see Chapter 8, "Managing Watch Statements," for information on these two commands).

7.1 Breakpoint Set Command

The Breakpoint Set command (**BP**) creates a breakpoint at a specified address. Any time a breakpoint is encountered during program execution, the program halts and waits for a new command.

The CodeView debugger allows up to 20 breakpoints (0 through 19). Each new breakpoint is assigned to the next available number. Breakpoints remain in memory until you delete them or until you quit the debugger. They are not canceled when you restart the program. Because breakpoints are not automatically canceled, you are able to set up a complicated series of breakpoints, then execute through the program several times without resetting.

If you try to set a breakpoint at a comment line or other source line that does not correspond to code, the CodeView debugger displays the following message:

```
No code at this line number
```

Mouse

To set a breakpoint with the mouse, point to the source line or instruction where you want to set the breakpoint and then click the left button. The line will be displayed in high-intensity text and will remain so until you remove or disable the breakpoint.

Keyboard

To set a breakpoint with a keyboard command in window mode, move the cursor to the source line or instruction where you want to set a breakpoint. You may have to press F6 to move the cursor to the display window. When the cursor is on the appropriate source line, press F9. The line will be displayed in high-intensity text and will remain so until you remove or disable the breakpoint.

In sequential mode, the F9 key can be used to set a breakpoint at the current location. You must use the dialog version of the command to set a breakpoint at any other location.

Dialog

To set a breakpoint using a dialog command, enter a command line with the following syntax:

BP [[*address* [[*passcount*]] [["*commands*"]]]]

If no *address* is given, a breakpoint is created on the current source line in source mode or on the current instruction in assembly mode. You can specify the *address* in the *segment*:*offset* format or as a source line, a routine name, or a label. If you give an offset address, the code segment is assumed.

The dialog version of the command is more powerful than the mouse or keyboard version in that it allows you to give a *passcount* and a string of *commands*.

The *passcount* specifies the first time the breakpoint is to be taken. For example, if the pass count is 5, the breakpoint will be ignored the first four times it is encountered, and taken the fifth time. Thereafter, the breakpoint is always taken.

The *commands* are a list of dialog commands enclosed in quotation marks (" ") and separated by semicolons (;). For example, if you specify the commands as `""? code;T""`, the CodeView debugger will automatically display the value of the variable `code` and then execute the Trace command each time the breakpoint is encountered. The Trace and Display Expression commands are described in Chapter 5, "Executing Code," and Chapter 6, "Examining Data and Expressions," respectively.

In window mode, a breakpoint entered with a dialog command has exactly the same effect as one created with a window command. The source line or instruction corresponding to the breakpoint location is shown in high-intensity text.

In sequential mode, information about the current instruction will be displayed each time you execute to a breakpoint. The register values, the current instruction, and the source line may be shown, depending on the display mode. See Chapter 9, "Examining Code," for more information about display modes.

When a breakpoint address is shown in the assembly-language format, the breakpoint number will be shown as a comment to the right of the instruction. This comment appears even if the breakpoint is disabled (but not if it is deleted).

Examples

```
BP .19 10
>
```

The example above creates a breakpoint at line 19 of the current source file (or if there is no executable statement at line 19, at the first executable statement after line 19). The breakpoint is passed over nine times before being taken on the tenth pass.

```
BP STATS 10 "?COUNTER = COUNTER + 1;G"
>
```

The example above creates a breakpoint at the address of the routine `STATS`. The breakpoint is passed over nine times before being taken on the 10th pass. Each time execution stops for the breakpoint, the quoted commands are executed. The Display Expression command increments `COUNTER`, then the Go command restarts execution. If `COUNTER` is set to 0 when the breakpoint is set, this has the effect of counting the number of times the breakpoint is taken.

```
S-       ;* FORTRAN example - uses FORTRAN hexadecimal
notation
assembly
>BP #0A94
>G
AX=0006  BX=304A  CX=000B  DX=465D  SP=3050  BP=3050
SI=00BB  DI=40D1
DS=5064  ES=5064  SS=5064  CS=46A2  IP=0A94  NV UP EI PL NZ
NA PE NC
46A2:0A94 7205         JB      __chkstk+13 (0A9B)      ;BR1
>
```

The example above first sets the mode to assembly and then creates a breakpoint at the hexadecimal (offset) address `#0A94` in the default (CS) segment. (The same address would be specified as `0x0A94` with the C expression evaluator, and as `&H0A9` with the BASIC expression evaluator.) The Go command (**G**) is then used to execute to the breakpoint. Note that in the output to the Go command, the breakpoint number is shown as an assembly-language comment (`;BR1`) to the right of the current instruction. The Go command displays this output only in sequential mode; in window mode no assembly-language information appears.

7.2 Breakpoint Clear Command

The Breakpoint Clear command (**BC**) permanently removes one or more previously set breakpoints.

Mouse

To clear a single breakpoint with the mouse, point to the breakpoint line or instruction you want to clear. Breakpoint lines are shown in high-intensity text. Press the left mouse button. The line will be shown in normal text to indicate that the breakpoint has been removed.

To remove all breakpoints with the mouse, point to Run on the menu bar, press a mouse button and drag the highlight down to the Clear Breakpoints selection, then release the button.

Keyboard

To clear a single breakpoint with a keyboard command, move the cursor to the breakpoint line or instruction you want to clear. Breakpoint lines are shown in high-intensity text. Press F9. The line will be shown in normal text to indicate that the breakpoint has been removed.

To remove all breakpoints using a keyboard command, press ALT+R to open the Run menu, and then press ALT+C to select Clear Breakpoints.

Dialog

To clear breakpoints using a dialog command, enter a command line with the following syntax:

BC *list*
BC *

If *list* is specified, the command removes the breakpoints named in the list. The *list* can be any combination of integer values from 0 to 19. You can use the Breakpoint List command (**BL**) if you need to see the numbers for each existing breakpoint. If an asterisk (*****) is given as the argument, all breakpoints are removed.

Examples

```
BC 0 4 8
>
```

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

```
BC *
>
```

The example above removes all breakpoints.

7.3 Breakpoint Disable Command

The Breakpoint Disable command (**BD**) temporarily disables one or more existing breakpoints. The breakpoints are not deleted. They can be restored at any time using the Breakpoint Enable command (**BE**).

When a breakpoint is disabled in window mode, it is shown in the display window with normal text; when enabled, it is shown in high-intensity text.

NOTE *All disabled breakpoints are automatically enabled whenever you restart the program being debugged. The program can be restarted with the Start or Restart selection from the Run menu, or with the Restart dialog command (**L**). See Chapter 5, "Executing Code."*

Mouse

The Breakpoint Disable command cannot be executed with the mouse.

Keyboard

The Breakpoint Disable command cannot be executed with a keyboard command.

Dialog

To disable breakpoints with a dialog command, enter a command line with the following syntax:

BD *list*
BD *

If *list* is specified, the command disables the breakpoints named in the list. The *list* can be any combination of integer values from 0 to 19. Use the Breakpoint List command (**BL**) if you need to see the numbers for each existing breakpoint. If an asterisk (*) is given as the argument, all breakpoints are disabled.

The window commands for setting and clearing breakpoints can also be used to enable or clear disabled breakpoints.

Examples

```
BD 0 4 8
>
```

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

```
BD *
>
```

The example above disables all breakpoints.

7.4 Breakpoint Enable Command

The Breakpoint Enable command (**BE**) enables breakpoints that have been temporarily disabled with the Breakpoint Disable command.

Mouse

To enable a disabled breakpoint with the mouse, point to the source line or instruction of the breakpoint, and click Left. The line will be displayed in high-intensity text, and will remain so until you remove or disable the breakpoint. This is the same as creating a new breakpoint at that location.

Keyboard

To enable a disabled breakpoint using a keyboard command, move the cursor to the source line or instruction of the breakpoint, and press F9. The line is displayed in high-intensity text and remains so until you remove or disable the breakpoint. This is the same as creating a new breakpoint at that location.

Dialog

To enable breakpoints using a dialog command, enter a command line with the following syntax:

BE *list*
BE *

If *list* is specified, the command enables the breakpoints named in the list. The *list* can be any combination of integer values from 0 to 19. Use the Breakpoint List command (**BL**) if you need to see the numbers for each existing breakpoint. If an asterisk (*) is given as the argument, all breakpoints are enabled. The CodeView debugger ignores all or part of the command if you try to enable a breakpoint that is not disabled.

Examples

```
BE 0 4 8
>
```

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

```
BE*
>
```

The example above enables all disabled breakpoints.

7.5 Breakpoint List Command

The Breakpoint List command (**BL**) lists current information about all breakpoints.

Mouse

The Breakpoint List command cannot be executed with the mouse.

Keyboard

The Breakpoint List command cannot be executed with a keyboard command.

Dialog

To list breakpoints with a dialog command, enter a command line with the following syntax:

BL

The command displays the breakpoint number, the enabled status (`e` for "enabled," `d` for "disabled"), the address, the routine, and the line number.

If the breakpoint does not fall on a line number, an offset is shown from the nearest previous line number. The pass count and break commands are shown if they have been set. If no breakpoints are currently defined, nothing is displayed.

Example

```
BL
0 e 56C4:0105  _ARCTAN:10
1 d 56C4:011E  _ARCTAN:19              (pass = 10) "T;T"
2 e 56C4:00FD  _ARCTAN:9+6
>
```

In the example above, breakpoint 0 is enabled at address `56C4:0105`. This address is in routine `ARCTAN` and is at line `10` of the current source file. No pass count or break commands have been set.

Breakpoint 1 is currently disabled, as indicated by the `d` after the breakpoint number. It also has a pass count of 10, meaning that the breakpoint will not be taken until the 10th time it is encountered. The command string at the end of the line indicates that each time the breakpoint is taken, the Trace command will automatically be executed twice.

The line number for breakpoint 2 has an offset. The address is six bytes beyond the address for line 9 in the current source file. Therefore, the breakpoint was probably set in assembly mode, since it would be difficult to set a breakpoint anywhere except on a source line in source mode.

CHAPTER 8

8 Managing Watch Statements

Watch Statement commands are among the Microsoft CodeView debugger's most powerful features. They enable you to set, delete, and list watch statements. Watch statements describe expressions or areas of memory to watch. Some watch statements specify conditional breakpoints, which depend upon the value of the expression or memory area.

Syntax for each CodeView command is always the same, regardless of the expression evaluator; however, the method for specifying an *argument* may vary with the language. Therefore, each example in this chapter is repeated with C, FORTRAN, and BASIC arguments. The sample screens throughout the text that present these examples feature BASIC. At the end of this chapter are C and FORTRAN sample screens that incorporate all the previous examples (except for Watch Delete and Watch List).

8.1 Watch Statement Commands

The Watch Statement commands are summarized below:

Command	Action
Watch (**W**)	Sets an expression or range of memory to be watched
Watchpoint (**WP**)	Sets a conditional breakpoint that will be taken when the expression becomes nonzero (true)

Tracepoint (**TP**)	Sets a conditional breakpoint that will be taken when a given expression or range of memory changes
Watch Delete (**Y**)	Deletes one or more watch statements
Watch List (**W**)	Lists current watch statements

Watch statements, like breakpoints, remain in memory until you specifically remove them or quit the CodeView debugger. They are not canceled when you restart the program being debugged. Therefore, you can set a complicated series of watch statements once and then execute through the program several times without resetting.

In window mode, Watch Statement commands can be entered either in the dialog window or with menu selections. Current watch statements are shown in a watch window that appears between the menu bar and the source window.

In sequential mode, the Watch, Tracepoint, and Watchpoint commands can be used, but since there is no watch window, you cannot see the watch statements and their values. You must use the Watch List command to examine the current watch statements.

To set a watch statement containing a local variable, you must be in the function where the variable is defined. If the current line is not in the function, the CodeView debugger displays the message `UNKNOWN SYMBOL`. When you exit from a function containing a local variable referenced in a watch statement, the value of the statement is displayed as `UNKNOWN SYMBOL`. When you reenter the function, the local variable will again have a value. With the C and FORTRAN expression evaluators, you can avoid this limitation by using the period operator to specify both the function and the variable. For example, enter `main.x` instead of just `x`.

8.2 Setting Watch-Expression and Watch-Memory Statements

The Watch command is used to set a watch statement that specifies an expression (watch-expression statement) or a range of addresses in memory (watch-memory statement). The value or values specified by this watch statement are shown in the watch window. The watch window is updated to show new values each time the value of the watch statement changes during program execution. Since the watch window does not exist in sequential mode, you must use the Watch List command to examine the values of watch statements.

When setting a watch expression, you can specify the format in which the value will be displayed. Type the expression followed by a comma and a format specifier. If you do not give a format specifier, the CodeView debugger displays the value in a default format. See Section 6.1, "Display Expression Command," for more information about type specifiers and the default format.

NOTE *If your program directly accesses absolute addresses used by IBM or IBM-compatible computers, you may sometimes get unexpected results with the Display Expression and Dump commands. However, the Watch command will usually show the correct values. This problem can arise if the CodeView debugger and your program begin to use the same memory location.*

The problem often occurs when a program reads data directly from the screen buffer of the display adapter. If you have an array called `screen` *that is initialized to the starting address of the screen buffer, the command* `DB screen L 16` *will display data from the CodeView display rather than from the display of the program you are debugging. The command* `WB screen L 16` *will display data from the program's display (provided screen swapping or screen flipping was specified at start-up). The Watch command behaves differently from the Dump command because watch-statement values are updated during program execution, and any values read from the screen buffer will be taken from the output screen rather than from the debugging screen.*

Mouse

To set a watch-expression statement using the mouse, point to Watch on the menu bar, press a mouse button and drag the highlight down to the Add Watch selection, and then release the button. A dialog box appears, asking for the expression to be watched. Type the expression and press the ENTER key or a mouse button.

You cannot use the mouse version of the command to specify a range of memory to be watched, as you can with the dialog version.

Keyboard

To set a watch-expression statement with a keyboard command, press ALT+W to open the Watch menu, and then type **A** (uppercase or lowercase) to select Add Watch. You can also select the Add Watch command directly by pressing CONTROL+W. A dialog box appears, asking for the expression to be watched. Type the expression and press the ENTER key.

You cannot use the keyboard version of the command to specify a range of memory to be watched, as you can with the dialog version.

Dialog

To set a watch-expression statement with a dialog command, enter a command line with the following syntax:

W? *expression*[[,*format*]]

To set a watch-memory command with a dialog command, enter a command line with the following syntax:

W[[*type*]] *range*

An *expression* used with the Watch command can be either a simple variable or a complex expression using several variables and operators. The expression should be no longer than the width of the watch window. The characters permitted for *format* correspond to format arguments used in a C **printf** function call. See Section 6.1, "Display Expression Command," for more information on format arguments.

When watching a memory location, *type* is a one-letter size specifier from the following list:

Specifier	Size
None	Default type
B	Byte
A	ASCII
I	Integer (signed decimal word)
U	Unsigned (unsigned decimal word)
WP	Word
D	Double word
S	Short real
L	Long real
T	10-byte real

If no type size is specified, the default type used is the last type used by a Dump, Enter, Watch Memory, or Tracepoint Memory command. If none of these commands has been used during the session, the default type is byte.

The data will be displayed in a format similar to that used by the Dump commands (see Section 6.1, "Display Expression Command," for more information on format arguments). The *range* can be any length, but only one line of data will be displayed in the watch window. If you do not specify an ending address for the range, the default range is one object.

Examples

The following three examples display watch statements in the watch window.

```
W? n
```

The example above displays the current value of the variable `n`.

```
W? higher * 100
```

The example above displays the value of the expression `higher * 100`.

```
WL chance
```

The example above displays the value of `chance`, a double-precision floating-point variable. Exactly how `chance` is stored in memory is shown first. (The command `W? chance` would display the value of `chance` but not any actual bytes of memory.)

These commands, entered while debugging a BASIC program, produce the watch window in Figure 8.1. Corresponding C and FORTRAN examples are included with other commands in language-specific sections at the end of the chapter.

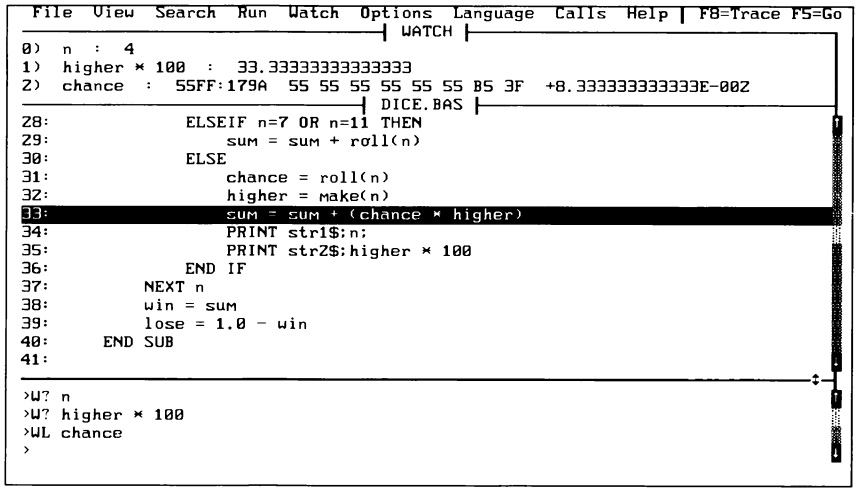


Figure 8.1 Watch Statements in the Watch Window

8.3 Setting Watchpoints

The Watchpoint command is used to set a conditional breakpoint called a watchpoint. A watchpoint breaks program execution when the expression described by its watch statement becomes true. You can think of watchpoints as "break when" points, since the break occurs when the specified expression becomes true (nonzero).

A watch statement created by the Watchpoint command describes the expression that will be watched and compared to 0. The statement remains in memory until you delete it or quit the CodeView debugger. Any valid CodeView expression can be used as the watchpoint expression as long as the expression is not wider than the watch window.

In window mode, watchpoint statements and their values are displayed in high-intensity text in the watch window. In sequential mode, there is no watch window, so the values of watchpoint statements can only be displayed with the Watch List command (see Section 8.6 "Listing Watchpoints and Tracepoints," for more information).

Although watchpoints can be any valid CodeView expression, the command works best with expressions that use the relational operators (such as < and > for C and BASIC, or **.LT.** and **.GT.** for FORTRAN). Relational expressions always evaluate to false (zero) or true (nonzero). Care must be taken with other kinds of expressions when they are used as watchpoints, because the watchpoints will break execution whenever they do not equal precisely zero. For example, your program might use a loop variable `I`, which ranges from 1 to 100. If you entered `I` as a watchpoint, then it would always suspend program execution, since `I` is never equal to 0. However, the relational expression `I>90` (or `I.GT.90`) would not suspend program execution until `I` exceeded 90.

Mouse

To set a watchpoint statement with the mouse, point to Watch on the menu bar, press a mouse button and drag the highlight down to the Watchpoint selection, and then release the button. A dialog box appears, asking for the expression to be watched. Type the expression and press the ENTER key or a mouse button.

Keyboard

To execute the Watchpoint command with a keyboard command, press ALT+W to open the Watch menu, and then press ALT+W to select Watchpoint. A dialog box appears, asking for the expression to be watched. Type the expression and press the ENTER key.

Dialog

To set a watchpoint using a dialog command, enter a command line with the following syntax:

WP? *expression*[[,*format*]]

The *expression* can be any valid CodeView expression (usually a relational expression). You can enter a format specifier, but there is little reason to do so, since the expression value is normally either 1 or 0.

Examples

The following dialog commands display two watch statements (watchpoints) in the watch window:

```
WP? higher > chance       ;* BASIC/C
WP? higher .gt. chance    ;* FORTRAN example
```

The examples above instruct the CodeView debugger to break execution when the variable `higher` is greater than the variable `chance`. (Note that BASIC and C happen to use the same syntax in this case, but FORTRAN uses its own.) After setting this watchpoint, you could use the Go command to execute until the condition becomes true.

```
WP? n=7 or n=11           ;* BASIC example
WP? n==7 || n==11         ;* C example
WP? n.eq.7 .or. n.eq.11   ;* FORTRAN example
```

The examples above instruct the CodeView debugger to break execution when the variable `n` is equal to 7 or 11.

NOTE *BASIC and C will each display a numerical result in response to a Boolean expression (0 being equivalent to false, nonzero to true). However, the corresponding FORTRAN condition will be displayed with either .TRUE. or .FALSE. in the watch window.*

These commands, entered while debugging a BASIC program, produce the watch window shown in Figure 8.2. Corresponding C and FORTRAN examples are included with other commands, at the end of the chapter.

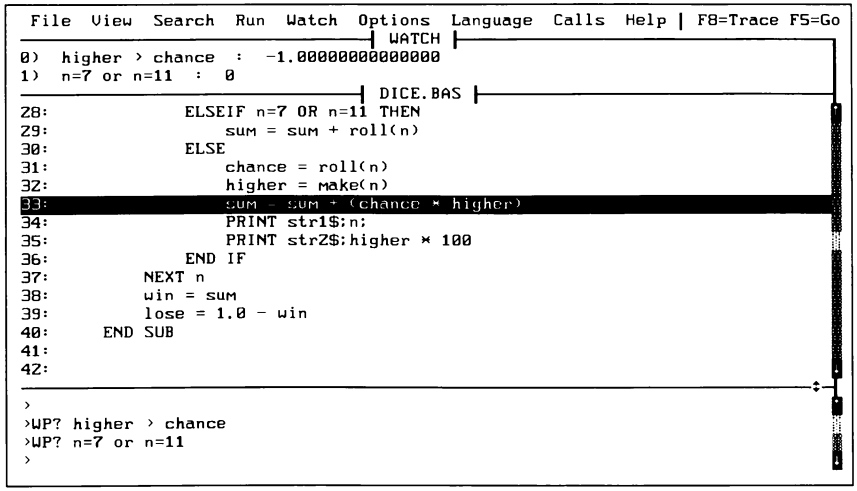


Figure 8.2 Watchpoints in the Watch Window

NOTE *Setting watchpoints significantly slows execution of the program being debugged. The CodeView debugger checks if the expression is true each time a source line is executed in source mode, or each time an instruction is executed in assembly mode. Be careful when setting watchpoints near large or nested loops. A loop that executes almost instantly when run from MS-DOS can take many minutes if executed from within the debugger with several watchpoints set.*

Tracepoints do not slow CodeView execution as much as watchpoints, so you should use tracepoints when possible. For example, although you can set a watchpoint on a Boolean variable (`WP? moving`*), a trace-point on the same variable (*`TP? moving`*) has essentially the same effect and does not slow execution as much.*

If you enter a seemingly endless loop, press CONTROL+BREAK or CONTROL+C to exit. You will soon learn the size of loop you can safely execute when watchpoints are set.

8.4 Setting Tracepoints

The Tracepoint command is used to set a conditional breakpoint called a tracepoint. A tracepoint breaks program execution when the value of a specified expression or range of memory changes.

The watch statement created by the Tracepoint command describes the expression or memory range to be watched and tested for change. The statement remains in memory until you delete it or quit the CodeView debugger.

In window mode, tracepoint statements and their values are shown in high-intensity text in the watch window. In sequential mode, there is no watch window, so the values of tracepoint statements can only be displayed with the Watch List command (see Section 8.5, "Listing Watchpoints and Tracepoints," for more information).

An expression used with the Tracepoint command must evaluate to an "lvalue." In other words, the expression must refer to an area of memory rather than a constant. Furthermore, the area of memory must be not more than 128 bytes in size. For example, `i==10` (which is similar to `I.EQ.10` in FORTRAN and `I=10` in BASIC) would be invalid because it is either 1 (true) or 0 (false) rather than a value stored in memory. The expression `sym1+sym2` is invalid because it is the calculated sum of the value of two memory locations. The expression `buffer` would be invalid if `buffer` is an array of 130 bytes, but valid if the array is 120 bytes. (However, using array names this way is not valid with BASIC modules because BASIC uses array descriptors.) Note that if `buffer` is declared as an array of 64 bytes, then the Tracepoint command given with the expression `buffer` checks all 64 bytes of the array. The same command given with the C expression `buffer[32]`, or `BUFFER(33)` in FORTRAN or BASIC, means that only one byte (the 33rd) will be checked. (Note that C and FORTRAN index the same element differently.)

NOTE *The following is relevant only to C programs.*

Register variables are not considered values. Therefore, if `i` *is declared as* `register int i`, *the command* `TP? i` *is invalid. However, you can still check for changes in the value of* `i`. *Use the Examine Symbols command to learn which register contains the value of* `i`.

Then learn the value of `i`. *Finally, set up a watchpoint to test the value. For example, use the following sequence of commands:*

```
XX? i
3A79:0264 int               div()
            SI        int               i
>?i
10
>WP? @SI!=10
>
```

When setting a tracepoint expression, you can specify the format in which the value will be displayed. Type the expression followed by a comma and a type specifier. If you do not give a type specifier, the CodeView debugger displays the value in a default format. See Section 6.1,"Display Expression Command," for more information about type specifiers and the default format.

Mouse

To set a tracepoint-expression statement with the mouse, point to Watch on the menu bar, press a mouse button and drag the highlight down to the Tracepoint selection, and then release the button. A dialog box appears, asking for the expression to be watched. Type the expression, and press the ENTER key or a mouse button.

You cannot specify a range of memory to be watched with the mouse version of the command, as you can with the dialog version.

Keyboard

To set a tracepoint-expression statement with a keyboard command, press ALT+W to open the Watch menu, and then press ALT+T to select Tracepoint. A dialog box appears, asking for the expression to be watched. Type the expression and press the ENTER key.

You cannot use the keyboard version of the command to specify a range of memory to be watched, as you can with the dialog version.

Dialog

To set a tracepoint with a dialog command, enter a command line with one of the following forms of syntax:

TP? *expression*,[[*format*]]
TP[[*type*]] *range*

The first syntax line above sets a tracepoint expression; the second line sets a tracepoint memory.

An *expression* used with the Tracepoint command can be either a simple variable or a complex expression using several variables and operators. The expression should not be longer than the width of the watch window. You can specify *format* using a C **printf** type specifier if you do not want the value to be displayed in the default format (decimal for integers or floating point for real numbers). See Section 6.1, "Display Expression Command," for more information on format arguments.

In the memory-tracepoint form, *range* must be a valid address range and *type* must be a one-letter memory-size specifier. If you specify only the start of the range, the CodeView debugger displays one object as the default.

Although no more than one line of data will be displayed in the watch window, the range to be checked for change can be any size up to 128 bytes.

The data will be displayed in the format used by the Dump commands (see Section 6.1, "Display Expression Command," for more information on format arguments). The valid memory-size specifiers are listed below:

Specifier	Size
None	Default type
B	Byte
A	ASCII
I	Integer (signed decimal word)
U	Unsigned (unsigned decimal word)
WP	Word
D	Double word
S	Short real
L	Long real
T	10-byte real

The default type used if no type size is specified is the last type used by a Dump, Enter, Watch Memory, or Tracepoint Memory command. If none of these commands has been used during the session, the default type is byte.

Examples

The two dialog commands below display watch statements (tracepoints) in the watch window.

```
TP? sum
```

The example above instructs the CodeView debugger to suspend program execution whenever the value of the variable `sum` changes.

```
TPB n
```

The example above instructs the CodeView debugger to suspend program execution whenever the first byte at the address of `n` changes; the address of this byte and its contents are displayed. The value of `n` may change because of a change in the *second* byte at the address of `n` ; but that change (by itself) would have no effect on this tracepoint.

These commands, entered while debugging a BASIC program, produce the watch window in Figure 8.3. Corresponding C and FORTRAN examples are included, with other commands, at the end of the chapter.

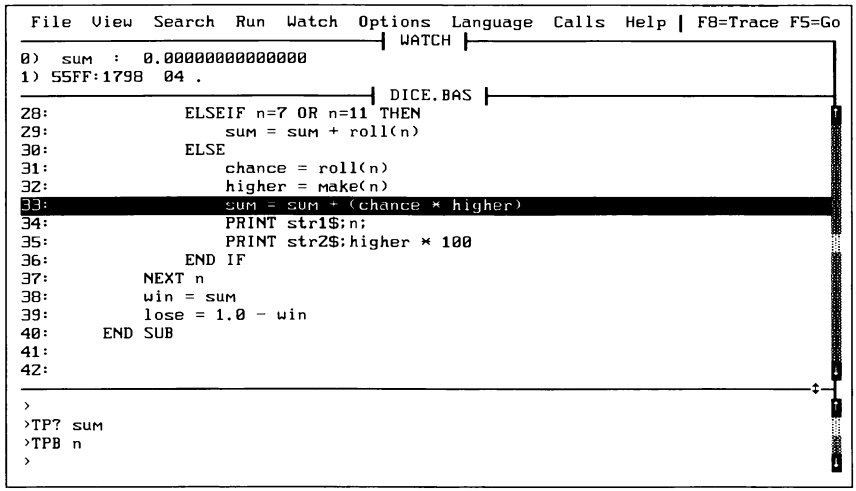


Figure 8.3 Tracepoints in the Watch Window

NOTE *Setting tracepoints significantly slows execution of the program being debugged. The CodeView debugger has to check to see if the expression or memory range has changed each time a source line is executed in source mode or each time an instruction is executed in assembly mode. However, tracepoints do not slow execution as much as do watchpoints.*

Be careful when setting tracepoints near large or nested loops. A loop that executes almost instantly when run from the MS-DOS operating system can take many minutes if executed from within the debugger with several tracepoints set. If you enter a seemingly endless loop, press CONTROL+BREAK or CONTROL+C to exit. Often you can tell how far you went in the loop by the value of the tracepoint when you exited.

8.5 Deleting Watch Statements

The Watch Delete command enables you to delete watch statements that were set previously with the Watch, Watchpoint, or Tracepoint command.

When you delete a watch statement in window mode, the statement disappears and the watch window closes around it. For example, if there are three watch statements in the window and you delete statement 1, the window is redrawn with one less line. Statement 0 remains unchanged, but statement 2 becomes statement 1. If there is only one statement, the window disappears.

Mouse

To delete a watch statement with the mouse, point to Watch on the menu bar, press a mouse button and drag the highlight down to the Delete Watch selection, and then release the button. A dialog box appears, containing all the watch statements. Point to the statement you want to delete and press the ENTER key or a mouse button. The dialog box disappears, and the watch window is redrawn without the watch statement.

You can also delete all the statements in the watch window at once, simply by selecting the Delete All selection.

Keyboard

To execute the Delete Watch command with a keyboard command, press ALT+W to open the Watch menu, and then type **D** (uppercase or lowercase) to select Delete Watch. You can also select the Delete Watch command directly by pressing CONTROL+U. A dialog box appears, containing all the watch statements. Use the UP and DOWN keys to move the cursor to the statement you want to delete, and then press the ENTER key. The dialog box disappears, and the watch window is redrawn without the watch statement.

You can also delete all the statements in the watch window at once, simply by selecting the Delete All selection. Do this by pressing **L** (upppercase or lowercase) after the Watch menu is open.

Dialog

To delete watch statements with a dialog command, enter a command line with the following syntax:

Y *number*

When you set a watch statement, it is automatically assigned a number (starting with 0). In window mode, the number appears to the left of the watch statement in the watch window. In sequential mode, you can use the Watch List (**W**) command to view the numbers of current watch statements.

You can delete existing watch statements by specifying the *number* of the statement you want to delete with the Delete Watch command. (The **Y** is a mnemonic for "yank.")

You can use the asterisk (*) to represent all watch statements.

Examples

```
Y 2
>
```

The command above deletes watch statement 2.

```
Y *
>
```

The command above deletes all watch statements and closes the watch window.

8.6 Listing Watchpoints and Tracepoints

The Watch List command lists all previously set watchpoints and tracepoints with their assigned numbers and their current values.

This command is the only way to examine current watch statements in sequential mode. The command has little use in window mode, since watch statements are already visible in the watch window as shown in Figure 8.4.

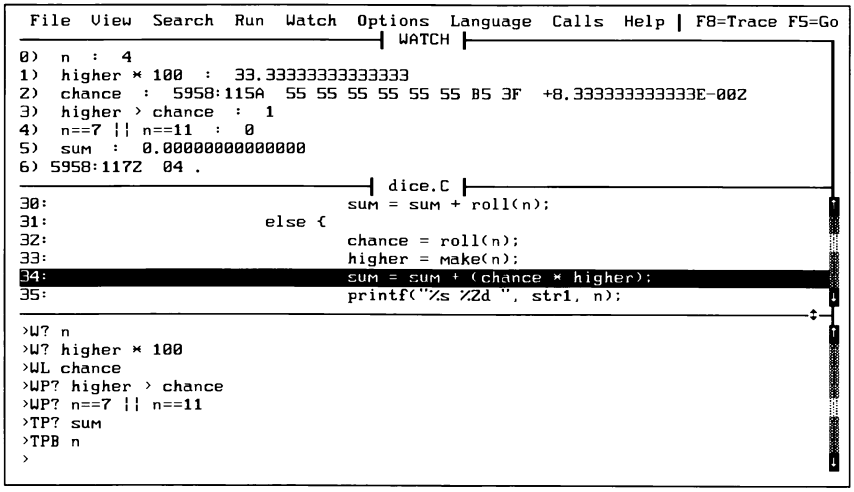


Figure 8.4 C Watch Statements

Mouse

The Watch List command cannot be executed with the mouse.

Keyboard

The Watch List command cannot be executed with a keyboard command.

Dialog

To list watch statements with a dialog command, enter a command line with the following syntax:

W

The display is the same as the display that appears in the watch window in window mode shown in Figure 8.5.

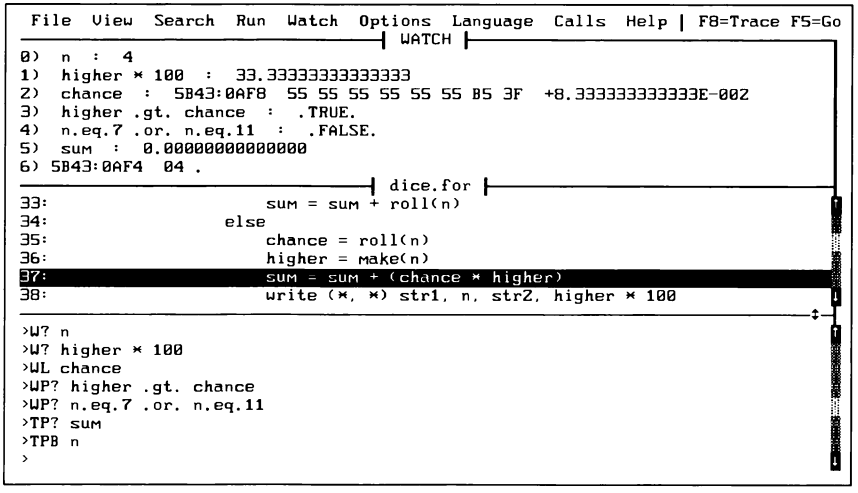


Figure 8.5 FORTRAN Watch Statements

Example

```
W
0) code,c  :  I
1) (float)letters/words,f  :  4.777778
2) 3F65:0B20  20 20 43 4F 55 4E 54 COUNT
3) lines==11 :  0
>
```

NOTE *The command letter for the Watch List command is the same as the command letter for the memory version of the Watch command when no memory size is given. The difference between the commands is that the Watch List command never takes an argument. The Watch command always requires at least one argument.*

8.7 C Examples

The seven examples shown previously in a BASIC screen would be entered in a C debugging session as follows:

The first three items in the watch window are simple watch statements. They display values but never cause execution to break.

The next two items are watchpoints; they cause execution to break whenever they evaluate to true (nonzero). The fourth item will break execution whenever

`higher` is greater than `chance`, and the fifth item will break execution whenever `n` is equal to 7 or 11.

The last two items are tracepoints, which cause execution to break whenever any bytes change within a specified area of memory. The sixth item breaks execution whenever the value of `sum` changes; the seventh item breaks execution whenever there is a change in the first byte at the address of `n`.

8.8 FORTRAN Examples

The seven examples shown previously in a BASIC screen would be entered in a FORTRAN debugging session as follows:

The first three items in the watch window are simple watch statements. They display values but never cause execution to break.

The next two items are watchpoints; they cause execution to break whenever they evaluate to true (nonzero). The fourth item will break execution whenever `higher` is greater than `chance`, and the fifth item will break execution whenever `n is equal to 7 or 11`.

The last two items are tracepoints, which cause execution to break whenever any bytes change within a specified area of memory. The sixth item breaks execution whenever the value of `sum` changes; the seventh item breaks execution whenever there is a change in the first byte at the address of `n`.

8.9 Assembly Examples

By default, assembly source modules are debugged with the C expression evaluator. Therefore, refer to the C examples for appropriate syntax for entering watch expressions.

In addition, certain C expressions tend to be more useful for debugging assembly modules. The following examples show some typical cases used with watch and tracepoint commands.

Examples

```
WW sp L 8
>WW bp L 8
>W? wo bp+4,d
>W? by bp-2,d
>TPW arr L 5
>
```

The first two examples watch a range of memory. The watch command `WW sp L 8` is particularly useful because it will cause the debugger to watch the stack

dynamically; the debugger will continually display the first eight words on the top of the stack as items are pushed and popped. The expression `WW bp L 8` is similar; it causes the debugger to watch the first eight words in memory pointed to by **BP** (the framepointer).

The third example, `W? wo bp+4,d`, is useful if you are using the stack to pass parameters. In this case, the position on the stack four bytes above **BP** holds one of three integer parameters. The **WO** operator returns the same value as the assembler expression `WORD PTR [bp+4]` ; the result is displayed in decimal.

You must use the expression `bp+4` in order to watch this parameter; you cannot specify a parameter by name. The assembler does not emit symbolic information for parameters. The fourth command, `W? by bp-2,d`, is similar to the third, but watches a local variable instead of watching a parameter. The operator **BY** returns the same value as the assembler expression `BYTE PTR [bp-2]`.

The final example sets a tracepoint on a range of memory, which corresponds to the first five words of the array `arr`. Range arguments for tracepoint and watch expressions are particularly useful for large data structures, such as arrays.

The five examples above produce the screen shown in Figure 8.6 when entered in a CodeView debugging session.

```
 File  View  Search  Run  Watch  Options  Language  Calls  Help | F8=Trace F5=Go
                                  WATCH
0)  sp L 8  :  531C:09AZ  0044 09B4 0037 0005 000F 001B 000F 0005      AX = 001B
1)  bp L 8  :  531C:09A4  09B4 0037 0005 000F 001B 000F 0005 001B      BX = 09AZ
2)  wo bp+4,d  :  5                                                    CX = 0044
3)  by bp-2,d  :  68                                                   DX = 00B0
4) 531F:0006  01 00 0Z 00 03 .....                                     SP = 09AZ
                                  test.ASM                             BP = 09A4
70:      ;  First parameter largest                                    SI = 0098
71:      ;                                                             DI = 0A8C
72:              mov     BYTE PTR [bp-Z],1  ; Load indicator value     DS = 531C
73:                                         ;  of 1 into local variabl ES = 531C
74:              jmp     SHORT finished     ;  and finish up           SS = 531C
75:      next_test:                                                    CS = 5ZD7
76:              mov     ax,[bp+8]          ; Load 3rd parm into ax    IP = 005D
77:              cmp     [bp+6],ax          ; If 2nd parm <= 3rd parm
78:              jle     last_test          :   go to last test          NV UP
79:      ;                                                             EI NG
                                                                       NZ AC
>WW sp L 8                                                             PE CY
>WW bp L 8
>W? wo bp+4,d                                                          SS:09AA
>W? by bp-2,d                                                             000F
>TPB arr L 5
>
```

Figure 8.6 Assembly Watch Statements

CHAPTER 9 Examining Code

9

Several CodeView commands allow you to examine program code or data related to code. The following commands are discussed in this chapter:

Command	Action
Set Mode (**S**)	Sets format for code displays
Unassemble (**U**)	Displays assembly instructions
View (**V**)	Displays source lines
Current Location (**.**)	Displays the current location line
Stack Trace (**K**)	Displays routines or procedures

9.1 Set Mode Command

The Set Mode command sets the mode in which code is displayed. The two basic display modes are source mode in which the program is displayed as source lines, and assembly mode in which the program is displayed as assembly-language instructions. These two modes can be combined in mixed mode in which the program is displayed with both source lines and assembly-language instructions.

In sequential mode, there are three display modes: source, assembly, and mixed. These modes affect the output of commands that display code (Register, Trace, Program Step, Go, Execute, and Unassemble).

In window mode, these same display modes are available, but affect what kind of code appears in the display window.

Source and mixed modes are only available if the executable file contains symbols in the CodeView format. Programs that do not contain symbolic information (including all .COM files) are displayed in assembly mode.

Mouse

To set the display mode with the mouse, point to View on the menu bar, press a mouse button and drag the highlight to either the Source selection for source mode, the Mixed selection for mixed mode, or the Assembly selection for assembly mode. Then release the button.

You can further control the display of assembly-language instructions by making selections from the Options menu. See Section 2.1.3.6 for more information.

Keyboard

To change the display mode with a keyboard command, press F3. This will rotate the mode to the next setting; you may need to press F3 twice to get the desired mode. This command works in either window or sequential mode. In sequential mode, the word `source`, `mixed`, or `assembly` is displayed to indicate the new mode.

Dialog

To set the display mode from the dialog window, enter a command line with the following syntax:

S[[+ | – | &]]

If the plus sign is specified (**S+**), source mode is selected, and the word `source` is displayed.

If the minus sign is specified (**S–**), assembly mode is selected, and the word `assembly` is displayed. In window mode, the display will include any assembly options, except the Mixed Source option, previously toggled on from the Options menu. The Mixed Source option is always turned off by the **S–**command.

If the ampersand is specified (**S&**), mixed mode is selected, and the word `mixed` is displayed. In window mode, the display will include any assembly options previously toggled on from the Options menu. In addition, the Mixed Source option will be turned on by the **S&** command.

If no argument is specified (**S**), the current mode (`source`, `assembly`, or `mixed`) is displayed.

The Unassemble command in sequential mode is an exception in that it displays mixed, source, and assembly with both the source (**S+**) and mixed (**S&**) modes.

When you enter the dialog version of the Set Mode command, the CodeView outputs the name of the new display mode: `source`, `assembly`, or `mixed`.

Examples

```
S+
source
>S-
assembly
>S&
mixed
>
```

The examples above show the source mode being changed to `source`, `assembly`, and `mixed`. In window mode, the commands change the format of the display window. In sequential mode, the commands change the output from the commands that display code (Register, Trace, Program Step, Go, Execute, and Unassemble). See the sections below on individual commands for examples of how they are affected by the display mode.

9.2 Unassemble Command

The Unassemble command displays the assembly-language instructions of the program being debugged. It is most useful in sequential mode, where it is the only method of examining a sequence of assembly-language instructions. In window mode, it can be used to display a specific portion of assembly-language code in the display window.

Occasionally, code similar to the following will be displayed:

```
FE30   ???   Byte Ptr  [BX + SI]
```

If you attempt to unassemble data, the CodeView debugger may display meaningless instructions.

Mouse

The Unassemble command has no direct mouse equivalent, but you can view unassembled code at any time by changing the mode to assembly or mixed (see Section 9.1, "Set Mode Command," for more information).

Keyboard

The Unassemble command has no direct keyboard equivalent, but you can view unassembled code at any time by changing the mode to assembly or mixed (see Section 9.1, "Set Mode Command," for more information).

Dialog

To display unassembled code using a dialog command, enter a command line with the following syntax:

U [[*address* | *range*]]

The effect of the command varies depending on whether you are in sequential or window mode.

In sequential mode, if you do not specify *address* or *range*, the disassembled code begins at the current unassemble address and shows the next eight lines of instructions. The unassemble address is the address of the instruction after the last instruction displayed by the previous Unassemble command. If the Unassemble command has not been used during the session, the unassemble address is the current instruction.

If you specify an *address*, the disassembly starts at that address and shows the next eight lines of instructions. If you specify a *range*, the instructions within the range will be displayed.

The sequential mode format of the display depends on the current display mode (see Section 9.1, "Set Mode Command," for more information). If the mode is source (S+) or mixed (S&), the CodeView debugger displays source lines mixed with unassembled instructions. One source line is shown for each corresponding group of assembly-language instructions. If the display mode is assembly, only assembly-language instructions are shown.

In window mode, the Unassemble command changes the mode of the display window to assembly. The display format will reflect any options previously set from the Options menu. There is no output to the dialog window. If *address* is given, the instructions in the display window will begin at the specified address. If *range* is given, only the starting address will be used. If no argument is given, the debugger scrolls down and displays the next screen of assembly-language instructions.

NOTE *The 80286 protected-mode mnemonics (also available with the 80386) cannot be displayed with the Unassemble command.*

Examples

```
>S&
mixed
>U 0x11
49D0:0011 35068E        XOR   AX,__sqrtjmptab+8cd4 (8E06)
49D0:0014 189A2300      SBB   Byte Ptr [BP+SI+0023],BL
49D0:0018 FC            CLD
49D0:0019 49            DEC   CX
49D0:001A CD351ED418    INT   35 ;FSTP    DWord Ptr
[__fpinit+ee (18D4)]
49D0:001F CD3D          INT   3D ;FWAIT
7:              A = 0.0
49D0:0021 CD35EE        INT   35 ;FLDZ
```

The sequential mode example above sets the mode to mixed and unassembles eight lines of machine code, plus whatever source lines are encountered within those lines. The display would be the same if the mode were source.

The example is taken from a FORTRAN debugging session, but produces results similar to what would be produced using the same commands with a C or BASIC program.

```
>S-
assembly
>U 0x11
49D0:0011 35068E        XOR   AX,__sqrtjmptab+8cd4 (8E06)
49D0:0014 189A2300      SBB   Byte Ptr [BP+SI+0023],BL
49D0:0018 FC            CLD
49D0:0019 49            DEC   CX
49D0:001A CD351ED418    INT   35 ;FSTP    DWord Ptr
[__fpinit+ee (18D4)]
49D0:001F CD3D          INT   3D ;FWAIT
49D0:0021 CD35EE        INT   35 ;FLDZ
>
```

The sequential mode example above sets the mode to assembly and repeats the same command.

9.3 View Command

The View command displays the lines of a text file (usually a source module or include file). It is most useful in sequential mode where it is the only method of examining a sequence of source lines. In window mode the View command can be used to page through the source file or to load a new source file.

Mouse

To load a new source file with the button, point to File on the menu bar, press a mouse button and drag the highlight to the Load selection, then release the button. A dialog box appears, asking for the name of the file you wish to load. Type the name of the file, and press ENTER or a mouse button. The new file appears in the display window.

The paging capabilities of the View command have no direct mouse equivalent, but you can move about in the source file by pointing to the up or down arrows on the scroll bars and then clicking different mouse buttons. See Section 2.1.2.2, "Controlling Program Execution with the Mouse," for more information.

Keyboard

To load a new source file with a keyboard command, press ALT+F to open the File menu, then press L to select Load. A dialog box appears, asking for the name of the file you wish to load. Type the name of the file, and press ENTER. The new file appears in the display window.

The paging capabilities of the View command have no direct keyboard equivalent, but you can move about in the source file by first putting the cursor in the display window with the F6 key, then pressing the PGUP, PGDN, HOME, END,

UP ARROW, and DOWN ARROW keys. See Section 2.1.1.3, "Controlling Program Execution with Keyboard Commands," for more information.

Dialog

To display source lines using a dialog command, enter a command line with the following syntax:

V [[*expression*]]

Since addresses for the View command are often specified as a line number (with an optional source file), a more specific syntax for the command would be as follows:

V [[**.**[[*filename*:]]*linenumber*]]

The effect of the command varies, depending on whether you are in sequential or window mode.

In sequential mode, the View command displays eight source lines. The starting source line is one of the following:

- The current source line if no argument is given.
- The specified *linenumber*. If *filename* is given, the specified file is loaded, and the *linenumber* refers to lines in it.
- The address that *expression* evaluates to. For example, *expression* could be a procedure name or an address in the *segment*:*offset* format. The code segment is assumed if no segment is given.

In sequential mode, the View command is not affected by the current display mode (source, assembly, or mixed); source lines are displayed without regard to mode.

In window mode, if you enter the View command while the display mode is assembly, the CodeView debugger will automatically switch back to source mode. If you give *linenumber* or *expression*, the display window will be redrawn so that the source line corresponding to the given *address* will appear at the top of the source window. If you specify a *filename* with a *linenumber*, the specified file will be loaded.

If you enter the View command with no arguments, the display will scroll down one line short of a page; that is, the source line that was at the bottom of the window will be at the top.

NOTE *The View command with no argument is similar to pressing the PGDN key or clicking Right on the down arrow with the mouse. The difference is that pressing the PGDN key enables you to scroll down one more line.*

Examples

```
V BUBBLE                ;* Example 1, FORTRAN source code
51:           IF (N .LE. 1) GOTO 101
52:           DO 201 I = 1,N-1
53:           DO 301 J = I + 1,N
54:           IF (X(I) .LE. X(J)) GOTO 301
55:           TEMP = X(I)
56:           X(I) = X(J)
57:           X(J) = TEMP
58:       301 CONTINUE
```

Example 1 (shown in sequential mode) displays eight source lines, beginning at routine `BUBBLE`.

```
V .math.c:30    ;* Example 2, C source code
30:                register int j;
31:
32:                for (j = q; j >= 0; j--)
33:                        if (t[j] + p [j] > 9) {
34:                                p[j] += t[j] - 10;
35:                                p[j-1] += 1;
36:                        } else
37:                                p[j] += t[j];
>
```

Example 2 loads the source file `math.c` and displays eight source lines starting at line `30`.

All forms of the View command are supported with all languages that work with the CodeView debugger.

9.4 Current Location Command

The Current Location command displays the source line or assembly-language instruction corresponding to the current program location.

Mouse

The Current Location command cannot be executed with the mouse.

Keyboard

The Current Location command cannot be executed with a keyboard command.

Dialog

To display the current location line using a dialog command, enter a command line with the following syntax (a period only):

```
.
```

In sequential mode, the command displays the current source line. The line is displayed regardless of whether the current debugging mode is source or assembly. If the program being debugged has no symbolic information, the command will be ignored.

In window mode, the command puts the current program location (marked with reverse video or a contrasting color) in the center of the display window. The display mode (source or assembly) will not be affected. This command is useful if you have scrolled through the source code or assembly-language instructions so that the current location line is no longer visible.

For example, if you are in window mode and have executed the program being debugged to somewhere near the start of the program, but you have scrolled the display to a point near the end, the Current Location command returns the display to the current program location.

Example

```
.
MINDAT = 1.0E6
>
```

The example above illustrates how to display the current source line in sequential mode. The same command in window mode would not produce any output, but it could change the text that is shown in the display window.

9.5 Stack Trace Command

The Stack Trace command allows you to display routines that have been called during program execution (see note below). The first line of the display shows the name of the current routine. The succeeding lines (if any) list any other routines that were called to reach the current address. The dialog version of the Stack Trace command also displays the source lines where each routine was called.

For each routine, the values of any arguments are shown in parentheses after the routine name. Values are shown in the current radix (the default is decimal).

The term "stack trace" is used because as each routine is called, its address and arguments are stored on (pushed onto) the program stack. Therefore, tracing through the stack shows the currently active routines. With C and FORTRAN programs, the **main** routine will always be at the bottom of the stack. With BASIC programs, the main program is not listed on the stack because BASIC programs have no standard label (such as **main**) corresponding to the first line of a program. Only routines called by the main program will be displayed. In assembly-language programs, the bottom routine displayed in the stack trace is **astart** instead of **main**.

NOTE *This discussion uses the term "routines," which is a general term for functions (C, FORTRAN, Pascal), subroutines (FORTRAN), procedures (Pascal), and subprograms and function procedures (BASIC)–each of which uses the stack to transfer control to an independent program unit. In assembly mode, the term "procedure" may be more accurate.* ***GOSUB*** *and* ***DEF FN*** *routines in BASIC will not work with the Stack Trace command, since they do not follow the same convention for setting up the stack.*

If you are using the CodeView debugger to debug assembly-language programs, the Stack Trace command will work only if procedures were called with the calling convention used by Microsoft languages. This calling convention is explained in the Microsoft Mixed-Language Programming Guide.

The Stack Trace command does not work reliably until you execute at least to the beginning of the main procedure. The main procedure sets up the frame pointer (**BP**), which CodeView uses to locate parameters, local variables, and return addresses. If your main module is written in assembly, you must execute at least to the beginning of the first procedure called. Furthermore, your procedures must follow the standard Microsoft calling conventions.

Mouse

To view a stack trace with the mouse, point to Calls on the menu bar and press a mouse button. The Calls menu will appear, showing the current routine at the top and other routines below it in the reverse order in which they were called; for example, the first routine called (which is always **main** in a C or FORTRAN program) will be at the bottom. The values of any routine arguments will be shown in parentheses following the routines.

If you want to view one of the routines that was previously called, select the routine by dragging down the highlight to the routine you wish to see, then releasing the mouse button. (You can also select a routine by clicking a selection once the menu is open.) The effect of selecting a routine in the Calls menu is to cause the debugger to display that routine. The cursor will be on the last statement executed in the routine.

Keyboard

To view a stack trace with a keyboard command, press ALT+C to open the Calls menu. The menu will show the current routine at the top and other routines below it in the reverse order in which they were called; for example, the first routine called will be at the bottom. The values of any routine arguments will be shown in parentheses following the routine.

If you want to view one of the routines that was previously called, select the routine by moving the cursor with the arrow keys and then pressing ENTER, or by typing the number or letter to the left of the routine. The effect of selecting a routine in the Calls menu is to cause the debugger to display that routine. The cursor will be on the last statement that was executed in the routine.

Dialog

To display a stack trace with a dialog command, enter a command line with the following syntax:

K

The output from the Stack Trace dialog command lists the routines in the reverse order in which they were called. The arguments to each routine are shown in parentheses. Finally, the line number from which the routine was called is shown.

You can enter the line number as an argument to the View or Unassemble command if you want to view code at the point where the routine was called.

In window mode, the output from the Stack Trace dialog command appears in the dialog window.

FORTRAN Example

```
K
ANALYZE(67,0), line 94
COUNTWORDS(0,512), line 73
MAIN(2,5098), line 42
>
```

In the example above, the first line of output indicates that the current routine is `ANALYZE`. Its first argument currently has a decimal value of `67`, and its second argument, a value of `0`. The current location in this routine is `line 94`.

The second line indicates that `ANALYZE` was called by `COUNTWORDS`, and that its arguments have the values `0` and `512`. Routine `ANALYZE` was called from `line 73` of routine `COUNTWORDS`.

Likewise, `COUNTWORDS` was called from `line 42` of `MAIN`, and its arguments have the values `2` and `5098`.

If the radix had been set to 16 or 8 using the Radix (**N**) command, the arguments would be shown in that radix. For example, the last line would be shown as `MAIN( 2,13ea )` in hexadecimal or `MAIN( 2,11752 )` in octal.

C Example

```
K
analyze(67,0), line 94
countwords(0,512), line 73
main(2,5098)
>
```

As with the FORTRAN example, the example above shows the routines on the stack in the reverse order in which they were called. Since `analyze` is on the top, it has been called most recently; in other words, it is the current routine.

Each routine is shown with the arguments it was passed, along with the last source line that it had been executing. Note that `main` is shown with the command line arguments `argc` (which is equal to 2) and `argv` (which is a pointer equal to 5,098 decimal). Since the language is C, `main` will always be on the bottom of the stack.

BASIC Example

```
K
ROLL#(19122:6040)
MAKE#(19122:6040)
CALC(19122:5982, 19122:5990)
>
```

As with the FORTRAN example, the example above shows the routines on the stack in the reverse order in which they were called. Since `ROLL#` is on the top, it has been called most recently; in other words, it is the current routine.

Each routine is displayed along with the arguments by which it was passed. In BASIC, arguments passed to routines are always addresses.

This example shows some features peculiar to BASIC. First of all, there is no `MAIN` displayed because the BASIC compiler does not produce any such symbol. Furthermore, each routine will have a type tag if it is a function; the tag indicates what the function returns. `ROLL#` and `MAKE#` are both functions returning a double-precision floating point. A function that returned a short integer would have a `%` type tag. `CALC` has no type tag since it is a subprogram, and therefore does not return a value of any type.

CHAPTER 10
Modifying Code or Data

The CodeView debugger provides the following commands for modifying code or data in memory:

Command	Action
Assemble (**A**)	Modifies code
Enter (**E**)	Modifies memory, usually data
Register (**R**)	Modifies registers and flags
Fill Memory (**F**)	Fills a block of memory
Move Memory (**M**)	Copies one block of memory to another
Port Output (**O**)	Outputs a byte to a hardware port

These commands change code temporarily. You can use the alterations for testing in the CodeView debugger, but you cannot save them or permanently change the program. To make permanent changes, you must modify the source code and recompile.

10.1 Assemble Command

The Assemble command assembles 8086-family (8086, 8087, 8088, 80186, 80286, 80287, and 80286 and 80386 unprotected) instruction mnemonics and places the resulting instruction code into memory at a specified address. The only 8086-family mnemonics that cannot be assembled are 80286 protected-mode mnemonics. In addition, the debugger will also assemble 80286 instructions that utilize the expanded 386 registers.

NOTE *The effects of the Assemble command are temporary. Any instructions that you assemble are lost as soon as you exit the program.*

The instructions you assemble are also lost when you restart the program with the Start or Restart command because the original code is reloaded on top of memory you may have altered.

To test the results of an Assemble command, you may need to manipulate the IP register (and possibly the CS register) to the starting address of the instructions you have assembled. If you do this, you must use the Current Line command (.) to reset the debugger's internal variables so that it will trace properly.

Mouse

The Assemble command cannot be executed with the mouse.

Keyboard

The Assemble command cannot be executed with a keyboard command.

Dialog

To assemble code using a dialog command, enter a command line with the following syntax:

A [[*address*]]

If *address* is specified, the assembly starts at that address; otherwise the current assembly address is assumed.

The assembly address is normally the current address (the address pointed to by the CS and IP registers). However, when you use the Assemble command, the assembly address is set to the address immediately following the last assembled instruction. When you enter any command that executes code (Trace, Program Step, Go, or Execute), the assembly address is reset to the current address.

When you type the Assemble command, the assembly address is displayed. The CodeView debugger then waits for you to enter a new instruction in the standard 8086-family instruction-mnemonic form. You can enter instructions in uppercase, lowercase, or both.

To assemble a new instruction, type the desired mnemonic and press ENTER. The CodeView debugger assembles the instruction into memory and displays the next available address. Continue entering new instructions until you have assembled all the instructions you want. To conclude assembly and return to the CodeView prompt, press ENTER only.

If an instruction you enter contains a syntax error, the debugger displays the message `^ Syntax error`, redisplays the current assembly address, and waits for you to enter a correct instruction. The caret symbol in the message will point to the first character the CodeView debugger could not interpret.

The following eight principles govern entry of instruction mnemonics:

1. The far-return mnemonic is **RETF**.

2. String mnemonics must explicitly state the string size. For example, **MOVSW** must be used to move word strings, and **MOVSB** must be used to move byte strings.

3. The CodeView debugger automatically assembles short, near, or far jumps and calls, depending on byte displacement to the destination address. These may be overridden with the **NEAR** or **FAR** prefix, as shown in the following examples:

```
JMP     0x502
JMP     NEAR 0x505
JMP     FAR  0x50A
```

   The **NEAR** prefix can be abbreviated to **NE**, but the **FAR** prefix cannot be abbreviated. The examples above use the C notation for hexadecimal numbers. If the FORTRAN option were selected, you would enter the operands as `#502`, `#505`, and `#50A`; if the BASIC option were selected, you would enter them as `&H502`, `&H505`, and `&H50A`.

4. The CodeView debugger cannot determine whether some operands refer to a word memory location or to a byte memory location. In these cases, the data type must be explicitly stated with the prefix **WORD PTR** or **BYTE PTR**. Acceptable abbreviations are **WO** and **BY**. Examples are shown below:

```
MOV     WORD PTR [BP],1
MOV     BYTE PTR [SI-1],symbol
MOV     WO PTR [BP],1
MOV     BY PTR [SI-1],symbol
```

5. The CodeView debugger cannot determine whether an operand refers to a memory location or to an immediate operand. The debugger uses the convention that operands enclosed in square brackets refer to memory. Two examples are shown below:

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

The first statement moves 21 hexadecimal into AX. The second statement moves the data at offset 21 hexadecimal into AX. Both statements use the FORTRAN notation for the hexadecimal number 21. If the C option were selected, this number would be represented as 0x21, and if the BASIC option were selected, the number would be represented as &H21.

6. The CodeView debugger supports all forms of indirect register instructions, as shown in the following examples:

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

7. All instruction-name synonyms are supported. If you assemble instructions and then examine them with the Unassemble command (**U**), the CodeView debugger may show synonymous instructions, rather than the ones you assembled, as shown in the following examples:

```
LOOPZ   &H100
LOOPE   &H100
JA      &H200
JNBE    &H200
```

The examples above use the BASIC hexadecimal notation. Instead of using the &H prefix, you would use 0x with the C option selected, and # with the FORTRAN option selected.

8. Do not assemble and execute 8087 or 80287 instructions if your system is not equipped with one of these math coprocessor chips. If you try to execute the **WAIT** instruction without the appropriate chip, for example, your system will crash.

Example

```
U #40 L 1
39B0:0040 89C3            MOV         BX,AX
>A #40
39B?0:0040 MOV      CX,AX
39B0:0042
>U #40 L 1
39B0:0040 89C1            MOV         CX,AX
>
```

The example above (in FORTRAN notation) modifies the instruction at address 40 hexadecimal so that it moves data into the CX register instead of the BX register (40 hexadecimal is notated as 0x40 in C, and as &H40 in BASIC). The Unassemble command (**U**) is used to show the instruction before and after the assembly.

You can modify a portion of code for testing, as in the example, but you cannot save the modified program. You must modify your source code and recompile.

10.2 Enter Commands

The CodeView debugger has several commands for entering data to memory. You can use these commands to modify either code or data, though code can usually be modified more easily with the Assemble command (**A**). The Enter commands are listed below:

Command	Command Name
E	Enter (size is the default type)
EB	Enter Bytes
EA	Enter ASCII
EI	Enter Integers
EU	Enter Unsigned Integers
EW	Enter Words
ED	Enter Double Words
ES	Enter Short Reals
EL	Enter Long Reals
ET	Enter 10-Byte Reals

Mouse

The Enter commands cannot be executed with the mouse.

Keyboard

The Enter commands cannot be executed with keyboard commands.

Dialog

To enter data (or code) to memory with a dialog command, enter a command line with the following syntax:

E[[*type*]] *address* [[*list*]]

The *type* is a one-letter specifier that indicates the type of the data to be entered. The *address* indicates where the data will be entered. If no segment is given in the address, the data segment (DS) is assumed.

The *list* can consist of one or more expressions that evaluate to data of the size specified by *type* (the expressions in the list are separated by spaces). This data will be entered to memory at *address*. If one of the values in the list is invalid, an error message will be displayed. The values preceding the error are entered; values at and following the error are not entered.

The expressions in the list are evaluated in the current radix, regardless of the size and type of data being entered. For example, if the radix is 10 and you give the value 10 in a list with the Enter Words command, the decimal value 10 will be entered even though word values are normally entered in hexadecimal. This means that the Enter Words, Enter Integers, and Enter Unsigned Integers commands are identical when used with the list method since two-byte data are being entered for each command.

If *list* is not given, the CodeView debugger will prompt for values to be entered to memory. Values entered in response to prompts are accepted in hexadecimal for the Enter Bytes, Enter ASCII, Enter Words, and Enter Double Words commands. The Enter Integers command accepts signed decimal integers, while the Enter Unsigned Integers command accepts unsigned decimal integers. The Enter Short Reals, Enter Long Reals, and Enter 10-Byte Reals commands accept decimal floating-point values.

With the prompting method of data entry, the CodeView debugger prompts for a new value at *address* by displaying the address and its current value. As explained below, you can then replace the value, skip to the next value, return to a previous value, or exit the command.

1. To replace the value, type the new value after the current value.
2. To skip to the next value, press the SPACEBAR. Once you have skipped to the next value, you can change its value or skip to the following value. If you pass the end of the display, the CodeView debugger displays a new address to start a new display line.
3. To return to the preceding value, type a backslash (\). When you return to the preceding value, the debugger starts a new display line with the address and value.
4. To stop entering values and return to the CodeView prompt, press ENTER. You can exit the command at any time.

Sections 10.2.1–10.2.10 discuss the Enter commands in order of the size of data they accept.

Examples

```
EW PLACE 16 32
```

The example above shows how to enter two word-sized values at the `PLACE` address.

```
EW PLACE

3DA5:0B20    00F3._
```

The example above illustrates the prompting method of entering data. When you supply the address where you want to enter data but supply no data, the CodeView debugger displays the current value of the address and waits for you to enter a new value. The underscore in this example and the examples below represents the CodeView cursor. You change the value `F3` to the new value 16 (10 hexadecimal) by typing `10` (without pressing ENTER yet). The value must be typed in hexadecimal for the Enter Words command, as shown below:

```
EW PLACE

3DA5:0B20   00F3.10_
```

You can then skip to the next value by pressing the SPACEBAR. The CodeView debugger responds by displaying the next value, as shown below:

```
EW PLACE

3DA5:0B20   00F3.10   4F20._
```

You can then type another hexadecimal value, such as `30`:

```
EW PLACE

3DA5:0B20   00F3.10   4F20.30_
```

To move to the next value, press the SPACEBAR.

```
EW PLACE

3DA5:0B20   00F3.10   4F20.30   3DC1._
```

Assume you realize that the last value entered, `30`, is incorrect. You really wanted to enter `20`. You could return to the previous value by typing a backslash. The CodeView debugger starts a new line, starting with the previous value. Note that the backslash is not echoed on the screen:

```
EW PLACE

3DA5:0B20   00F3.10   4F20.30   3DC1.
3DA5:0B22   0030._
```

Type the correct value, `20`:

```
EW PLACE

3DA5:0B20   00F3.10   4F20.30   3DC1.
3DA5:0B22   0030.20_
```

If this is the last value you want to enter, press ENTER to stop. The CodeView prompt reappears, as shown below:

```
EW PLACE

3DA5:0B20  00F3.10  4F20.30  3DC1.
3DA5:0B22  0030.20
>_
```

10.2.1 Enter Command

Syntax

E *address* [[*list*]]

The Enter command enters one or more values into memory at the specified *address*. The data are entered in the format of the default type, which is the last type specified with a Dump, Enter, Watch Memory, or Tracepoint Memory command. If none of these commands has been entered during the session, the default type is bytes.

Use this command with caution when entering values in the list format; values will be truncated if you enter a word-sized value when the default type is actually bytes. If you are not sure of the current default type, specify the size in the command.

NOTE *The Execute command and the Enter command have the same command letter (**E**). The difference is that the Execute command never takes an argument; the Enter command always requires at least one argument.*

10.2.2 Enter Bytes Command

Syntax

EB *address* [[*list*]]

The Enter Bytes command enters one or more byte values into memory at *address*. The optional *list* can be entered as a list of expressions separated by spaces. The expressions are evaluated and entered in the current radix. If *list* is not given, the CodeView debugger prompts for new values that must be entered in hexadecimal.

The Enter Bytes command can also be used to enter strings, as described in Section 10.2.3, "Enter ASCII Command."

Examples

```
EB 256 10 20 30
>
```

If the current radix is 10, the above example replaces the three bytes at DS:256, DS:257, and DS:258 with the decimal values `10`, `20`, and `30`. (These three bytes correspond to the hexadecimal addresses DS:0100, DS:0101, and DS:0102.)

```
EB 256

3DA5:0100  130F.A
>
```

The example above replaces the byte at DS:256 (DS:0100 hexadecimal) with 10 (0A hexadecimal).

10.2.3 Enter ASCII Command

Syntax

EA *address* [[*list*]]

The Enter ASCII command works in the same way as the Enter Bytes command (**EB**) described in Section 10.2.2 above. The *list* version of this command can be used to enter a string expression.

Example

```
EA message "File cannot be found"
>
```

In the example above, the string `File cannot be found` is entered starting at the symbolic address `message`. (Note that the double quotation marks are CodeView string delimiters.)

You can also use the Enter Bytes command to enter a string expression, or you can enter nonstring values using the Enter ASCII command.

10.2.4 Enter Integers Command

Syntax

EI *address* [[*list*]]

The Enter Integers command enters one or more word values into memory at *address* using the signed-integers format. With the CodeView debugger, a signed integer can be any decimal integer between –32,768 and 32,767.

The optional *list* can be entered as a list of expressions separated by spaces. The expressions are entered and evaluated in the current radix. If *list* is not given, the CodeView debugger prompts for new values that must be entered in decimal.

Examples

```
EI 256 -10 10 -20
>
```

If the current radix is 10, the example above replaces the three integers at DS:256, DS:258, and DS:260 with the decimal values `-10`, `10`, and `-20`. (The three addresses correspond to the three hexadecimal addresses DS:0100, DS:0102, and DS:0104.)

```
EI 256

3DA5:0100  130F.-10
>
```

The example above replaces the integer at DS:256 (hexadecimal address DS:0100) with `-10`.

10.2.5 Enter Unsigned Integers Command

Syntax

EU *address* [[*list*]]

The Enter Unsigned Integers command enters one or more word values into memory at *address* using the unsigned-integers format. With the CodeView debugger, an unsigned integer can be any decimal integer between 0 and 65,535. The optional *list* can be entered as a list of expressions separated by spaces. The expressions are entered and evaluated in the current radix. If *list* is not given, the CodeView debugger prompts for new values that must be entered in decimal.

Examples

```
EU 256 10 20 30
>
```

If the current radix is 10, the example above replaces the three unsigned integers at DS:256, DS:258, and DS:260 with the decimal values `10`, `20`, and `30`. (These addresses correspond to the hexadecimal addresses DS:0100, DS:0102, and DS:0104.)

```
EU 256

3DA5:0100  130F.10
>
```

The example above replaces the integer at DS:256 (DS:0100 hexadecimal) with `10.`

10.2.6 Enter Words Command

Syntax

EW *address* [[*list*]]

The Enter Words command enters one or more word values into memory at *address*.

The optional *list* can be entered as a list of expressions separated by spaces. The expressions are entered and evaluated in the current radix. If *list* is not given, CodeView prompts for new values that must be entered in hexadecimal.

Examples

```
EW 256 10 20 30
>
```

If the current radix is 10, the example above replaces the three words at DS:256, DS:258, and DS:260 with the decimal values `10`, `20`, and `30`. (These addresses correspond to the hexadecimal addresses DS:0100, DS:0102, and DS:0104.)

```
EW 256

3DA5:0100  130F.A
>
```

The example above replaces the integer at DS:256 (DS:0100 hexadecimal) with 10 (0A hexadecimal).

10.2.7 Enter Double Words Command

Syntax

ED *address* [[*list*]]

The Enter Double Words command enters one or more double-word values into memory at *address*. Double words are displayed and entered in the address

format *segment*:*offset*—that is, two words separated by a colon (:). If the colon is omitted and only one word entered, only the offset portion of the address will be changed.

The optional *list* can be entered as a list of expressions separated by spaces. The expressions are entered and evaluated in the current radix. If *list* is not given, CodeView prompts for new values that must be entered in hexadecimal.

Examples

```
ED 256 8700:12008
>
```

If the current radix is 10, the example above replaces the double words at DS:256 (DS:0100 hexadecimal) with the decimal address `8700:12008` (hexadecimal address 21FC:2EE8).

```
ED 256

3DA5:0100  21FC:2EE8.2EE9
>
```

The example above replaces the offset portion of the double word at DS:256 (DS:0100 hexadecimal) with `2EE9` hexadecimal. Since the segment portion of the address is not provided, the existing segment (`21FC` hexadecimal) is unchanged.

10.2.8 Enter Short Reals Command

Syntax

ES *address* [[*list*]]

The Enter Short Reals command enters one or more short-real values into memory at *address*.

The optional *list* can be entered as a list of real numbers separated by spaces. The numbers must be entered in decimal, regardless of the current radix. If *list* is not given, the CodeView debugger prompts for new values that must be entered in decimal. Short-real numbers can be entered either in floating-point format or in scientific-notation format.

Examples

```
ES 256 23.479 1/4 -1.65E+4 235
>
```

The example above replaces the four numbers at DS:256, DS:260, DS:264, and DS:268 with the real numbers `23.479`, `0.25`, `-1650.0`, and `235.0`. (These addresses correspond to the hexadecimal addresses DS:0100, DS:0104, DS:0108, and DS:0112.)

```
ES PI
3DA5:0064  42 79 74 65   7.215589E+022  3.141593
>
```

The example above replaces the number at the symbolic address `PI` with `3.141593`.

10.2.9 Enter Long Reals Command

Syntax

EL *address* [[*list*]]

The Enter Long Reals command enters one or more long-real values into memory at *address*.

The optional *list* can be entered as a list of real numbers separated by spaces. The numbers must be entered in decimal, regardless of the current radix. If *list* is not given, the CodeView debugger prompts for new values that must be entered in decimal. Long-real numbers can be entered either in floating-point format or in scientific-notation format.

Examples

```
EL 256 23.479 1/4 -1.65E+4 235
>
```

The example above replaces the four numbers at DS:256, DS:264, DS:272, and DS:280 with the real numbers `23.479`, `0.25`, `-1650.0`, and `235.0`. (These addresses correspond to the hexadecimal addresses DS:0100, DS:0108, DS:0110, and DS:0118.)

```
EL PI
3DA5:0064  42 79 74 65 DC OF 49 40   5.012391E+001  3.141593
>
```

The example above replaces the number at the symbolic address `PI` with `3.141593`.

10.2.10 Enter 10-Byte Reals Command

Syntax

ET *address* [[*list*]]

The Enter 10-Byte Reals command enters one or more 10-byte-real values into memory at *address*.

The optional *list* can be entered as a list of real numbers separated by spaces. The numbers must be entered in decimal, regardless of the current radix. If *list* is not given, the CodeView debugger prompts for new values that must be entered in decimal. The numbers can be entered either in floating-point format or in scientific-notation format.

Examples

```
ET 256 23.479 1/4 -1.65E+4 235
>
```

The example above replaces the four numbers at DS:256, DS:266, DS:276, and DS:286 with the real numbers `23.479`, `0.25`, `-1650.0`, and `235.0`. (These addresses correspond to the hexadecimal addresses DS:0100, DS:010A, DS:0114, and DS:011E.)

```
>ET PI
3DA5:0064  42 79 74 65 DC 0F 49 40 7F BD  -3.292601E-193
3.141593
>
```

The example above replaces the number at the symbolic address `PI` with `3.141593`.

10.3 Fill Memory Command

The Fill Memory command provides an efficient way of filling up a large or small block of memory with any values you specify. It is primarily of interest to assembly programmers because the command enters values directly into memory. However, you may find it useful for initializing large data areas such as an array or structure.

You can enter arguments to the Fill Memory command using any radix.

Mouse

The Fill Memory command cannot be executed with a mouse.

Keyboard

The Fill Memory command cannot be executed with a keyboard command.

Dialog

To fill an area of memory with values you specify, enter the Fill Memory command as follow:

F *range list*

The Fill Memory command fills the addresses in the specified *range* with the byte values specified in *list*. The values in the list are repeated until the whole range is filled. (Thus, if you specify only one value, the entire range is filled with that same value.) If the *list* has more values than the number of bytes in the *range*, the command ignores any extra values.

Examples

```
F 100 L 100 0   ;* hexadecimal radix assumed
>
```

The first example fills 255 (100 hexadecimal) bytes of memory starting at DS:0100 with the value 0. This command could be used to reinitialize the program's data without having to restart the program.

```
F table L 64 42 79 74 ;* hexadecimal radix assumed
>
```

The second example fills the 100 (64 hexadecimal) bytes starting at `table` with the following hexadecimal byte values: 42, 79, 74. These three values are repeated until all 100 bytes are filled.

10.4 Move Memory Command

The Move Memory command enables you to copy all the values in one block of memory directly to another block of memory of the same size. This command is of most interest to assembly programmers, but it can be used by anyone who wants to do large data transfers. For example, you can use this command to copy all the values in one array to the elements of another.

Mouse

The Move Memory command cannot be executed with the mouse.

Keyboard

The Move Memory command cannot be executed with a keyboard command.

Dialog

To copy the values in one block of memory to another, enter the Move Memory command with the following syntax:

M *range address*

The values in the block of memory specified by *range* are copied to a block of the same size beginning at *address*. All data in *range* are guaranteed to be copied completely over to the destination block, even if the two blocks overlap. However, if they do overlap, some of the original data in *range* will be altered.

To prevent loss of data, the Move Memory command copies data starting at the source block's lowest address whenever the source is at a higher address than the destination. If the source is at a lower address, the Move Memory command copies data beginning at the source block's highest address.

Example

```
M arr1(1) L arsize arr2(1)  ;* FORTRAN example
>
```

In the example above, the block of memory beginning with the first element of `arr1` and `arsize` bytes long is copied directly to a block of the same size beginning at the address of the first element of `arr2` . In C, this command would be entered as `M arr1[0] L arsize arr2[0]` .

10.5 Port Output Command

The Port Output command sends specific byte values to hardware ports. It is primarily of use to assembly programmers writing code that interacts directly with hardware.

Mouse

The Port Output command cannot be executed with a mouse.

Keyboard

The Port Output command cannot be executed with a keyboard command.

Dialog

To output to a hardware port, enter the Port Output command with the following syntax:

O *port byte*

The specified *byte* is sent to the specified *port* in which *port* is a 16-bit port address.

Example

```
O 2F8 4F        ;* hexadecimal system radix assumed
>
```

The byte value `4F` hexadecimal is sent to output port `2F8`.

The example above assumes that the system radix is hexadecimal. However (as with all other CodeView commands), you can enter the Port Output command using any radix you prefer. Both the *port* and *byte* arguments assume system radix unless you specify a radix override.

The Port Output command is often used in conjunction with the Port Input command discussed in Section 6.7.

10.6 Register Command

The Register command has two functions: it displays the contents of the central processing unit registers and it changes the values of those registers. The modification features of the command are explained in this section. The display features of the Register command are explained in Section 6.8.

Mouse

The only register that can be changed with the mouse is the flags register. The register's individual bits (called flags) can be set or cleared. To change a flag, first make sure the register window is open. The window can be opened by selecting Registers from the Options menu or by pressing F2.

The flag values are shown as mnemonics in the bottom of the window. Point to the flag you want to change and click either button. The mnemonic word representing the flag value will change. The mnemonics for each flag are shown in the second and third columns of Table 10.1 below. The color or highlighting of the flag will also be reversed when you change a flag. Set flags are shown in red on color monitors and in high-intensity text on two-color monitors. Cleared flags are shown in light blue on color monitors or normal text on two-color monitors.

Keyboard

The registers cannot be changed with keyboard commands.

Dialog

To change the value of a register with a dialog command, enter a command line with the following syntax:

R [[*registername*[[[=]]*expression*]]]

To modify the value in a register, type the command letter `R` followed by *registername*. The CodeView debugger displays the current value of the register

and prompts for a new value. Press ENTER if you only want to examine the value. If you want to change it, type an expression for the new value and press ENTER.

As an alternative, you can type both *registername* and *expression* in the same command. You can use the equal sign (=) between *registername* and *expression*, but a space has the same effect.

The register name can be any of the following: AX, BX, CX, DX, CS, DS, SS, ES, SP, BP, SI, DI, IP, or F (for flags). If you have a 386-based machine and have turned the 386 option on, the register name can be one of the 32-bit register names shown in table 4.9.

To change a flag value, supply the register name F when you enter the Register command. The command displays the value of each flag as a two-letter name.

At the end of the list of values, the command displays a dash (–). Enter new values after the dash for the flags you wish to change, then press ENTER. You can enter flag values in any order. Flags for which new values are not entered remain unchanged. If you do not want to change any flags, press ENTER.

If you enter an illegal flag name, an error message is displayed. The flags preceding the error are changed; flags at and following the error are not changed.

The flag values are shown in Table 10.1.

Table 10.1 Flag-Value Mnemonics

Flag Name	Set	Clear
Overflow	OV	NV
Direction	DN	UP
Interrupt	EI	DI
Sign	NG	PL
Zero	ZR	NZ
Auxiliary carry	AC	NA
Parity	PE	PO
Carry	CY	NC

Examples

```
R IP 256
>
```

The example above changes the `IP` register to the value `256` (0100 hexadecimal).

```
R AX
AX OEOO
:_
```

The example above displays the current value of the `AX` register and prompts for a new value (the underscore represents the CodeView cursor). You can now type any 16-bit value after the colon.

```
R AX
AX 0E00
:256
>_
```

The example above changes the value of `AX` to 256 (in the current radix).

```
R F UP EI PL
```

The example above shows the command-line method of changing flag values.

```
R F
NV(OV) UP(DN) EI(DI) PL(NG) NZ(ZR) AC(NA) PE(PO) NC(CY) -OV
DI ZR
>R F
OV(NV)  UP(DN) DI(EI) PL(NG) ZR(NZ) AC(NA) PE(PO) NC(CY) -
>
```

With the prompting method of changing flag values (shown above), the first mnemonic for each flag is the current value, and the second mnemonic (in parentheses) is the alternate value. You can enter one or more mnemonics at the dash prompt. In the example, the command is given a second time to show the results of the first command.

CHAPTER 11

CodeView Control Commands

This chapter discusses commands that control the operation of the CodeView debugger. The commands in this category are listed below:

Command	Action
Help (**H**)	Displays help
Quit (**Q**)	Returns to DOS
Radix (**N**)	Changes radix
Redraw (**@**)	Redraws screen
Screen Exchange (****)	Switches to output screen
Search (**/**)	Searches for regular expression
Shell Escape (**!**)	Starts new DOS shell
Tab Set (**#**)	Sets tab size
Option (**O**)	Views or sets CodeView options
Redirection and related commands	Control redirection of CodeView output or input

11.1 Help Command

CodeView has two help systems: a complete on-line Help system available only in window mode and a syntax summary available with sequential mode.

Mouse

To enter the complete on-line Help system with the mouse, point to View on the menu bar, press a mouse button and drag the highlight down to a Help selection, then release the button. The appropriate help screen will appear.

Keyboard

If you are in window mode, press F1 to enter the complete on-line Help system. If you are in sequential mode, a syntax-summary screen appears when you press F1.

Dialog

If you are in window mode, you can view the complete on-line Help system with the following command:

H

If you are in sequential mode, this command displays a screen containing all CodeView dialog commands with the syntax for each. This screen is the only help available in sequential mode.

11.2 Quit Command

The Quit command terminates CodeView and returns control to DOS.

Mouse

To quit the CodeView debugger with the mouse, point to File on the menu, press a mouse button and drag the highlight down to the Exit selection, then release the button. The CodeView screen is replaced by the DOS screen with the cursor at the DOS prompt.

Keyboard

To quit the CodeView debugger with a keyboard command, press ALT+F to open the File menu, then press X to select Exit. The CodeView screen is replaced by the DOS screen with the cursor at the DOS prompt.

Dialog

To quit the CodeView debugger with a dialog command, enter a command line with the following syntax:

Q

When the command is entered, the CodeView screen is replaced by the DOS screen with the cursor at the DOS prompt.

11.3 Radix Command

The Radix command changes the current radix for entering arguments and displaying the value of expressions. The default radix when you start the CodeView debugger is 10 (decimal). Radixes 8 (octal) and 16 (hexadecimal) can also be set. Binary and other radixes are not allowed.

The following seven conditions are exceptions; they are not affected by the Radix command:

1. The radix for entering a new radix is always decimal.
2. Format specifiers given with the Display Expression command or any of the Watch Statement commands override the current radix.
3. Addresses output by the Assemble, Dump, Enter, Examine Symbol, and Unassemble commands are always shown in hexadecimal.
4. In assembly mode, all values are shown in hexadecimal.
5. The display radix for Dump, Watch Memory, and Tracepoint Memory commands is always hexadecimal if the size is bytes, words, or double words and always decimal if the size is integers, unsigned integers, short reals, long reals, or 10-byte reals.
6. The input radix for the Enter commands with the prompting method is always hexadecimal if the size is bytes, words, or double words and always decimal if the size is integers, unsigned integers, short reals, long reals, or 10-byte reals. The current radix is used for all values given as part of a list, except real numbers, which must be entered in decimal.
7. The register display is always in hexadecimal.

Mouse

You cannot change the input radix with the mouse.

Keyboard

You cannot change the input radix with a keyboard command.

Dialog

To change the input radix with a dialog command, enter a command line with the following syntax:

N[[*radixnumber*]]

The *radixnumber* can be 8 (octal), 10 (decimal), or 16 (hexadecimal). The default radix when you start the CodeView debugger is 10 (decimal), unless your

main program is written with the Microsoft Macro Assembler, in which case the default radix is 16 (hexadecimal). If you give the Radix command with no argument, the debugger displays the current radix.

Examples

```
N10
>N
10
>? prime
107
>

>N8      ;* C example
>? prime
0153
>

>N16     ;* FORTRAN example
>? prime
#006b
>

>N8      ;* BASIC example
>? prime
&O153
>
```

The example above shows how `107` decimal, stored in the variable `prime`, would be displayed with different radixes. Examples are taken from different languages; there is no logical connection between the radix and the language used in each example.

```
N8
>? 34,i
28
>N10
>? 28,i
28
>N16
>? 1C,i
28
>
```

In the example above, the same number is entered in different radixes, but the `i` format specifier is used to display the result as a decimal integer in all three cases. See Chapter 6, "Examining Data and Expressions," for more information on format specifiers.

11.4 Redraw Command

The Redraw command can be used only in window mode; it redraws the CodeView screen. This command is seldom necessary, but you might need it if the output of the program being debugged disturbs the CodeView display temporarily.

Mouse

You cannot redraw the screen using the mouse.

Keyboard

You cannot redraw the screen using a keyboard command.

Dialog

To redraw the screen with a dialog command, enter a command line with the following syntax:

@

11.5 Screen Exchange Command

The Screen Exchange command allows you to switch temporarily from the debugging screen to the output screen.

The CodeView debugger will use either screen flipping or screen swapping to store the output and debugging screens. See Chapter 1, "Getting Started," for an explanation of flipping and swapping.

Mouse

To execute the Screen Exchange command with the mouse, open the View menu, then select Output. Press any key when you are ready to return to the debugging screen.

Keyboard

To execute the Screen Exchange command with a keyboard command, press F4. Press any key when you are ready to return to the debugging screen.

Dialog

To execute the Screen Exchange command from the dialog window, enter a command line with the following syntax:

\

The output screen will then appear. Press any key when you are ready to return to the CodeView debugging screen.

11.6 Search Command

The Search command allows you to search for a regular expression in a source file. The expression being sought is specified either in a dialog box or as an argument to a dialog command. Once you have found an expression, you can search for the next or previous occurrence of the expression.

Regular expressions are patterns of characters that may match one or many different strings. The use of patterns to match more than one string is similar to the DOS method of using wild-card characters in file names. Regular expressions are explained in detail in Appendix A.

You can use the Search command without understanding regular expressions. Since text strings are the simplest form of regular expressions, you can enter a string of characters as the expression you want to find. For example, you could enter `COUNT` if you wanted to search for the word "COUNT" in the source file.

The following characters have special meanings in regular expressions: backslash (\), asterisk (*), left bracket ([), period (.), dollar sign ($), and caret (^). To find strings containing these characters, you must precede the characters with a backslash; this cancels their special meanings.

For example, you would use `\*` to find `x*y`. The periods in the relational operators must also be preceded by a backslash.

The Case Sense selection from the Options menu has no effect on searches for regular expressions.

NOTE *When you search for the next occurrence of a regular expression, the CodeView debugger searches to the end of the file, then wraps around and begins again at the start of the file. This can have unexpected results if the expression occurs only once. When you give the command repeatedly, nothing seems to happen. Actually, the debugger is repeatedly wrapping around and finding the same expression each time.*

Mouse

To find a regular expression with the mouse, point to Search on the menu bar, press a mouse button and drag the highlight down to the Find selection, then release the button. A dialog box appears, asking for the regular expression to be found. Type the expression and press either the ENTER key or a mouse button. The CodeView debugger starts searching at the current cursor position and puts the cursor at the next line containing the regular expression. An error message appears if the expression is not found. If you are in assembly mode, the debugger automatically switches to source mode when the expression is found.

After you have found a regular expression, you can search for the next or previous occurrence of the expression. Point to Search on the menu bar, press a mouse button and drag the highlight down to the Next or Previous selection, then release the button. The cursor moves to the next or previous match of the expression.

You can also search the executable code for a label (such as a routine name or an assembly-language label). Point to Search on the menu bar, press a mouse button and drag the highlight down to the Label selection, then release the button. A dialog box appears, asking for the label to be found. Type the label name, and press either ENTER or a mouse button. The cursor will move to the line containing the label. This selection differs from other search selections because it searches executable code rather than source code. The CodeView debugger switches to assembly mode, if necessary, to display a label in a library routine or assembly-language module.

Keyboard

To find a regular expression with a keyboard command, press ALT+S to open the Search menu, and then press F to select Find. A dialog box appears, asking for the regular expression to be found. Type the expression and press ENTER. The CodeView debugger starts searching at the current cursor position and puts the cursor at the next line containing the regular expression. An error message appears if the expression is not found. If you are in assembly mode, the debugger automatically switches to source mode when the expression is found.

After you have found a regular expression, you can search for the next or previous occurrence of the expression. Press ALT+S to open the Search menu and then press N to select Next or P to select Previous. The cursor will move to the next or previous match of the expression.

You can also search the executable code for a label (such as a routine name or an assembly-language label). Press ALT+S to open the Search menu and then press L to select Label. A dialog box appears, asking for the label to be found. Type the label name and press ENTER. The cursor moves to the line containing the label. This selection differs from other search selections because it searches executable code rather than source code. The CodeView debugger switches to assembly mode, if necessary, to display a label in a library routine or assembly-language module.

Dialog

To find a regular expression using a dialog command, enter a command line with the following syntax:

/[[*regularexpression*]]

If *regularexpression* is given, the CodeView debugger searches the source file for the first line containing the expression. If no argument is given, the debugger searches for the next occurrence of the last regular expression specified.

In window mode, the CodeView debugger starts searching at the current cursor position and puts the cursor at the next line containing the regular expression. In sequential mode, the debugger starts searching at the last source line displayed. It displays the source line in which the expression is found. An error message appears if the expression is not found. If you are in assembly mode, the CodeView debugger automatically switches to source mode when the expression is found.

You cannot search for a label with the dialog version of the Search command, but you can use the View command with the label as an argument for the same effect.

11.7 Shell Escape Command

The Shell Escape command allows you to exit from the CodeView debugger to a DOS shell. You can execute DOS commands or programs from within the debugger, or you can exit from the debugger to DOS while retaining your current debugging context.

The Shell Escape command works by saving the current processes in memory and then executing a second copy of COMMAND.COM. The COMSPEC environment variable is used to locate a copy of COMMAND.COM.

Opening a shell requires a significant amount of free memory (usually more than 200K) because the CodeView debugger, the symbol table, COMMAND.COM, and the program being debugged must all be saved in memory. If you do not have enough memory, an error message appears. Even if you have enough memory to start a new shell, you may not have enough memory left to execute large programs from the shell.

If you change directories while working in the shell, make sure you return to the original directory before returning to the CodeView debugger. If you don't, the debugger may not be able to find and load source files when it needs them.

NOTE *In order to use the Shell Escape command, the executable file being debugged must release unneeded memory. Programs created with Microsoft compilers release memory during start-up.*

You cannot use the Shell Escape command with assembler programs unless the program specifically releases memory by using the DOS function call 4A hexadecimal (Set Block) or is linked with the /CPARMAXALLOC link option.

Mouse

To open a DOS shell with the mouse, point to File on the menu bar, press a mouse button and drag the highlight down to the DOS Shell selection, then release the button. If there is enough memory to open the shell, the DOS screen

appears. You can execute any DOS command or any program. When you are ready to return to the debugging session, type the command `exit` (in any combination of uppercase and lowercase letters). The debugging screen will appear with the same status it had when you left it.

Keyboard

To open a DOS shell with a keyboard command, press ALT+F to open the File menu, then press D to select DOS Shell. If there is enough memory to open the shell, the DOS screen appears. You can execute any DOS internal command or any program. When you are ready to return to the debugging session, type the command `exit` (in any combination of uppercase and lowercase letters). The debugging screen will appear with the same status it had when you left it.

Dialog

To open a DOS shell using a dialog command, enter a command line with the following syntax:

![[*command*]]

If you want to exit to DOS and execute several programs or commands, enter the command with no arguments. The CodeView debugger executes a new copy of COMMAND.COM, and the DOS screen appears. You can run programs or DOS internal commands. When you are ready to return to the debugger, type the command `exit` (in any combination of uppercase and lowercase letters). The debugging screen appears with the same status it had when you left it.

If you want to execute a program or DOS internal command from within CodeView, enter the Shell Escape command (**!**) followed by the name of the command or the program you want to execute. The output screen appears, and the debugger executes the command or program. When the output from the command or program is completed, the message `Press any key to continue...` appears at the bottom of the screen. Press a key to make the debugging screen reappear with the same status it had when you left it.

Examples

```
!
```

In the above example, the CodeView debugger saves the current debugging context and executes a copy of COMMAND.COM. The DOS screen appears, and you can enter any number of commands. To return to the debugger, enter `exit`.

```
!DIR a:*.for
```

In the example above, the DOS command `DIR` is executed with the argument `a:*.for`. The directory listing will be followed by a prompt telling you to press any key to return to the CodeView debugging screen.

```
!CHKDSK a:
```

In the example above, the DOS command `CHKDSK` is executed, and the status of the disk in Drive A is displayed in the dialog window. The program name specified could be for any executable file, not just that for a DOS program.

11.8 Tab Set Command

The Tab Set command sets the width in spaces that the CodeView debugger fills for each tab character. The default tab is eight spaces. You might want to set a smaller tab size if your source code has so many levels of indentation that source lines extend beyond the edge of the screen. This command has no effect if your source code was written with an editor that indents with spaces rather than tab characters.

Mouse

You cannot set the tab size by using the mouse.

Keyboard

You cannot set the tab size by using a keyboard command.

Dialog

To set the tab size with a dialog command, enter a command line with the following syntax:

#*number*

The *number* is the new number of characters for each tab character. In window mode, the screen is redrawn with the new tab width when you enter the command. In sequential mode, any output of source lines reflects the new tab size.

Example

```
.
32:                 IF (X(I)) .LE. X(J)) GOTO 301
>#4
>.
32:         IF (X(I)) .LE. X(J)) GOTO 301
>
```

In the example above, the Source Line command (.) is used to show the source line with the default tab width of eight spaces. Next, the Tab Set command is used to set the tab width to four spaces. The Source Line command then shows the same line.

11.9 Option Command

The Option command allows you to view the state of options in the Option menu (Flip/Swap, Bytes Coded, Case Sense, and 386) and to turn any of the these options on or off.

For each different kind of source module that you debug, there is a different set of default settings. However, the use of the Option command overrides any of these settings.

Mouse

To view the state of the options with a mouse, simply point to Options on the menu bar and click either button. Each option is displayed. Those options that are turned on have a double arrow immediately to the left. Options that are turned off have no double arrow.

To change one of the Option settings, drag the highlight down to the option you wish to change and release the button. This reverses the state of the option. (An option that was on will be turned off and vice versa.)

Keyboard

To view the state of the Options menu with a keyboard command, press ALT+O to open the Options menu. Each option is displayed. Those options that are turned on have a double arrow immediately to the left. Options that are turned off have no double arrow.

To change one of the Option settings, press the letter key corresponding to the option's mnemonic. This reverses the state of the option. (An option that was on will be turned off and vice versa.) You can also reverse an option by moving the highlight down with the arrow key and pressing ENTER.

Dialog

To view or change options with a dialog command, enter a command line with the following syntax:

O[[*option* [[+ | -]]]]

In the above display, *option* is one of the following characters: **F**, **B**, **C**, or **3**. If used, there must be no spaces between the character and the **O**. These characters correspond to the options as shown below:

Command	Correspondence
OF	Flip/Swap option
OB	Bytes-Coded option

OC	Case-Sense option
O3	386 option
O	All options

The **O** form of the command (all options) takes no arguments; it displays the state of all four options. The other forms of the command (**OF**, **OB**, **OC**, and **O3**) can be used either with no arguments (in which case they display the state of the option) or they can take the argument **+** or **-**.

The **+** argument turns the option on. The **-** argument turns the option off.

Examples

```
O
Flip/Swap on
Bytes Coded on
Case Sense off
386 off
>O3
386 off
>O3+
386 on
>OF
Flip/Swap on
>OF-
Flip/Swap off
```

In the example above, the **O**, **O3**, and **OF** commands are used to view the current state of an option. Each of the **O3+** and **OF-** commands modifies an option and then reports the results of the modification.

The dialog version of the Option command is particularly useful for redirected CodeView commands (which cannot access menus) and for making the debugger start up with certain options. For example, the following DOS-level command line brings up CodeView with the 386 option on and Bytes Coded off:

```
CV /c"O3+;OB-" test
```

This command line could then be placed into a batch file for convenient execution.

11.10 Redirection Commands

The CodeView debugger provides several options for redirecting commands from or to devices or files. Furthermore, the debugger provides several other commands, which are relevant only when used with redirected files.

Mouse

None of the redirection or related commands can be executed with the mouse.

Keyboard

None of the redirection or related commands can be executed with keyboard commands.

Dialog

The redirection commands are entered with dialog commands, as shown in Sections 11.10.1–11.10.4.3 below.

11.10.1 Redirecting CodeView Input

Syntax

< *devicename*

The Redirected Input command causes the CodeView debugger to read all subsequent command input from a device, such as another terminal or a file. The sample session supplied with most versions of the debugger is an example of commands being redirected from a file.

Examples

```
<COM1
```

The example above redirects commands from the device (probably a remote terminal) designated as `COM1` to the CodeView terminal.

```
<INFILE.TXT
```

The example above redirects command input from file `INFILE.TXT` to the CodeView debugger. You might use this command to prepare a CodeView session for someone else to run. You create a text file containing a series of commands separated by carriage-return and line-feed combinations or semicolons. When you redirect the file, the debugger executes the commands to the end of the file. One way to create such a file is to redirect commands from the CodeView debugger to a file (see Section 11.10.3 below) and then edit the file to eliminate the output and add comments.

11.10.2 Redirecting CodeView Output

Syntax

[[**T**]>[[>]] *devicename*

The Redirected Output command causes the CodeView debugger to write all subsequent command output to a device, such as another terminal, a printer, or a file. The term "output" includes not only the output from commands but also the command characters that are echoed as you type them.

The optional **T** indicates that the output should be echoed to the CodeView screen. Normally, you will want to use the **T** if you are redirecting output to a file so you can see what you are typing. However, if you are redirecting output to another terminal, you may not want to see the output on the CodeView terminal.

The second greater-than symbol (optional) appends the output to an existing file. If you redirect output to an existing file without this symbol, the existing file is replaced.

Examples

```
>COM1
```

In the example above, output is redirected to the device designated as `COM1` (probably a remote terminal). You might want to enter this command, for example, when you are debugging a graphics program and want CodeView commands to be displayed on a remote terminal while the program display appears on the originating terminal.

```
T>OUTFILE.TXT
.
.
.
>>CON
.
.
.
```

In the example above, output is redirected to the file `OUTFILE.TXT`. This command is helpful in keeping a permanent record of a CodeView session. Note that the optional **T** is used so that the session is echoed to the CodeView screen as well as to the file. After redirecting some commands to a file, output is returned to the console (terminal) with the command `>CON`.

```
T>>OUTFILE.TXT
```

If, later in the session, you want to redirect more commands to the same file, use the double greater-than symbol, as in the example above, to append the output to the existing file.

11.10.3 Redirecting CodeView Input and Output

Syntax

= *devicename*

The Redirected Input and Output command causes the CodeView debugger to write all subsequent command output to a device and simultaneously to receive input from the same device. This command is practical only if the device is a remote terminal.

Redirecting input and output works best if you start in sequential mode (using the /T option). The CodeView debugger's window interface has little purpose in this situation since the remote terminal can act only as a sequential (nonwindow) device.

Example

```
=COM1
```

In the example above, output and input are redirected to the device designated as `COM1` . This command would be useful if you wanted to enter debugging commands and see the debugger output on a remote terminal while entering program commands and viewing program output on the terminal where the debugger is running.

11.10.4 Commands Used with Redirection

The following commands are intended for use when redirecting commands to or from a file. Although they are always available, these commands have little practical use during a normal debugging session.

Command	Action
Comment (*)	Displays comment
Delay (:)	Delays execution of commands from a redirected file
Pause (")	Interrupts execution of commands from a redirected file until a key is pressed

11.10.4.1 Comment Command

Syntax

**\comment*

The Comment command is an asterisk (*) followed by text. The CodeView debugger echoes the text of the comment to the screen (or other output device). This command is useful in combination with the redirection commands when you are saving a commented session or when writing a commented session that will be redirected to the debugger.

Examples

```
T>OUTPUT.TXT
>* Dump first 20 bytes of screen buffer
>D #B800:0 L 20
B800:0000 54 17 6F 17 20 17 72 17 65 17 74 17 75 17 72 17
T.o. .r.e.t.u.r.
B800:0010 6E 17 20 17
n. .
>
```

In the example above, the user is sending a copy of a CodeView session to file `OUTPUT.TXT`. Comments are added to explain the purpose of the command. The text file will contain commands, comments, and command output.

```
* Dump first 20 bytes of screen buffer
D #B800:0 L 20
.
.
.
< CON
```

The example above illustrates another way to use the Comment command. You can put comments into a text file of commands that are executed automatically when you redirect the file into the CodeView debugger. In this example, an editing program was used to create the text file called `INPUT.TXT`.

```
<INPUT.TXT
>* Dump first 20 bytes of screen buffer
>D #B800:0 L 20
B800:0000 54 17 6F 17 20 17 72 17 65 17 74 17 75 17 72 17
T.o. .r.e.t.u.r.
B800:0010 6E 17 20 17
n. .
.
.
.
>< CON
>
```

When you read the file into the debugger by using the Redirected Input command, you see the comment, the command, and the output from the command, as in the example above.

11.10.4.2 Delay Command

Syntax

:

The Delay command interrupts execution of commands from a redirected file and waits about half a second before continuing. You can put multiple Delay commands on a single line to increase the length of delay. The delay is the same length regardless of the processing speed of the computer.

Example

```
: ;* That was a short delay...
::::: ;* That was a longer delay...
```

In the example above, the Delay command is used to slow execution of the redirected file into the CodeView debugger.

11.10.4.3 Pause Command

Syntax

"

The Pause command interrupts execution of commands from a redirected file and waits for the user to press a key. Execution of the redirected commands begins as soon as a key is pressed.

Example

```
* Press any key to continue
"
```

In the example above from a text file that might be redirected into the CodeView debugger, a Comment command is used to prompt the user to press a key. The Pause command is then used to halt execution until the user responds.

```
* Press any key to continue
>"
```

The example above shows the output when the text is redirected into the debugger. The next CodeView prompt does not appear until the user presses a key.

CHAPTER 12

Debugging in Protected Mode

The protected-mode CodeView debugger (CVP.EXE) supports all the special programming features of OS/2 protected mode: dynamic-link libraries, multiple processes, and multiple threads within a process. CodeView lets you stop execution, then switch between individual processes and threads.

The support for thread debugging is especially powerful because it lets you block (or "freeze") selected threads while letting others run. You can set breakpoints applicable to all threads or only to a specific thread.

Note that you must use the protected-mode CodeView debugger (CVP.EXE) in order to run CodeView in protected mode or debug protected-mode programs. Furthermore, before you run CVP you must set IOPL=YES in your CONFIG.SYS file.

This chapter deals with the following topics:

- Using CodeView in real and protected mode
- Debugging dynamic-link libraries
- Debugging multiple processes
- Debugging multiple threads

All the techniques presented in this chapter can be used together. You can debug multiple processes, and within each process debug multiple threads.

12.1 Using CodeView in Different Modes

Chapters 1–11 assume that the real-mode CodeView debugger (CV.EXE) is running in real mode, debugging a real-mode program. This section summarizes the major considerations and limitations of other situations.

As noted above, to debug a protected-mode program, you must run CVP.EXE in protected mode. You must first set IOPL=YES. This setting enables applications to run at Ring 3, giving programs low-level access. CodeView needs to run at this level because it does direct-hardware access.

WARNING *If you do not set IOPL=YES before running CVP, CVP fails to run and the system does not give you a clear message explaining why CodeView failed. The system may become unstable and fail at any time.*

You must also use CVP in order to debug a bound program. The restrictions mentioned above apply. Real-mode CodeView cannot debug a bound program.

The real-mode debugger can run in the 3.x compatibility box. However, when you run CodeView in the compatibility box, include /S on the command line. Otherwise, the mouse pointer does not appear.

12.2 Debugging Dynamic-Link Libraries

The protected-mode CodeView debugger (CVP) can debug dynamic-link modules but only if it is told what libraries to search at run time. For more information on dynamic-link libraries, refer to the *Microsoft Operating System/2 Programmer's Guide*.

When you place a module in a dynamic-link library, neither code nor symbolic information for that module is stored in the executable (.EXE) file; instead, the code and symbols are stored in the library and are not brought together with the main program until run time.

Thus, the protected-mode debugger needs to search the dynamic-link library for symbolic information. Because the debugger does not automatically know what libraries to look for, CVP has an additional command-line option that enables you to specify dynamic-link libraries:

Syntax

/L *file*

The /L option directs the CodeView debugger to search *file* for symbolic information. When you use this option, at least one space must separate /L from *file*.

Example

```
CVP /L DLIB1.DLL /L GRAFLIB.DLL PROG
```

In the example above, CVP is invoked to debug the program PROG.EXE. To find symbolic information needed for debugging each module, CVP searches the libraries `DLIB1.DLL` and `GRAFLIB.DLL`, as well as the executable file `PROG.EXE`.

12.3 Debugging Multiple Processes

To enable debugging of multiple processes, you must first start up CodeView with the /O (offspring) option. The syntax of this option is simple, as it takes no arguments.

Syntax

/O

If you do not use the /O option (or the option cannot be used), CodeView lets your program spawn new processes, but you will not be able to view or trace through these processes. They run in the background as far as CodeView is concerned.

For example, to debug multiple processes of the program SPACEMAN.EXE you would use the following command:

```
CVP /O SPACEMAN
```

The /O option has two limitations:

1. You must have OS/2 Version 1.1 or later to use it.
2. This option is incompatible with the /2 option.

The rest of this section assumes that you have successfully started CodeView with the /O option.

Every time your program executes a line of code that spawns a child process, CodeView responds by displaying the process ID number (Pid) and asking if you wish to debug the child process. The message displayed is similar to the following:

```
Pid 24 started. Do you wish to debug (y/n)?
```

To debug the child process, type Y and then press ENTER. Type any other letter for no. CodeView takes a different course of action depending on your response.

- If you respond yes, CodeView spawns a new CodeView process. This process controls execution of your program's child process. Each instance of CodeView spawned in this way becomes a separate debugging session.

  A new process runs in the same screen group as its parent process (unless you call the DosStartSession system function). Using CodeView does not change this. However, each new instance of CodeView always runs in its own screen group. Since OS/2 supports a maximum of 16 screen groups, the number of child processes that you can debug at one time is limited.

- If you respond no, CodeView lets the program spawn the child process. However, you will not be able to control or trace the child process with CodeView. The child process is active but not accessible to CodeView commands.

You can move between different CodeView processes in the following two ways: by using the OS/2 Session Manager or the Process command (|). The Process command, in turn, has two forms. You can use this command to view status of child processes or to switch directly to the debugging session of the child process.

NOTE *You may need to make note of process ID numbers when CodeView spawns a process. CodeView identifies multiple processes only by their ID numbers.*

12.3.1 Viewing Status

To view the status of the child processes (of the current process), enter the Process command followed by no arguments:

```
|
```

CodeView responds by displaying three fields: process ID number, session (screen group) ID number, and yes or no, depending on whether or not each process has its own instance of CodeView. The following example shows a sample process status for a process with three children:

```
001|
ProcessID    SessionID    Debugging
00024        00006        Yes
00026        00006        Yes
00028        00006        No
```

In the example above, only processes 24 and 26 can be debugged. Each of these processes corresponds to a different instance of CodeView, and each instance runs in a separate screen group. Process 28 is active but cannot be debugged.

12.3.2 Switching to a Child Process

If a child process can be debugged, you can switch to that process directly by using the Process command. Use of this command accomplishes the same end as using the Session Manager but is more direct.

To switch to the debugging session for a child process, enter the Process command with the following syntax:

| *processID*

in which *processID* is the process ID (Pid) of the process you wish to debug.

NOTE *The Process command only works with direct children. In other words, you can spawn a child which in turns spawns another child. The Process command does not give you direct access to the "grandchild." Instead, you must switch to the intermediate parent.*

To return to debugging a parent or grandparent, you must use the OS/2 Session Manager.

12.4 Debugging Multiple Threads

A program running in OS/2 protected mode has one or more threads. Threads are the fundamental units of execution; OS/2 can execute a number of different threads concurrently. A thread is similar to a process, yet it can be created or terminated much faster. Threads begin at a function-definition heading in the same program in which they are invoked.

The existence of multiple threads within a program presents a dilemma for debugging. For example, thread 1 may be executing source line 23 while thread 2 is executing source line 78. Which line of code does the CodeView debugger consider to be the current line?

Conversely, you cannot always tell which thread is executing even if you know what the current source line is. In OS/2 protected mode, you can write a program in which two threads enter the same function.

In Figure 12.1, the function `main` uses the DOSCREATETHREAD system call to begin execution of thread 2. The function `f2` is the entry point of the new thread. Thread 2 begins and terminates inside the function `f2`. Before it terminates, however, thread 2 can enter other functions by means of ordinary function calls.

Thread 1 begins execution in the function `main`, and thread 2 begins execution in the function `f2`. Later, both thread 1 and thread 2 enter the function `f3`. (Note that each thread returns to the proper place because each thread has its own stack.) When you use the debugger to examine the behavior of code within the function `f3`, how can you tell which thread you are tracking?

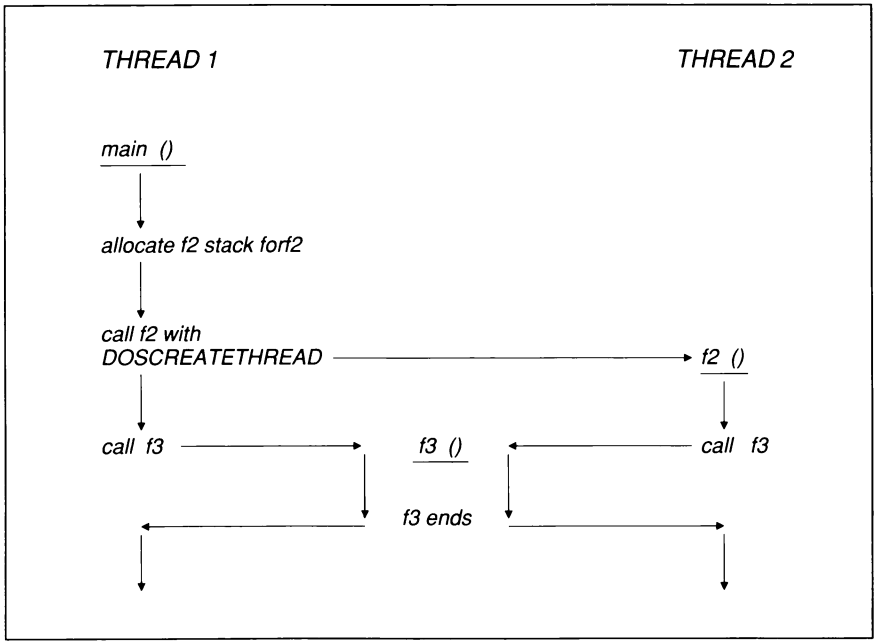


Figure 12.1 Multiple-Thread Program

The protected-mode CodeView debugger solves this dilemma by using a modified CodeView prompt and by providing the Thread command, which is only available with CVP.

The command prompt for the protected-mode CodeView debugger is preceded by a three-digit number indicating the current thread.

Example

```
001>
```

The example above displays the protected-mode CodeView prompt, indicating that thread 1 is the current thread. Thread 1 is always the current thread when you begin a program. If the program never calls the DOSCREATETHREAD function, thread 1 will remain the only thread. Note that certain C library functions (such as BeginThread) call DOSCREATETHREAD for you.

Each thread has its own stack and its own register values. When you change the current thread, you see several changes to the CodeView debugger display:

- The CodeView prompt displays a different three-digit number.
- The register contents change.

- The current source line and current instruction both change to reflect the new value of CS:IP. If you are running the debugger in window mode, you are likely to see different code in the display window.
- The Calls menu and the Stack Trace command displays a different group of functions.

12.5 The Thread Command

This section discusses the Thread command and lists other CodeView commands that may work differently because of multiple threads.

The syntax of the Thread command is displayed below:

Syntax

~[[*specifier*[[*command*]]]]

In the syntax display above, the *specifier* determines to which thread or threads the command applies. You can specify all threads or just a particular thread. The *command* determines which activity the debugger carries out with regard to the specified thread. For example, you can execute the thread, freeze its execution, or select it as the current thread. If you omit *command*, the debugger displays the status of the specified thread. If you omit both *command* and *specifier*, the debugger displays the status of all threads.

The status display for threads consists of the two fields

thread-id thread-state

in which *thread-id* is an integer and *thread-state* has the value `runnable` or `frozen`. All threads not frozen by the debugger are displayed as `runnable`; this includes threads that may be blocked for reasons that have nothing to do with the debugger, such as a critical section.

12.5.1 Legal Values for Specifier

The legal values for *specifier* are listed below along with their effects.

Symbol	Function
(blank)	Displays the status of all threads. If you omit the *specifier* field you cannot enter a *command*. Instead, you enter the tilde (~) by itself.
#	Specifies the last thread that was executed.

This thread is not necessarily the current thread. For example, suppose you are tracing execution of thread 1, and then switch the current thread to thread 2. Until you execute some code in thread 2, the debugger still considers thread 1 to be the last thread executed.

*	Specifies all threads.
n	Specifies the indicated thread. The value of *n* must be a number corresponding to an existing thread. You can determine corresponding numbers for all threads by entering the command **~***, which gives status of all threads.
.	Specifies the current thread.

12.5.2 Legal Values for Command

The legal values for *command* are listed below, along with their effects.

Command	Function
(blank)	The status of the selected thread (or threads) is displayed.
BP	A breakpoint is set for the specified thread or threads. As explained earlier, it is possible to write your program so that the same function is executed by more than one thread. By using this version of the Thread command, you can specify a breakpoint that applies only to a particular thread. The letters **BP** are followed by normal syntax for the Breakpoint Set command, as described in Chapter 7, "Managing Breakpoints." Therefore you can include the optional pass count and command fields.
E	The specified thread is executed in slow motion. When you specify a single thread with **E**, the specified thread becomes the current thread and is executed without any other threads running in the background. The command **~*E** is a special case. It is legal only in source mode and executes the current thread in slow motion, but lets all other threads run (except those that are frozen). You see only the current thread executing in the debugger display.

F The specified thread (or threads) is frozen.

A frozen thread will not run in the background or in response to the debugger Go command. However, if you use the **E**, **G**, **P**, or **T** variation of the Thread command, the specified thread is temporarily unfrozen while the debugger executes the command.

G Control is passed to the specified thread until it terminates or until a breakpoint is reached.

If you give the command **~*G**, all threads execute concurrently (except for those that are frozen). If you specify a particular thread, the debugger temporarily freezes all other threads and executes the specified thread.

P The debugger executes a program step for the specified thread.

If you specify a particular thread, the debugger executes one source line or instruction of the thread. All other threads are temporarily frozen. This version of the Thread command does not change the current thread. Therefore if you specify a thread other than the current thread, you will not see immediate results. However, the subsequent behavior of the current thread may be affected.

The command **~*P** is a special case. It is legal only in source mode, and causes the debugger to step to the next source line while letting all other threads run (except for those that are frozen). You see only the current thread execute in the debugger display.

S The specified thread is selected as the current thread.

This version of the Thread command can apply to only one thread at a time. Thus, the command **~*S** results in an error message. Note that the command **~.S** is legal, but has no effect.

T The specified thread is traced.

This version of the Thread command works in a manner identical to **P**, described above, except that **T** traces through function calls and interrupts, whereas **P** does not.

U The specified thread or threads are unfrozen. This command reverses the effect of a freeze.

With the Thread command, only the **S** (select) and **E** (execute) variations cause the debugger to switch the current thread. However, when a thread causes program execution to stop by hitting a breakpoint, the debugger selects that thread as the current thread.

You can prevent the debugger from changing the current thread by including the breakpoint command **"~.S"**. This command directs the debugger to switch to the current thread rather than the thread that hit the breakpoint. For example, the following command sets a breakpoint at line `120` and prevents the current thread from changing:

```
BP .120 "~.S"
```

12.5.3 Entries to the Thread Command

The Thread command can have several possible entries. They are summarized in the syntax display below.

Syntax

~{#|*|*n*|.}[[BP|E|F|G|P|S|T|U]]

Note that you must include one of the symbols from the first set (which gives possible values for the specifier), but you do not have to include a symbol from the second set (which gives possible values for the command).

Examples

```
004>~
```

The example above displays the status of all threads, including their corresponding numbers.

```
004>~2
```

The example above displays the status of thread 2.

```
004>~5S
```

The example above selects thread 5 as the current thread. Since the current thread was 4 (a fact apparent from the CodeView prompt), the current thread is changing and therefore the registers and the code displayed can be expected to all change.

```
005>~3BP .64
```

The example above sets a breakpoint at source line 64, an action that stops program execution only when thread 3 executes to this line.

```
005>~1F
```

The example above freezes thread 1.

```
005>~*U
```

The example above thaws (unfreezes) all threads; any threads that were frozen before will now be free to execute whenever the Go command is given. If no threads are frozen, this command has no effect.

```
005>~2E
```

The example above selects thread 2 as the current thread, then proceeds to execute thread 2 in slow motion.

```
002>~3S
003>~.F
003>~#S
002>
```

The example above selects thread 3 as the current thread, freezes the current thread (thread 3), and switches back to thread 2. After switching to thread 3, no code was executed; therefore, the debugger considers the last-thread-executed symbol (#) to refer to thread 2.

12.5.4 Effect of Threads on CodeView Commands

Whether or not you use the Thread Command, the existence of threads affects your CodeView debugging session at all times. Particular debugger commands are strongly affected. Each of these commands is discussed below.

Command	Behavior in Multiple-Thread Programs
.	The Current Line command always uses the current value of CS:IP to determine what the current instruction is. Thus, the Current Line command applies to the current thread.
E	When the debugger is in source mode, the Execute command is equivalent to the **~*E** command; the current thread is executed in slow motion while all other threads are also running. When the debugger is in mixed or assembly mode, the Execute command is equivalent to the command **~.P**, which does not let other threads run concurrently.
BP	The Set Breakpoint command is equivalent to the **~*BP** command; the breakpoint applies to all threads.

G	The Go command is equivalent to the **~*G** command; control is passed to the operating system, which executes all threads in the program except for those that are frozen.
P	When the debugger is in source mode, the Program Step command is equivalent to the command **~*P**, which lets other threads run concurrently. When the debugger is in mixed or assembly mode, the Program Step command is equivalent to the command **~.P**, which lets no other threads run.
K	The Stack Trace command displays the stack of the current thread.
T	When the debugger is in source mode, the Trace command is equivalent to the command **~*T**, which lets other threads run concurrently. When the debugger is in mixed or assembly mode, the Trace command is equivalent to the command **~.T**, which lets no other threads run.

In general, CodeView commands apply to all threads unless the nature of the command makes it appropriate to deal with only one thread at a time. In this case, the command applies only to the current thread. (For example, since each thread has its own stack, the Stack Trace command does not apply to all threads.)

PART 2

Utilities

PART 2

Utilities

Part 2 describes the use of each of the DOS and OS/2 programming utilities (exit codes and messages for these utilities are presented in the appendixes).

Some of the material is this part, most notably the information on LINK, is presented in partial form in the user's guides for Microsoft compilers. However, you will find here the only complete, authoritative reference on utility operation and available options.

Chapters 13 –17 document utilities you can use to produce either OS/2 or DOS programs. Chapters 18–21 document utilities and special concepts–such as module-definition files–developed specifically for the creation of OS/2 programs.

CHAPTERS

13 Linking Object Files with LINK 225

14 Incremental Linking with ILINK 261

15 Managing Libraries with LIB 269

16 NMAKE 285

17 Using Other Utilities 307

18 Linking for Windows and OS/2 Systems 315

19 Using Module-Definition Files 321

20 Creating Dual-Mode Programs with BIND 341

21 Using EXEHDR 347

CHAPTER 13

Linking Object Files with LINK

The Microsoft Segmented-Executable Linker (LINK) is used to combine object files into a single executable file. It can be used with object files compiled or assembled for 8086/8088, 80286, or 80386 machines. The format of input to the linker is the Microsoft Relocatable Object-Module Format (OMF), which is based on the Intel 8086 OMF.

The output file from LINK (that is, the executable file) is not bound to specific memory addresses. Thus, the operating system can load and execute this file at any convenient address. LINK can produce executable files containing up to 1 megabyte of code and data.

The following sections explain how to run the linker and specify options that control its operation.

13.1 Determining Linker Output

The linker can produce either an application that runs under real mode (DOS 2.x, 3.x, or 3.x compatibility box), an application that runs under OS/2 protected mode (or Microsoft Windows), or an OS/2 dynamic-link library. If you are developing programs for real mode only, leave the *deffile* field on the LINK command line empty and ignore the following discussion. If you are developing programs for OS/2, read this section to understand what kind of executable file the linker produces. Chapters 18 and 19 explain in detail the terms and concepts mentioned below.

The following rules determine what output the linker produces:

1. If no module-definition file or import library resolves any external references, the linker produces a real-mode application. (In other words, the linker creates a real-mode application unless you specify a module-definition file or import library, and that file or library resolves at least one external reference.)

2. If a module-definition file with a LIBRARY statement is given, the linker produces a dynamic-link library for OS/2 protected mode or Windows.
3. Otherwise, the linker produces an application for OS/2 protected mode or Windows.

You can therefore produce an OS/2 protected-mode application by linking with an import library or a module-definition file, as long as you do not use a **LIBRARY** statement. (The LIBRARY statement is described in Chapter 19, "Using Module-Definition Files.") The file DOSCALLS.LIB is an import library. Therefore, if you link with DOSCALLS.LIB, you produce either an OS/2 application or a dynamic-link library.

When you use a Microsoft high-level language to compile for protected mode, it automatically specifies DOSCALLS.LIB as a default library.

NOTE *Throughout this chapter, all references to OS/2 protected mode also apply to Microsoft Windows. Other chapters may make a distinction between protected mode and Windows. Each reference to "library," unless otherwise noted, refers to a standard (object-code) library, not a dynamic-link library.*

The linker produces files that run in protected mode only or in real mode only. However, OS/2 applications that make dynamic-link calls only to the Family API (a subset of the functions defined in DOSCALLS.LIB) can be made to run under DOS 3.x with the BIND utility. (BIND is discussed in Chapter 20).

13.2 Specifying Files for Linking

Instead of using high-level-language commands to invoke the linker, you can use the LINK command to invoke LINK directly.You can specify the input required for this command in one of three ways:

1. By placing it on the command line.
2. By responding to prompts.
3. By specifying a file containing responses to prompts. This type of file is known as a "response file."

Regardless of the way in which LINK was invoked, press CTRL+C at any time to terminate LINK operation and exit back to DOS.

13.2.1 Specifying File Names

You can use any combination of uppercase and lowercase letters for the file names you specify on the LINK command line or give in response to the LINK command prompts. For example, LINK considers the following three file names to be equivalent:

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

If you specify file names without extensions, LINK uses the following default file-name extensions:

File Type	Default Extension
Object	.OBJ
Executable	.EXE
Map	.MAP
Library	.LIB
Module definition (protected-mode use only)	.DEF

You can override the default extension for a particular command-line field or prompt by specifying a different extension. To enter a file name that has no extension, type the name followed by a period.

Examples

Consider the following two file specifications:

```
ABC.
ABC
```

If you use the first file specification, LINK assumes that the file has no extension. If you use the second file specification, LINK uses the default extension for that prompt.

13.2.2 Linking with the LINK Command Line

Use the following form of the LINK command to specify input on the command line:

LINK *objfiles*[[, [[*exefile*]][[,[[*mapfile*]] [[, [[*libraries*]][[,*deffile*]]]]]]][[;]]

Each of the command-line fields is explained below. In these explanations, reference is made to libraries. Unless qualified by the term "dynamic-link," the word "libraries" refers to import libraries and standard (object-code) libraries, both of which have the default extension .LIB. (Note that dynamic-link libraries have the default extension .DLL, and therefore are usually easy to distinguish from other libraries.) You can specify import libraries anywhere you can specify standard libraries.You can also combine import libraries and standard libraries by using the Library Manager; these combined libraries can then be specified in place of standard libraries.

The objfiles field

The *objfiles* field allows you to specify the names of the object files you are linking. At least one object-file name is required. A space or plus sign (+) must separate each pair of object-file names. LINK automatically supplies the .OBJ extension when you give a filename without an extension. If your object file has a different extension, or if it appears in another directory or on another disk, you must give the full name—including the extension and path name—for the file to be found. If LINK cannot find a given object file, and the drive associated with the object file is a removable (floppy) drive, LINK displays a message and waits for you to change disks.

You may also specify one or more libraries in the *objfiles* field. To enter a library in this field, make sure that you include the .LIB extension; otherwise LINK assumes an .OBJ extension. Libraries entered in this field are called "load libraries" as opposed to "regular libraries." LINK automatically links in every object module in a load library; it does not search for unresolved external references first. The effect of entering a load library is exactly the same as if you had entered all the names of the library's object modules into the *objfiles* field. This feature is useful if you are developing software using many modules and wish to avoid having to retype each module on the LINK command line.

The exefile field

The *exefile* field allows you to specify the name of the executable file. If the file name you give does not have an extension, LINK automatically adds .EXE as the extension. You can give any file name you like; however, if you are specifying an extension, you should always use .EXE because DOS expects executable files to have either this extension or the .COM extension.

The mapfile field

The *mapfile* field allows you to specify the name of the map file if you are creating one. To include public symbols and their addresses in the map file, specify the /MAP option on the LINK command line. (See Section 13.3.15, "Listing Public Symbols.") If you specify a map-filename without an extension, LINK automatically adds an extension of .MAP. LINK creates the map file in the current working directory unless you specify a path name for the map file.

The libraries field

The *libraries* field allows you to specify the name of a library that you want linked to the object file(s). (When LINK finds the name of a library in this field, the library is a "regular library," and LINK will link in only those object modules needed to resolve external references.) Each time you compile a source file for a high-level language, the compiler places the name of one or more libraries in the object file that it creates; the linker automatically searches for a library with this name. Because of this, you do not need to give library names on the LINK command line unless you want to add the names of other libraries, search for libraries in different locations, or override the use of the library named in the object file.

The deffile field

The *deffile* field allows you to specify the file name of a module-definition file. Leave this field blank if you are linking for real mode. The use of a module-definition file is optional for applications, but required for dynamic-link libraries. See Chapters 18 and 19 for more information on module-definition files.

You may specify command-line options after any field—but before the comma that terminates the field. The options are described in sections 13.3.1–13.3.32. You do not have to give any options when you run the linker.

If you include a comma (to indicate where a field would be) but do not put a file name before the comma, LINK will select the default for that field. However, if you use a comma to include the *mapfile* field (but do not include a name), LINK creates a map file. This file has the same base name as the executable file. Use NUL for the map-filename if you do not want to produce a map file.

You can also select default responses by using a semicolon (;). The semicolon tells LINK to use the defaults for all remaining fields. If you do not give all file names on the command line, or if you do not end the command line with a semicolon, the linker prompts you for the files you omitted, using the prompts described in Section 13.2.3, "Linking with the LINK Prompts."

If you do not specify a drive or directory for a file, the linker assumes that the file is on the current drive and directory. If you want the linker to create files in

a location other than the current drive and directory, you must specify the new drive and directory for each such file on the command line.

NOTE *The OS/2 linker supports overlays only when producing a real-mode application.*

Examples

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

The command line above causes LINK to load and link the object modules FUN.OBJ, TEXT.OBJ, TABLE.OBJ, and CARE.OBJ, and to search for unresolved references in the library file XLIB.LIB and the default libraries. By default, the executable file produced by LINK is named FUN.EXE. LINK also produces a map file, FUNLIST.MAP.

```
LINK FUN,,;
```

This command line produces a map file named FUN.MAP since a comma appears as a placeholder for the *mapfile* specification on the command line.

```
LINK FUN,;
LINK FUN;
```

These command lines do not produce a map file, since commas do not appear as placeholders for the *mapfile* specification.

```
LINK MAIN+GETDATA+PRINTIT, , MAIN;
```

The command in the line above causes LINK to link the files MAIN.OBJ, GETDATA.OBJ, and PRINTIT.OBJ into a real-mode executable file since there is no module-definition file. A map file named MAIN.MAP is also produced.

```
LINK GETDATA+PRINTIT, , , MODDEF
```

This command causes LINK to link GETDATA.OBJ and PRINTIT.OBJ into an OS/2 dynamic-link library if MODDEF.DEF contains a **LIBRARY** statement. Otherwise, an OS/2 protected-mode application is produced.

13.2.3 Linking with the LINK Prompts

If you want to use the LINK prompts to specify input to the linker, start the linker by typing LINK at the DOS command level. LINK prompts you for the input it needs by displaying the following lines, one at a time:

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

LINK waits for you to respond to each prompt before printing the next one. Section 13.2.1 gives the rules for specifying file names in response to these prompts.

The responses you give to the LINK command prompts correspond to the fields on the LINK command line. (See Section 13.2.2 for a discussion of the LINK command line.) The following list shows these correspondences:

Prompt	Command-Line Fields
`Object Module`	*objfiles*
`Run File`	*exefile*
`List File`	*mapfile*
`Libraries`	*libraries*
`Definitions File`	*deffile*

If a plus sign (+) is the last character you type on a response line, the prompt appears on the next line, and you can continue typing responses. In this case, the plus sign must appear at the end of a complete file or library name, path name, or drive name.

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, type a semicolon (;) followed immediately by a carriage return. After you enter a semicolon, you cannot respond to any of the remaining prompts for that link session. Use this option to save time when you want to use the default responses. However, you cannot enter a semicolon in response to the `Object Modules` prompt, because there is no default response for that prompt.

The following list shows the defaults for the other linker prompts:

Prompt	Default
`Run File`	The name of the first object file submitted for the `Object modules` prompt, with the .EXE extension replacing the .OBJ extension
`List File`	The special file name NUL.MAP, which tells LINK not to create a map file
`Libraries`	The default libraries encoded in the object module (see Section 13.2.5, "How LINK Searches for Libraries")
`Definitions File`	The special file name NUL.DEF, which tells LINK not to use a definitions file

13.2.4 Linking with a Response File

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

LINK @*responsefile*

Here *responsefile* specifies the name or pathname of the response file that starts the linker. You can also enter the name of a response file after any LINK command prompt or at any position in the LINK command line.

A response file contains responses to the LINK prompts. The responses must be in the same order as the LINK prompts discussed in Section 13.2.3. Each new response must appear on a new line or begin with 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 may give options at the end of any response or place them on one or more separate lines.

LINK treats the input from the response file just as if you had entered it in response to prompts or in a command line. It treats any carriage-return and line-feed combination in the response file the same as if you had pressed ENTER in response to a prompt or included a comma in a command line.

You can use options and command characters in the response file in the same way as you would in responses you type 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 enter 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 include a line with a file name, semicolon, or carriage return for each prompt, LINK displays the appropriate prompt and waits for you to enter a response. When you type an acceptable response, LINK continues.

Example

Assume that the following response file is named `FUN.LNK`:

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

You can type the following command to run LINK and tell it to use the responses in `FUN.LNK`:

```
LINK @FUN.LNK
```

The response file tells LINK to load the four object modules `FUN`, `TEXT`, `TABLE`, and `CARE`. LINK produces an executable file named `FUN.EXE` and a map file named `FUNLIST.MAP`. The `/PAUSE` option tells LINK to pause before it produces the executable file so you can 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 discussions of the /PAUSE and /MAP options in Sections 13.3.28 and 13.3.15, respectively, for more information about these options.

13.2.5 How LINK Searches for Libraries

The material in this section does not apply to libraries that LINK finds in the *objectfiles* field, either on the command line or in response to the `Object Modules` prompt. Those libraries are treated simply as a series of object files, and LINK does not conduct extensive searches in such cases.

LINK may be directed to find a particular library by the user (who specifies a library in the *libraries* field) or by an object module. (When a compiler creates an object module from a higher-level-language program, that object module will contain the names of one or more "default" libraries.) However the linker is directed to a library, LINK uses the same method for finding that library.

If the library name includes a path specification, LINK searches only that directory for the library. Libraries specified by object modules (that is, default libraries) will normally not include a path specification.

If a library name is given without a path specification, LINK searches in the following three locations to find the given library file:

1. The current working directory
2. Any path specifications or drive names that you give on the command line or type in response to the `Libraries` prompt, in the order in which they appear (see Section 13.2.3)
3. The locations given by the LIB environment variable

Because object files created by compilers and assemblers usually contain the names of all the standard libraries you need, you are not required to specify a library on the LINK command line or in response to the LINK `Libraries` prompt unless you want to do one of the following:

- Add the names of additional libraries to be searched
- Search for libraries in different locations
- Override the use of one or more default libraries

For example, if you have developed your own customized libraries, you might want to include one or more of them as additional libraries at linking time.

Searching Additional Libraries

You can tell LINK to search additional libraries by specifying one or more library files on the command line or in response to the `Libraries` prompt. LINK searches these libraries before it searches default libraries. It searches these libraries in the order you specify.

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

Searching Different Locations for Libraries

You can tell LINK to search additional locations for libraries by giving a drive name or path specification in the *libraries* field on the command line or in response to the `Libraries` prompt. You can specify up to 32 additional paths. If you give more than 32 paths, LINK ignores the additional paths without displaying an error message.

Overriding Libraries Named in Object Files

If you do not want to link with the library whose name is included in the object file, you can give the name of a different library instead. You might want to specify a different library name in the following cases:

- If you assigned a "custom" name to a standard library when you set up your libraries
- If you want to link with a library that supports a different math package other than the math package you gave on the compiler command line (or the default)

If you specify a new library name on the LINK command line, the linker searches the new library to resolve external references before it searches the library specified in the object file.

If you want the linker to ignore the library whose name is included in the object file, use the /NOD option. This option tells LINK to ignore the default-library information that is encoded in the object files created by high-level-language compilers. Use this option with caution; see the discussion of the /NOD option in Section 13.3.16 for more information.

Example

```
LINK

Object Modules [.OBJ]: FUN TEXT TABLE CARE
Run File [FUN.EXE]:
List File [NUL.MAP]:
Libraries [.LIB]: C:\TESTLIB\ NEWLIBV3
```

This example links four object modules to create an executable file named `FUN.EXE`. LINK searches `NEWLIBV3.LIB` before searching the default libraries to resolve references. To locate `NEWLIBV3.LIB` and the default libraries, the linker searches the current working directory, then the `C:\TESTLIB\` directory, and finally the locations given by the LIB environment variable.

13.2.6 LINK Memory Requirements

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. This temporary file is handled in one of the following ways, depending on the DOS version:

- The linker will use the directory specified by the TMP environment variable, for the purpose of creating a temporary file. For example, if the TMP variable were set to `C:\TEMPDIR`, LINK would put the temporary file in `C:\TEMPDIR`.

  If there is no TMP environment variable, or if the directory specified by TMP does not exist, then LINK will put the temporary file in the current working directory.

- 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 temporary-file directory.

- If the linker is running on a version of DOS prior to 3.0, it creates a temporary file named VM.TMP.

When the linker creates a temporary disk file, you see the message

```
Temporary file tempfile has been created.
Do not change diskette in drive, letter.
```

In the message displayed above, *`tempfile`* is ".\" followed by either VM.TMP or a name generated by DOS, and `letter` is the drive containing the temporary file.

The message `Do not change diskette in drive` will not appear unless the drive is a removable disk. After this message appears, do 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, and you may see the following message:

```
Unexpected end-of-file on scratch file
```

When this happens, 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 linker displays an error message if it encounters an existing file with this name.*

13.3 Specifying Linker Options

This section explains how to use linker options to specify and control the tasks performed by LINK. All options begin with the linker's option character, the forward slash (/).

When you use the LINK command line to invoke LINK, options can appear at the end of the line or after individual fields on the line. However, they must precede the comma that separates each field from the next.

If you respond to the individual prompts for the LINK command, you can specify linker options at the end of any response. When you specify more than one option, you can either group the options at the end of a single response or distribute the options among several responses. Every option must begin with the slash character (/), even if other options precede it on the same line. Similarly, in a response file, options can appear on a line by themselves or after individual response lines.

Abbreviations

Since linker options are named according to their functions, some of these names are quite long. You can abbreviate the options to save space and effort. Be sure that your abbreviation is unique so the linker can determine which option you want. (The minimum legal abbreviation for each option is indicated in the syntax description of the option.)

Abbreviations must begin with the first letter of the option and must be continuous through the last letter typed. No gaps or transpositions are allowed. Options may be entered as uppercase or lowercase.

Numerical Arguments

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

- A decimal number from 0 to 65,535.
- An octal number from 0 to 177,777. A number is interpreted as octal if it starts with 0. For example, the number `10` is interpreted as a decimal number, but the number `010` is interpreted as an octal number, equivalent to 8 in decimal.
- A hexadecimal number from 0 to FFFF. A number is interpreted as hexadecimal if it starts with 0X. For example, `0X10` is a hexadecimal number, equivalent to 16 in decimal.

13.3.1 Aligning Segment Data (/A)

Option

/A[[LIGNMENT]]:*size*

The ALIGNMENT option directs LINK to align segment data in the executable file along the boundaries specified by *size*. The *size* argument must be a power of two. For example,

```
/ALIGNMENT:16
```

indicates an alignment boundary of 16 bytes. The default alignment for OS/2-application and dynamic-link segments is 512. This option is used for linking Windows applications or protected-mode programs.

13.3.2 Running in Batch Mode (/BA)

Option

/BA[[TCH]]

By default, the linker prompts you for a new path name whenever it cannot find a library it has been directed to use. It also prompts you if it cannot find an object file, and it expects that file to be on a removable disk. If the /BA option is used, however, the linker will not prompt you for any libraries or object files it cannot find. Instead, the linker will generate an error or warning message, if appropriate. The /BA option also prevents LINK from printing the sign-on banner and echoing input from response files.

Using this option can cause unresolved external references. It is intended primarily for users who use batch or MAKE files for linking many executable files with a single command, and who wish to prevent linker operation from halting.

NOTE *This option does not prevent the linker from prompting for command-line arguments. You can prevent such prompting only by using a semicolon on the command line.*

13.3.3 Producing a .COM File (/BI)

/BI[[NARY]]

The /BI option directs the linker to produce a .COM file instead of an .EXE file. Not every program can be produced in the .COM format. The following restrictions apply:

1. The program must consist of only one physical segment. If you have written an assembly-language program, you can declare more than one segment; however, the segments must be in the same group.
2. The code must have no far-segment references.

   Specifically, segment addresses cannot be used as immediate data for instructions. You could not, for example, use the following instruction:

   ```
   mov     ax,CODESEG
   ```
3. Programs for the Windows and OS/2 protected-mode environments cannot be converted to .COM file.

When you use the /BI option, the default file extension of the output file is .COM rather than .EXE.

13.3.4 Preparing for Debugging (/CO)

Option

/CO[[DEVIEW]]

The /CO option is used to prepare for debugging with the CodeView window-oriented debugger. 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, use the special-format executable file only for debugging. You can then link a separate version without the /CO option after the program is debugged.

13.3.5 Setting the Maximum Allocation Space (/CP)

Option

/CP[[ARMAXALLOC]]:*number*

The /CP option sets the maximum number of 16-byte paragraphs needed by the program when it is loaded into memory. The operating system uses this value when allocating space for the program before 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. This option is valid only when linking real-mode programs.

LINK normally requests the operating system to set the maximum number of paragraphs to 65,535. Since this represents more memory than could be available under DOS, 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 allocates no more space than the option specified. This means any additional space in memory is free for other programs.

The *number* can be any integer value in the range 1 to 65,535. 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 whatever the minimum value happens to be. 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 medium- and large-memory models, link with /CP:1; this leaves no space for the near heap.

NOTE *You can change the maximum allocation after linking by using the EXEMOD utility, which modifies the executable-file header, as described in Section 17.1. The format of the executable-file header is also discussed in that section, as well as in the Microsoft MS-DOS Programmer's Reference and in other reference books on DOS.*

13.3.6 Ordering Segments (/DO)

Option

/DO[[SSEG]]

The /DO option is automatically enabled by a special object-module record in Microsoft language libraries. If you are linking to one of these libraries, you do not need to specify this option.

This option is also enabled by assembly modules that use the Microsoft Macro Assembler directive **.DOSSEG**.

The /DO 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:
   a. Any segments of class BEGDATA (this class name reserved for Microsoft use)
   b. Any segments not of class BEGDATA, BSS, or STACK
   c. Segments of class BSS
   d. Segments of class STACK

Note that when the /DO option is in effect the linker initializes two special variables as follows:

```
_edata = DGROUP : BSS
_end   = DGROUP : STACK
```

The variables `_edata` and `_end` have special meanings for the Microsoft C and FORTRAN compilers, so it is not wise to give these names to your own program variables. Assembly modules can reference these variables but should not change them.

13.3.7 Controlling Data Loading (/DS)

Option

/DS[[ALLOCATE]]

By default, LINK loads all data starting at the low end of the data segment. At run time, the data segment (DS) register is set to the lowest possible address to allow the entire data segment to be used. This option is valid only when linking real-mode programs.

Use the /DS 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 /DS option is typically used with the /HI option, discussed in Section 13.3.11, to take advantage of unused memory within the data segment.

WARNING *This option should be used only with assembly-language programs.*

13.3.8 Packing Executable Files (/E)

Option

/E[[XEPACK]]

The /E option directs LINK to remove sequences of repeated bytes (typically null characters) and to optimize the load-time-relocation table before creating the executable file. (The load-time-relocation table is a table 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.)

Executable files linked with this option may be smaller, and thus load faster, than files linked without this option. However, you cannot use the Symbolic Debug Utility (SYMDEB) or the CodeView window-oriented debugger to debug packed files. The /EXEPACK option strips symbolic information from the input file and notifies you of this with a warning message.

The /E option does not always give a significant saving 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 are usually shorter if packed. LINK notifies you if the packed file is larger than the unpacked file.

13.3.9 Optimizing Far Calls (/F)

Option

/F[[ARCALLTRANSLATION]]

Using the /F option may result in slightly faster code and smaller executable-file size. It should be used with the /PACKD option, described in Section 13.3.25, in order to achieve significant results. The gain in speed is most apparent for 286- and 386-based machines. Though some assembly programs should not be linked with this option, it is generally safe for use with high-level-language programs. This option is off by default; furthermore, it can always be turned off with the /NOF option described in Section 13.3.18.

The rest of this section describes the low-level details of /F. It is not necessary that you understand these details in order to use the option.

The /F option directs the linker to optimize far calls to procedures that lie in the same segment as the caller. For example, a medium- or large-model program may have a machine instruction that makes a far call to a procedure in the same segment. Since the segment address is the same (for both the instruction and the procedure it calls), only a near call should be necessary.

A near-call instruction does not require an entry in the relocation table, whereas a far-call instruction does. Therefore, use of /F (together with /PACKD) often results in smaller executable files because the relocation table is smaller. Such files will load faster.

When /F has been specified, the linker will optimize code by removing the instruction `call FAR label` and substituting the following sequence:

```
push    cs
call    NEAR label
nop
```

Upon execution, the called procedure will still return with a far-return instruction. However, because both the code segment and the near address are on the stack, the far return will be executed correctly. The `nop` (no-op) instruction appears so that exactly five bytes replace the five-byte far-call instruction; the linker may in some cases place the `nop` at the beginning of the sequence.

The /F option has no effect on programs that only make near calls. Of the high-level Microsoft languages, only small- and compact-model C programs use near calls.

NOTE *There is a small risk involved with the /F option: the linker may mistakenly translate a byte in a code segment that happens to have the far-call opcode (9A hexadecimal). If a program linked with /F inexplicably fails, you may want to try linking with this option off. However, object modules produced by Microsoft high-level languages should be safe from this problem because little immediate data is stored in code segments.*

In general, assembly-language programs are safe for use with the /F option if they do not involve advanced system-level code, such as might be found in operating systems or interrupt handlers.

13.3.10 Viewing the Options List (/HE)

Option

/HE[LP]

The /HELP option causes LINK to display a list of the available options on the screen. This gives you a convenient reminder of the available options. Do not give a file name when using the /HELP option.

13.3.11 Controlling Executable-File Loading (/HI)

Option

/HI[[GH]]

At load time, the executable file can be placed either as low or as high in memory as possible. Use of the /HI option causes DOS to place the executable file as high as possible in memory. Without the /HI option, DOS places the executable file as low as possible. This option is valid only when linking real-mode programs.

NOTE *This option should be used only with assembly-language programs.*

13.3.12 Preparing for Incremental Linking (/INC)

Option

/INC[[REMENTAL]]

The /INC option must be used in order to prepare for subsequent linking with ILINK. The use of this option produces a .SYM file and an .ILK file, each containing extra information needed by ILINK. Note that this option is not compatible with the /EXEPACK option. (See Chapter 14, "Incremental Linking with ILINK," for more information on this option.)

13.3.13 Displaying Linker Process Information (/INF)

Option

/INF[[ORMATION]]

The /INF option tells the linker to display information about the linking process, including the phase of linking and the names of the object files being linked. This option is useful if you want to determine the locations of the object files being linked and the order in which they are linked.

Output from this option is sent to standard output.

The following is a sample of the linker output when the /INF and /MAP options are specified on the LINK command line:

```
**** PASS ONE ****
TEST.OBJ(test.for)
**** LIBRARY SEARCH ****
LLIBFOR7.LIB(wr)
LLIBFOR7.LIB(fmtout)
LLIBFOR7.LIB(ldout)
   .
   .
   .
**** ASSIGN ADDRESSES ****
  1 segment "TEST_TEXT" length 122H bytes
  2 segment "_DATA" length 912H bytes
  3 segment "CONST" length 12H bytes
   .
   .
   .
**** PASS TWO ****
TEST.OBJ(test.for)
LLIBFOR7.LIB(wr)
LLIBFOR7.LIB(fmtout)
LLIBFOR7.LIB(ldout)
   .
   .
   .
**** WRITING EXECUTABLE ****
```

13.3.14 Including Line Numbers in the Map File (/LI)

Option

/LI[[NENUMBERS]]

You can include the line numbers and associated addresses of your source program in the map file by using the /LI 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. You can use the /Zd option with any Microsoft compiler to include line numbers in the object file. If you give LINK an object file without line-number information, the /LI option has no effect.

The /LI option forces LINK to create a map file even if you did not explicitly tell the linker to create a map file. 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 specifying a new map file on the LINK command line or in response to the `List File` prompt.

13.3.15 Listing Public Symbols (/M)

Option

/M[[AP]]

You can list all public (global) symbols defined in the object file(s) by using the /M option. When you invoke LINK with the /M option, the map file will contain a list of all the symbols sorted by name and a list of all the symbols sorted by address. If you do not use this option, the map file contains only a list of segments.

When you use this option, the default for the *mapfile* field or prompts response is no longer NUL. Instead, the default is a name that combines the base name of the executable file with a .MAP extension. It is still possible for you to specify NUL in the *mapfile* field (which indicates that no map file is to be generated); if you do, then the /M option will have no effect.

13.3.16 Ignoring Default Libraries (/NOD)

Option

/NOD[[EFAULTLIBRARYSEARCH]][[:*filename*]]

The /NOD option tells LINK not to search any library specified in the object file to resolve external references. If you specify *filename*, LINK searches all libraries specified in the object file except for *filename*.

In general, higher-level-language programs do not work correctly without a standard library. Thus, if you use the /NOD option, you should explicitly specify the name of a standard library.

13.3.17 Ignoring Extended Dictionary (/NOE)

Option

/NOE[[XTDICTIONARY]]

The /NOE option prevents the linker from searching the extended dictionary, which is an internal list of symbol locations that the linker maintains. Normally, the linker consults this list to speed up library searches.The effect of the /NOE option is to slow the linker. You often need to use this option when a library symbol is redefined. The linker issues error `L2044` if you need to use this option.

13.3.18 Disabling Far-Call Optimization (/NOF)

Option

/NOF[ARCALLTRANSLATION]

This option normally is not necessary because far-call optimization (translation) is turned off by default. However, if an environment variable such as LINK (or CL) turns on far-call translation automatically, you can use /NOF to turn far-call translation back off again.

13.3.19 Preserving Compatibility (/NOG)

Option

/NOG[ROUPASSOCIATION]

The /NOG 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 early versions of Microsoft language compilers. This option is valid only when linking real-mode programs.

NOTE *This option should be used only with assembly-language programs.*

13.3.20 Preserving Case Sensitivity (/NOI)

Option

/NOI[GNORECASE]

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 /NOI option, the linker distinguishes between uppercase letters and lowercase letters, and considers `ABC`, `abc`, and `Abc` to be three separate names. Since names in some high-level languages are not case sensitive, this option can have minimal importance. However, in some languages—such as C— case is significant. If you plan to link your files from other high-level languages with C routines, you may want to use this option.

13.3.21 Ordering Segments without Inserting NULL Bytes (/NON)

Options

/NON[[ULLSDOSSEG]]

The /NON option directs the linker to arrange segments in the same order as they are arranged by the /DOSSEG option. The only difference is that the /DOSSEG option inserts 16 null bytes at the beginning of the _TEXT segment (if it is defined), whereas /NON does not insert these extra bytes.

If the linker is given both the /DOSSEG and /NON options, the /NON option will always take precedence. Therefore, you can use /NON to override the /DOSSEG comment record commonly found in run-time libraries. This option is for linking protected-mode programs or Windows applications.

13.3.22 Disabling Segment Packing (/NOP)

Option

/NOP[[ACKCODE]]

This option is normally not necessary because code-segment packing is turned off by default. However, if an environment variable such as LINK (or CL) turns on code-segment packing automatically, you can use /NOP to turn segment packing back off again.

13.3.23 Setting the Overlay Interrupt (/O)

Option

/O[[VERLAYINTERRUPT]]:*number*

By default, the interrupt number used for passing control to overlays is 63 (3F hexadecimal). The /O option allows the user to select a different interrupt number. This option is valid only when linking real-mode programs.

The *number* can be a decimal number from 0 to 255, an octal number from octal 0 to octal 0377, or a hexadecimal number from hexadecimal 0 to hexadecimal FF. Numbers that conflict with DOS interrupts can be used; however, their use is not advised.

You should use this option only when you want to uses overlays with a program that already reserves interrupt 63 for some other purpose.

13.3.24 Packing Contiguous Data Segments (/PACKC)

Option

/PACKC[[ODE]][[:*number*]]

This option only affects code segments in medium- and large-model programs. It is intended to be used with the /F option, which is described in Section 13.3.9. It is not necessary to understand the details of the /PACKC option in order to use it. You only need to know that this option, used in conjunction with /F, produces slightly faster and more compact code. The /PACKC option is off by default, and can always be turned off with the /NOP option described in Section 13.3.22.

The /PACKC option directs the linker to group together neighboring code segments. Segments in the same group are assigned the same segment address; offset addresses are adjusted upward accordingly. In other words, all items will have the correct physical address whether the /PACKC option is used or not. However, /PACKC changes segment and offset addresses so that all items in a group share the same segment address.

The *number* field specifies the maximum size of groups formed by /PACKC. The linker will stop adding segments to a group as soon as it cannot add another segment without exceeding *number*. At that point, the linker starts forming a new group. The default for *number* is 65,530.

The packaging of code segments provides more opportunities for far-call optimization, which is enabled with /F. Generally speaking, /F and /PACKC are designed to be used together.

Programs developed with Microsoft high-level languages can safely use /PACKC. The /PACKC option is unsafe only when used with assembly programs that make assumptions about the relative order of code segments. For example, the following assembly code attempts to calculate the distance between `CSEG1` and `CSEG2`. This code would produce incorrect results when used with /PACKC, because /PACKC causes the two segments to share segment address. Therefore the procedure would always return zero.

```
CSEG1       SEGMENT PUBLIC 'CODE'
.
.
.
CSEG1       ENDS

CSEG2       SEGMENT PARA PUBLIC 'CODE'
            ASSUME  cs:CSEG2

; Return the length of CSEG1 in AX.;
```

```
codesize    PROC  NEAR
            mov   ax,CSEG2    ; Load para address of CSEG1;
            sub   ax,CSEG1    ; Load para address of CSEG2;
            mov   cx,4        ; Load count, and
            shl   ax,c l      ; convert distance from paragraphs
                              ; to bytes;
codesize    ENDP

CSEG2       ENDS
```

13.3.25 Packing Contiguous Data Segments (/PACKD)

Option

/PACKD[[ATA]][[:*number*]]

This option only affects code segments in medium- and large-model programs. This option is also safe with all Microsoft high-level language compilers. It behaves exactly like /PACKCODE except it applies to data segments, not code segments. The linker recognizes data segments as any segment definition with a class name which does not end in `CODE`. The adjacent data segment definitions are combined into the same physical segment up to the given limit. The default limit is 65,536.

With large and compact-model programs containing many modules, it may be necessary to use this option to get around the limit of 255 physical data segments per executable file imposed by OS/2 and Windows. If you get error `L1073` from the liner, try using this option.

The *number* field specifies the maximum size of groups formed by /PACKD. The linker will stop adding segments to a group as soon as it cannot add another segment without exceeding *number*. At that point, the linker starts forming a new group. The default for *number* is 65,530.

This option may not be safe with other compilers that do not generate fixup records for all far data references. This option is valid for OS/2 and Windows programs only.

13.3.26 Padding Code Segments (/PADC)

Option

/PADC[[ODE]]:*padsize*

The /PADC option causes LINK to add filler bytes to the end of each code module for subsequent linking with ILINK. The option is followed by a colon

and the number of bytes to add. (A decimal radix is assumed, but you can specify octal or hexadecimal numbers by using a C-language prefix.) Thus

```
/PADCODE:256
```

adds an additional 256 bytes to each module. The default size for code-module padding is 0 bytes. If you are going to use this option, you must also specify the /INC option.

NOTE *Code padding is usually not necessary for large-and medium-memory-model programs, but is recommended for small-compact and mixed-memory-model programs, and for Microsoft Macro Assembler (MASM) programs in which code segments are grouped.*

To be recognized as a code segment, a segment must be declared with class name `'CODE'`. The class name need only end with 'CODE' (Microsoft high-level languages automatically use this declaration for code segments.)

13.3.27 Padding Data Segments (/PADD)

Syntax

/PADD[[ATA]]:*padsize*

The /PADD option performs a function similar to the /PADCODE option, except it specifies padding for data segments (or data modules, if the program uses small- or medium-memory model). This option is supplied for subsequent linking with ILINK. Thus

```
/PADDATA:32
```

adds an additional 32 bytes to each data module. The default size for data-segment padding is 16 bytes. If you are going to use the /PADD option, you must also specify the /INC option.

NOTE *If you specify too large a value for pad size, you may exceed the 64K limitation on the size of the default data segment.*

13.3.28 Pausing during Linking (/PAU)

Option

/PAU[[SE]]

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

If you specify the /PAU option, LINK displays the following message before it creates 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 ENTER .

NOTE *Do not remove the disk that will receive the list file or the disk used for the temporary file.*

If a temporary file is created on the disk you plan to swap, press CTRL+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 linking again.

For more information on how LINK determines where to put the temporary file, see Section 13.2.6, "LINK Memory Requirements."

13.3.29 Specifying User Libraries for Quick Languages (/Q)

Option

/Q[[UICKLIB]]

The /Q option directs the linker to produce a "Quick library,"suitable for use with Microsoft QuickBASIC or Microsoft QuickC® programs, instead of producing a stand-alone application. (Stand-alone applications are executable files that need only the presence of DOS to run. The linker produces these by default.)

No other option is necessary to enable Quick-library creation. When you use /Q, the *exefile* field refers to a Quick library instead of to an application. The default extension for this field is then .QLB instead of .EXE. You can use all of the linker features to build a Quick library that you would otherwise use to build an application. The principal difference is that a Quick library does not require (and should not contain) any main-program-level code.

A Quick library is similar to a standard software library in that both contain a collection of routines that may be called upon by a program. The two libraries are different, however: a standard library is brought together with a program at link time; a Quick library, by contrast, is brought together with a program at run time.

NOTE *Two special restrictions apply to use of a Quick library:*

1. Quick libraries can be loaded only by programs created with QuickC or QuickBASIC. These programs have the special code that properly loads a Quick library at run time.

2. Routines in a Quick library can be called from any module at run time. However, Quick-library routines cannot themselves make calls to routines outside the library. In other words, Quick libraries must be self-contained.

The linker creates a Quick library not by linking it to a program, but instead by placing into a file all of the object modules to be included and by adding a location table of all of the library routines. This table allows references to be resolved at run time, after the entire library is loaded into memory. For further information on the use of these libraries, consult the user's guide for QuickBASIC or QuickC.

13.3.30 Setting Maximum Number of Segments (/SE)

Option

/SE[[GMENTS]]:*number*

The /SE option controls the number of segments the linker allows a program to have. The default is 128, but you can set *number* to any value (decimal, octal, or hexadecimal) in the range 1 to 3,072 (decimal). However, the number of segment definitions is constrained by memory usage. Therefore, the practical limit to the number is around 1,500.

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 is able to link faster and allocate less storage space.

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* field to reflect the actual number of segments in the program. If the number of segments allocated is too high for the amount of memory LINK has available to it, you will see the following error message:

```
segment limit too high
```

To specify a number of segments that will fit in the amount of memory available, set the segment lower and relink the object files.

13.3.31 Controlling Stack Size (/ST)

Option

/ST[ACK]:*number*

The /ST option allows you to specify the size of the stack for your program. The *number* is any positive value (decimal, octal,or hexadecimal) up to 65,535 (decimal). It represents the size, in bytes, of the stack.

If you get a stack-overflow message, you may need to increase the size of the stack. In contrast, if your program uses the stack very little, you may save some space by decreasing the stack size.

NOTE *You can also use the EXEMOD utility, described in Section 17.1, to change the default stack size in DOS executable files by modifying the executable-file header. The format of the executable-file header is discussed in that section as well as in the Microsoft MS-DOS programmer's Reference and in other reference books on DOS.*

13.3.32 Issuing Fixup Warnings (/W)

Option

/W[ARNFIXUP]

The /WARNFIXUP option directs the linker to issue a warning for each segment relative fixup of location-type "offset," such that the segment is contained within a group but is not at the beginning of the group. The linker will include the displacement of the segment from the group in determining the final value of the fixup, contrary to what happens with DOS executable files. This option is for linking protected-mode programs or Windows applications.

13.4 Selecting Options with the LINK Environment Variable

You can use the LINK environment variable to cause certain options to be used each time you link. The linker checks the environment variable for options, if the variable exists.

The linker expects to find options listed in the variable exactly as you would type them on the command line. It will not accept other kinds of arguments; file names in the environment variable will cause the following error message:

```
unrecognized option
```

Each time you link, you can specify other options in addition to the ones specified in the LINK environment variable. If you type an option both on the command line and in the environment variable, the effect will be the same as if the option were given once.

Example

```
>SET LINK=/NOI /SE:256 /CO
>LINK TEST;
>LINK /NOD /CO PROG;
```

In the example above, the file `TEST.OBJ` is linked with the options `/NOI`, `/SE:256`, and `/CO`. The file `PROG.OBJ` is then linked with the option `/NOD`, in addition to `/NOI`, `/SE:256`, and `/CO`.

NOTE *A command-line option will override the effect of any environment-variable option that it conflicts with. For example, the command-line option* `/SE:512` *cancels the effect of the environment-variable option* `/SE:256`.

The only way to prevent an option in the environment variable from being used is to reset the environment variable itself.

13.5 Linker Operation

LINK performs the following steps to combine object modules and produce an executable file:

1. Reads the object modules submitted
2. Searches the given libraries, if necessary, to resolve external references
3. Assigns addresses to segments
4. Assigns addresses to public symbols
5. Reads code and data in the segments

6. Reads all relocation references in object modules
7. Performs fixups
8. Outputs an executable file (executable image and relocation information)

Steps 5, 6, and 7 are performed concurrently—in other words, LINK moves back and forth between these steps before it progresses to step 8.

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.

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

13.5.1 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.

13.5.2 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 (as described in Section 13.5.1, above). The address consists of an offset and a "canonical frame number." The canonical frame number specifies the address of the first paragraph in memory containing one or more bytes of the segment. (A paragraph is 16 bytes of memory; therefore, to compute a physical location in memory, multiply the frame number by 16 and add the offset.) 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. (An offset of zero means that the physical location is an exact multiple of 16.)

The frame number of a segment can be obtained from the map file created by LINK. The first four digits of the start address give the frame number in hexadecimal. For example, a start address of `0C0A6` gives a frame number of `0C0A`.

13.5.3 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 /DOSSEG option may change the way in which segments are ordered.

13.5.4 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**, and **PRIVATE.**

If a segment has combine type **PUBLIC**, LINK automatically combines 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**, then LINK carries out the same combine operation as for **PUBLIC** segments. The only exception is **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**, then 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.

13.5.5 Groups

Groups allow segments to be addressed 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, belong to the same class, or have the same combine type. The only requirement is 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. LINK does not explicitly check that all segments in a group fit within 64K of memory; however, LINK is likely to encounter a fixup-overflow error if this requirement is not met.

13.5.6 Fixups

Once the starting address of each segment in a program is known and all segment combinations and groups have been established, LINK can "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 fixups for the types of references shown in the following list:

Type of Reference	Description
Short	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 if the target is more than 128 bytes distant in either direction.

Near self relative	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 are not in the same segment or group.
Near segment relative	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.
Long	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.

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

13.6 Using Overlays

You can direct LINK to create an overlaid version of a program. In an overlaid version of a program, specified parts of the program (known as "overlays") are loaded only if and when they are needed. These parts share the same space in memory. Only code is overlaid; data are never overlaid. Programs that use overlays usually require less memory, but they run more slowly because of the time needed to read and reread the code from disk into memory.

When you use overlays, the linker loads in code for the overlay manager. This code resides in each of the Microsoft high-level language libraries (so you must link with at least one such library), and is between 2K and 3K in size.

You specify overlays by enclosing them in parentheses in the list of object files that you submit to the linker. Each module in parentheses represents one overlay. For example, you could give the following object-file list in the *objfiles* field of the LINK command line:

```
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 appear only once in a program.

The linker replaces 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). By default, the interrupt number is 63 (3F hexadecimal). You can use the /OVERLAYINTERRUPT option of the LINK command to change the interrupt number.

The CodeView debugger is now compatible with overlaid modules. In fact, in the case of large programs, you may need to use overlays to leave sufficient room for the debugger to operate.

13.6.1 Restrictions on Overlays

You can overlay only modules to which control is transferred and returned by a standard 8086 long (32-bit) call/return instruction. Therefore, because calls to subroutines modified with the **NEAR** attribute are short (16-bit) calls, you cannot overlay modules containing **NEAR** subroutines if other modules call those subroutines. You cannot use long jumps with the **longjmp** library function. Also, the linker does not produce overlay modules that can be called indirectly through function pointers.

13.6.2 Overlay-Manager Prompts

The overlay manager is part of the language's 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.

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 that an executable program called `PAYROLL.EXE` uses overlays and does not exist in either the current directory or the directories specified by PATH. If the user runs `PAYROLL.EXE` (by entering a complete path specification), the overlay manager displays 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 or directory, or both, 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 enter `\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 does not find `PAYROLL.EXE` and displays 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 displays the following message:

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

CHAPTER 14

14 Incremental Linking with ILINK

The Microsoft Incremental Linker (ILINK) is a utility that enables you to link your application much faster. You can benefit from its use when you change a small subset of the modules used to link a program. The program can use any memory model, but in the small model LINK is not efficient unless no symbolic change address is used. Furthermore, to benefit from ILINK, you need to follow certain restrictions that are described in this chapter. Should ILINK fail to link your changes into the executable file, it attempts to invoke the full linker, LINK, or carry out any other commands you specify on the command line. Before you can use ILINK, you must first run the full linker with special options.

NOTE *You can use ILINK to develop dynamic-link libraries as well as applications. Everything said in this chapter about applications and executable files applies to dynamic-link libraries as well. This chapter uses the term "library" to refer specifically to an object-code library (a .LIB file).*

14.1 Definitions

Incremental linking involves certain specialized concepts. You may need to review the following list of terms in order to understand the rest of this chapter:

Term	Meaning
Segment	A contiguous area of memory up to 64K in size. See the definitions of "physical segment" and "logical segment" below.

Module	A unit of code or data defined by one source file. In BASIC, Pascal, and large-memory-model C and FORTRAN programs, each module corresponds to a different segment. In small-memory-model programs, all code modules contribute to one code segment, and all data modules contribute to one data segment.
Memory model	The memory model determines the number of code and data segments in a program. BASIC programs are always large memory model.
Physical segment	A segment listed in the executable file's segment table. Each physical segment has a distinct segment address, whereas logical segments may share a segment address. A physical segment usually contains one logical segment, but it can contain more.
Logical segment	A segment defined in an object module. Each physical segment other than DGROUP contains exactly one logical segment, except when you use the GROUP directive in a MASM module. (Linking with the /PACKCODE option can also create more than one logical segment per physical segment.)
Code symbol	The address of a function, subroutine, or procedure.
Data symbol	The address of a global or static data object. The concept of data symbol includes all data objects except local (stack-allocated) or dynamically allocated data.

14.2 Guidelines for Using ILINK

Since ILINK can automatically invoke LINK when an incremental link fails, you do not have to concern yourself with the following guidelines. However, if you are interested in how ILINK works and want to take full advantage of ILINK, follow the guidelines presented in this section.

The incremental linker, ILINK, works much faster than the full linker because ILINK replaces only those modules that have changed since the last linking. It avoids much of the work done by LINK.

To enable incremental linking, follow the major guidelines below. If your changes exceed the scope allowed by these guidelines, a full link is necessary.

1. Do not alter any .LIB files you are using to create the executable file.
2. Put padding at the end of data and small-memory-model code modules by specifying the /PADCODE and /PADDATA options during full linking with LINK.

   By putting padding at the end of a module, you enable the module to grow without forcing a full relinking. However, if the module is the only module contributing to its physical segment, padding is not necessary.

   You can avoid padding if you have a BASIC, Pascal, FORTRAN, or C program (other than a small-memory-model C program), if you do not call a MASM module that uses the **GROUP** directive, and if you do not add to the size of the default data segment. (See your language documentation for information about what is placed in the default data segment.)
3. Do not delete code symbols (functions and procedures) referenced by another module. You can, however, move or add to these symbols.
4. Do not move or delete data symbols (global data). You can add data symbols at the end of your data definitions, but you cannot add new communal data symbols (for example, C uninitialized variables or BASIC **COMMON** statements).

14.3 The Development Process

To develop a software project with ILINK, follow the steps listed below:

1. Use the full linker during early stages of developing your application or dynamic-link library. ILINK produces no significant gain in speed until you have a number of different code and data modules present.
2. Prepare for incremental linking by using the special liner options mentioned below.
3. Incrementally link with ILINK after any small changes are made.
4. Relink with LINK after any major changes are made (for example, if you want to add an entirely new module, greatly expand one of the segments or modules, or redefine symbols that are shared between segments).
5. Repeat steps 3 and 4 as necessary.

You may find it easiest to use a make file to invoke ILINK and LINK. The following sample make file attempts to use ILINK each time, but invokes the full linker whenever incremental linking is not possible:

```
dog.exe: obj1.obj; obj2.obj; obj3.obj
    ILINK -e "LINK /incr @dog.lnk" -a dog
```

Three options—/INCREMENTAL, /PADCODE, and /PADDATA—have been added to LINK to allow the use of ILINK. Here is an example of how they might appear on the command line:

```
LINK /INCREMENTAL /PADDATA:16 /PADCODE:256 @PROJNAME.LNK
```

These options are described in detail in Sections 13.3.12, 13.3.27, and 13.3.26, respectively.

14.4 Running ILINK

You can attempt to link the project with ILINK at any time after the project has been linked with the /INCREMENTAL option. The following two sections discuss the files needed for linking with ILINK and the command required to invoke ILINK.

14.4.1 Files Required for Using ILINK

The files that are required for linking using ILINK are ILINK.EXE, ILINKSTB.OVL, and your project files that include the following:

1. *projname*.EXE (the file to be incrementally linked)
2. *projname*.SYM (the symbol file)
3. *projname*.ILK (the ILINK support file)
4. *.OBJ (the changed .OBJ files)

ILINK.EXE and ILINKSTB.OVL should be in a directory listed in the PATH environment variable, and the rest of the project files should be in the current directory.

14.4.2 The ILINK Command Line

The syntax for the ILINK command line is shown below. ILINK options are not case sensitive.

ILINK [[/a]] [[/c]] [[/v]] [[/i]] [[/e "*commands*"]] *projname*[[*modulelist*]]

The /a option specifies that all object files are to be checked to see if they have changed since the last linking process. This is done by comparing the dates and times of all .OBJ files with those of the executable file. An attempt is made to incrementally link all of the files that have changed.

The /c option specifies case sensitivity for all public symbol names.

The /v option specifies verbose mode and directs ILINK to display more information. Specifically, when in verbose mode ILINK lists the modules that have changed.

The /i option specifies that only an incremental link is to be attempted. If the incremental link fails, a full link is not performed.

The /e option specifies a list of commands to be executed if the incremental link fails. The commands are enclosed in double quotes, and if more than one command is given, they must be separated by spaces and a semicolon.

The *projname* field contains the name of the executable file that is to be incrementally linked.

You can use the optional *modulelist* field to list all the modules that have changed. (No extensions are required.) This field is an alternative to using the /a flag.

Examples

```
ILINK /i wizard input sort output
```

In the above example, the altered modules (`input`, `sort`, and `output`) are explicitly listed on the command line.

```
ILINK /e "link @%s.obj ; rc %s.exe" myproj
```

In the example above, the characters `%s` are replaced by *projname* when the command is carried out. If the incremental link fails, ILINK carries out the commands `link myproj.obj` and `rc myproj.exe`.

```
ILINK /a /e "link @%s.lnk ; rc %s.exe" wizard
```

In the final example above, the /a option directs ILINK to scan all files in the project in order to discover which modules have changed. This example also lists commands to be executed in case incremental linking fails.

14.5 How ILINK Works

Instead of searching for records and resolving external references for the entire program, ILINK carries out the following operations:

1. ILINK replaces the portion of each module that has changed since the last linking (incremental or full linking).
2. ILINK alters relocation-table entries for any far (segmented) code symbols that have moved within a segment. For each reference to a far code symbol, such as a far function call, there is an entry in the relocation table in the executable file's header. The relocation table of the application contains full addresses. Therefore, by fixing relocation table entries for a code symbol, ILINK ensures that all references to the symbol will be correct.)

ILINK makes no modification to modules that have not been changed since the last linking.

14.6 Incremental Violations

There are two kinds of ILINK failures: real errors and incremental violations. Real errors are errors that will not be resolved by a full link, such as undefined symbols or invalid .OBJ files. If ILINK detects a real error, it displays `ERROR` with an explanation and returns a nonzero error code to the operating system. Incremental violations consist of changes that are beyond the scope of incremental linking, but can generally be resolved by full linking.

Section C.2, "LINK Error Messages," explains real errors in detail. The rest of this section describes incremental violations.

14.6.1 Changing Libraries

An incremental violation occurs when a library changes. Furthermore, if an altered module shares a code segment with a library, ILINK needs access to the library as well as to the new module.

NOTE *If you add a function, procedure, or subroutine call to a library that has never been called before, ILINK fails with an undefined-symbol error. Performing a full link should resolve this problem.*

14.6.2 Exceeding Code/Data Padding

An incremental violation will occur if two or more modules contribute to the same physical segment and either module exceeds its padding. As discussed in Section 14.2, "Guidelines for Using ILINK," padding is the process of adding filler bytes to the end of a module. The filler bytes serve as a buffer zone whenever the module grows in size—that is, whenever the new version of the module is larger than the old.

14.6.3 Moving/Deleting Data Symbols

An incremental violation occurs if a data symbol is moved or deleted. To add new data symbols without requiring a full link, add the new symbols at the end of all other data symbols in the module.

14.6.4 Deleting Code Symbols

You can move or add code symbols, but an incremental violation occurs if you delete any code symbols from a module. Code symbols can be moved within a module but cannot be moved between modules.

14.6.5 Changing Segment Definitions

An incremental violation results if you add, delete, or change the order of segment definitions. If you are programming in MASM, an incremental violation will also result if you alter any **GROUP** directives.

If you are programming with a high-level language, remember not to add or delete modules between incremental links.

14.6.6 Adding CodeView Debugger Information

If you included CodeView debugger information for a module the last time you ran a full link (by compiling and linking with CodeView debugger support), ILINK fully supports CodeView debugger information for the module. ILINK maintains symbolic information for current symbols, and it adds information for any new symbols. However, if you include CodeView debugger information for a module that previously did not have CodeView debugger support, an incremental violation results.

CHAPTER 15

Managing Libraries with LIB

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. After you have linked a program with a run-time library file, that program can call a run-time routine exactly as if the function were included in the program. The call to the run-time routine is resolved by finding that 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 DOS object files and library files, LIB can read the contents of 286 XENIX® archives and Intel-style libraries and combine their contents with DOS libraries. To see how you can add the contents of a 286 XENIX archive or an Intel-style library to a DOS library, refer to Section 15.2.8, "Combining Libraries."

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: you can give LIB all the input it requires directly from the command line. You can also use one of the two alternative methods of invoking LIB by responding to prompts or by creating a response file, described in Sections 15.1.2 and 15.1.3 below.

15.1 Managing Libraries

You run LIB by typing the LIB command on the DOS command line. You can specify the input required for this command in one of three ways:

1. By placing it on the command line
2. By responding to prompts
3. By specifying a file containing responses to prompts (This type of file is known as a "response file.")

NOTE *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 an object-module name.*

15.1.1 Managing Libraries with the LIB Command Line

You can start LIB and specify all the input it needs from the command line. In this case, the LIB command line has the following form:

LIB *oldlibrary* [[*options*]] [[*commands*]][[,[[*listfile*]][[,[[*newlibrary*]]]]][[;]]

To tell LIB to use the default responses for the remaining fields, use a semicolon (;) after any field except the *oldlibrary* field. The semicolon should be the last character on the command line.

Sections 15.1.1.1–15.1.1.5 below describe the input you give in each command-line field.

15.1.1.1 Specifying the Library File

Field

oldlibrary[[;]]

The *oldlibrary* field allows you to specify the name of the existing library to be used. 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. There is no default for this field. 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. This message is suppressed if the nonexistent library name you give is followed immediately by commands, a comma, or a semicolon.

If you type a library 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.

15.1.1.2 Specifying Options

Field

[[*options*]]

The following list gives the options available and the function of each:

Option	Function
/PA[[GESIZE]] :*number*	This option allows you to specify the library-page size of a new library or change the library-page size of an existing library. The page size of a library affects the alignment of modules stored in the library. Modules in the library are always aligned to start at a position that is a multiple of the page size (in bytes) from the beginning of the file. The default page size for a new library is 16 bytes. See Section 15.2.11, "Setting the Library Page Size," for more information. The abbreviation for this option is /PA.

/NOI[GNORECASE]

This option tells LIB not to ignore case when comparing symbols. By default, LIB ignores case. Using this option allows symbols that are the same except for case to be put in the same library. The abbreviation for this option is /NOI.

Note that if a library is built with /NOI, the library is internally "marked" to indicate /NOI is in effect. All libraries built with earlier versions of LIB are not marked. If you combine multiple libraries, and any one of them is marked /NOI, then /NOI is assumed to be in effect for the output library.

/I[GNORECASE]

This option tells LIB to ignore case when comparing symbols, as LIB does by default. Use this option when you are combining a library that is marked /NOI with others that are unmarked and want the new library to be unmarked. (See the explanation for the /NOI option above.) The abbreviation for this option is /I.

/NOE[XTDICTIONARY]

This option is used to prevent LIB from creating an extended dictionary. The extended dictionary is used by LINK to speed up a library search. Without an extended dictionary, the .LIB extension is still a valid library, but LINK takes longer to find modules in this file. Use it if you get error messages `U1172` or `U4158`. The option /NOE [XTDICTIONARY] also occurs in LINK. In LINK the option means, "do not read an extended dictionary."

15.1.1.3 Giving LIB Commands

Field

[[*commands*]]

The *commands* field allows you to specify the command symbols for manipulating modules. To use this field, type a command symbol (such as +, –, – +, *, or –*), followed immediately by a module name or an object-file name. You can specify more than one operation in this field in any order. LIB does not make any changes to *oldlibrary* if you leave the *commands* field blank.

Command Symbol	Meaning
+	The add command symbol. A 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 may use path names for the object file. LIB automatically supplies the .OBJ extension so you can omit the extension from the object-file name. 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.
–	The delete command symbol. A 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.
– +	The replace command symbol. A minus sign followed by a plus sign replaces 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. 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.

*	The copy command symbol. An asterisk followed by a module name copies 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 to the module name the .OBJ extension and the drive designation and path name of the current working directory, thus forming a complete object-file name. You cannot override the .OBJ extension, drive designation, or path name given to the object file. However, you can later rename the file or copy it to any location you like.
-*	The move command symbol. A minus sign followed by an asterisk moves 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.

15.1.1.4 Specifying a Cross-Reference-Listing File

Field

[[*listfile*]]

The *listfile* field allows you to specify 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. The default when you omit the response to this prompt is the special file named NUL, which tells LIB not to create a listing file.

A cross-reference-listing file contains the following two lists:

1. An alphabetical list of all public symbols in the library.

   Each symbol name is followed by the module name in which it is referenced.

2. A list of the modules in the library.

   Under each module name is an alphabetical listing of the public symbols defined in that module.

15.1.1.5 Specifying an Output Library

Field

[[*newlibrary*]]

The *newlibrary* field allows you to specify the name of the new library file that contains the specified changes. You need not give this name unless you specify changes to the library in the *commands* field. 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.

Examples

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

The example above uses the replace command symbol (– +) 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 no listing file is created and the changes are written to the original library file instead of a new library file.

```
LIB LANG-HEAP+HEAP;

LIB LANG+HEAP-HEAP;
```

The examples above perform the same function as the first example in this section, but in two separate operations, using the add (+) and delete (–) command symbols. The effect is the same for these examples because delete operations are always carried out before add 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.

```
LIB FOR;
```

The example above causes LIB to perform a consistency check of the library file `FOR.LIB`. No other action is performed. LIB displays any consistency errors it finds and returns to the operating-system level.

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

This example tells LIB to perform a consistency check of the library file `LANG.LIB` and then create a cross-reference-listing file named `LCROSS.PUB`.

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

This 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`; the module 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 using the move command symbol (–*). The original library, `FIRST.LIB`, remains unchanged.

15.1.2 Managing Libraries with the LIB Prompts

If you want to respond to individual prompts to give input to LIB, start LIB at the DOS command level by typing LIB. The library manager prompts you for the input it needs by displaying the following four messages, one at a time:

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

LIB waits for you to respond to each prompt, then prints the next prompt.

The responses you give to the LIB command prompts correspond to the fields on the LIB command line. (See Section 15.1.1 for a discussion of the LIB command line.) The following list shows these correspondences:

Prompt	Command-Line Field
`Library name`	The *oldlibrary* field and the options (see Sections 15.1.1.1 and 15.1.1.2, respectively). If you want to perform a consistency check on the library, type a semicolon (;) immediately after the library name.
`Operations`	Any of the commands allowed in the *commands* field (see Section 15.1.1.3).
`List file`	The *listfile* field (see Section 15.1.1.4).
`Output library`	The *newlibrary* field (see Section 15.1.1.5).

15.1.2.1 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 operation's symbol and a name.

The ampersand causes LIB to repeat the `Operations` prompt, allowing you to type more operations.

15.1.2.2 Using Default Responses

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 necessary since LIB supplies the default responses wherever you omit responses.

The following list shows the defaults for LIB prompts:

Prompt	Default
`Operations`	No operation; no change to library file.
`List file`	The special file name NUL, which tells LIB not to create a listing file.
`Output library`	The current library name. Only if you specify at least one operation at the `Operations` prompt will this prompt appear.

15.1.3 Managing Libraries with a Response File

To operate LIB with a response file, you must first set up the response file and then type the following at the DOS command line:

LIB @*responsefile*

The *responsefile* 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 at any position in a command line or after any of the linker prompts. The input from the response file is treated exactly as if it had been entered in command lines or after prompts. A carriage-return and line-feed combination in the response file is treated the same as pressing ENTER in response to a prompt or using a comma in a command line.

Before you use this method, you must set up a response file containing responses to the LIB prompts. This method lets you conduct the library session without typing responses to prompts 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. You can type an ampersand at the end of the response to the `Operations` prompt and continue typing operations on the next line.

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 responses for all the prompts, LIB uses the default responses.

Example

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

The contents of the above response file cause LIB to delete the module HEAP from the LIBFOR.LIB library file, copy the module FOIBLES, place it in an object file FOIBLES.OBJ, and append the object files CURSOR.OBJ and HEAP.OBJ as the last two modules in the library. LIB creates a cross-reference-listing file named CROSSLST.

15.1.4 Terminating the LIB Session

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

15.2 Performing Library-Management Tasks with LIB

You can perform a number of library-management functions with LIB, including the following tasks:

- Create a library file
- Delete modules
- Copy a module to a separate object file
- Move a module out of a library and into an object file (extract module)
- Append an object file as a module 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 in the order listed below. It determines whether a new library is being created or an existing library is being examined or modified.

1. LIB processes any deletion and move commands.

   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.

2. LIB processes any additional 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 move commands, a new library file would be 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 linker uses the library index 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 change a library file, LIB lets you specify a different name for the file containing the changes. 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.

15.2.1 Creating a Library File

To create a new library file, give the name of the library file you want to create in the *oldlibrary* field of the command line or at the `Library name` prompt. LIB supplies the .LIB extension.

The name of the new library file must not be the name of an existing file. If it is, LIB assumes that you want to change 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. This message is suppressed if the nonexistent library name you give is followed immediately by commands, a comma, or a semicolon.

You can specify a page size for the library when you create it. The default page size is 16 bytes. See Section 15.2.11, "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 command symbol (+) in the *commands* field of the command line or at the `Operations` prompt. You can also add

the contents of another library, if you wish. See Section 15.2.3, "Adding Library Modules," and Section 15.2.8, "Combining Libraries," for a discussion of these options.

15.2.2 Changing a Library File

You can change an existing library file by giving the name of the library file at the `Library name` prompt. All operations you specify in the *oldlibrary* field of the command line or at the `Operations` prompt are performed on that library.

However, LIB lets you keep both the unchanged library file and the newly changed version, if you like. You can do this by giving the name of a new library file in the *newlibrary* field or at the `Output library` prompt. The changed library file is stored under the new library-file name, while the original library file remains unchanged.

If you don't give a new file name, the changed version of the library file replaces the original library file. Even in this case, LIB saves the original, unchanged library file with the extension .BAK instead of .LIB. Thus, at the end of the session you have two library files: the changed version with the .LIB extension and the original, unchanged version with the .BAK extension.

15.2.3 Adding Library Modules

Use the add command symbol (+) in the *commands* field of the command line or 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.

15.2.4 Deleting Library Modules

Use the delete command symbol (–) in the *commands* field of the command line or at the `Operations` prompt to delete an object module from a library. After the minus sign, give the name of the module to be deleted. A module name does not have a path name or extension; it is simply a name, such as `CURSOR`.

15.2.5 Replacing Library Modules

Use the replace command symbol (– +) in the *commands* field to replace a module in the library. Following the replace command symbol, give the name of the module to be replaced. Remember that module names do not have path names or 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.

15.2.6 Copying Library Modules

To copy a module from the library file into an object file of the same name, use the copy command symbol (*) followed by a module name in the *commands* field. 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. This forms 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 any location you like.

15.2.7 Moving Library Modules (Extracting)

Use the move command symbol (–*) in the *commands* field 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.

15.2.8 Combining Libraries

You can add the contents of a library to another library by using the add command symbol (+) with a library-file name instead of an object-file name in the *commands* field. In the *commands* field or at the `Operations` prompt, give the add command symbol (+) followed by the name of the library whose contents you wish to add to the library being changed. 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 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 DOS format.

LIB adds the modules of the library to the end of the library being changed. 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 in the *newlibrary* field of the command line or 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 changed. The original library is saved with the same base name and the extension .BAK.

15.2.9 Creating a Cross-Reference-Listing File

Create a cross-reference-listing file by giving a name for the listing file in the *listfile* field of the command line or at the `List file` prompt. If you do not give a listing-file name, LIB uses the special file name NUL, which means 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.

15.2.10 Performing Consistency Checks

When you give only a library name followed by a semicolon in the *oldlibrary* field of the command line or at the `Library name` prompt, LIB performs a consistency check, displaying messages about any errors it finds. No changes are made to the library. It is not usually necessary to perform consistency checks since LIB automatically checks object files for consistency before adding them to the library.

To produce a cross-reference-listing file with a consistency check, invoke LIB, specify the library name followed by a semicolon, and give the name of the listing file. LIB then performs the consistency check and creates the cross-reference-listing file.

15.2.11 Setting the Library-Page Size

You can set the library-page size while you are creating a library, and you can change the page size of an existing library 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:

/PA[GESIZE] :*number*

The *number* specifies the new page size. It must be an integer value representing a power of 2 between the values 16 and 32,768.

The page size of a library affects the alignment of modules stored in the library. Modules in the library are always aligned to 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 pagesize/2 bytes of storage space is wasted. 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 * 65,536. For example,* `/P:16` *means that the .LIB file has to be smaller than 1 megabyte (16 * 65,536 bytes).*

CHAPTER 16
NMAKE

The Microsoft Program-Maintenance Utility (NMAKE) can save you time by automating the process of updating project files. NMAKE compares the modification dates for one set of files, the target files, to those of another set of files, the dependent files. If any of the dependent files have changed more recently than the target files, NMAKE executes a specified series of commands.

NMAKE is typically used by specifying a project's executable files as target files and the project's source files as the dependent files. If any of the source files have changed since the executable file was created, NMAKE can issue a command to assemble or compile the changed source files and link them into the executable file.

NMAKE reads the target- and dependent-file specifications from a "description file," also called a "makefile." The description file comprises any number of description blocks. Each description block lists one or more targets and the dependent files related to those targets. The block also gives the commands that NMAKE must execute to bring the targets up to date. The description file may also contain macros, inference rules, and directives.

16.1 Invoking NMAKE

Two methods for invoking NMAKE are available:

1. Specify options, macro definitions, and the names of targets to be built on the DOS command line.
2. Specify options, macro definitions, and the names of targets to be built in a command file, and give the file name on the DOS command line.

16.1.1 Using a Command Line to Invoke NMAKE

The syntax for invoking NMAKE from the command line is as follows:

NMAKE [[*options*]] [[*macrodefinitions*]] [[*target...*]] [[*filename*]]

The *options* field specifies options that modify the action of NMAKE. (Options are not required.) They are described in Section 16.2.

The optional *macrodefinitions* field lists macro definitions for NMAKE to use. Macros provide a convenient method for replacing a string of text in the description file. Macro definitions that contain spaces must be enclosed by quotation marks. Macros are discussed in Section 16.3.2.

The optional *target...* field specifies the name of one or more targets to build. If you do not list any targets, NMAKE builds the first target in the description file.

The optional *filename* field gives the name of the description file from which NMAKE reads target- and dependent-file specifications and commands. A better way of designating the description file is to use the /F option (described in Section 16.2). By default, NMAKE looks for a file named MAKEFILE in the current directory. If MAKEFILE does not exist, NMAKE uses the *filename* field; it interprets the first string on the command line that is not an option or macro definition as the name of the description file, provided its file-name extension isn't listed in the .SUFFIXES list. (See Section 16.3.5 for more information on the .SUFFIXES list.)

NOTE *Unless you use the /F option, NMAKE always searches for a file named MAKEFILE in the current directory.*

Example

```
NMAKE /S "program = flash" sort.exe search.exe
```

This example invokes NMAKE with the /S option, a macro assigning `flash` to `program`, and two targets, `sort.exe` and `search.exe`. By default, NMAKE uses the file named MAKEFILE as the description file.

16.1.2 Using a Command File to Invoke NMAKE

To invoke NMAKE with a command file, first create the command file, then issue a command with the following syntax:

NMAKE @*commandfile*

Here *commandfile* is the name of a file containing the same information that would be specified on the command line: options, macro definitions, and targets. The command file is not the same as the description file.

A command file is useful for invoking NMAKE with a long string of command-line arguments, such as macro definitions, that might exceed the DOS limit of 128 characters. NMAKE treats line breaks that occur between arguments as spaces. Macro definitions can span multiple lines by ending each line except the last with a backslash (\). Macro definitions that contain spaces must be enclosed by quotation marks, just as if they were entered directly on the command line.

Example

```
/S "program \
= flash" sort.exe search.exe
```

Assume a file named `update` contains the text above. The command below invokes NMAKE with the description file MAKEFILE, the /S option, the macro definition `program = flash`, and the targets `sort.exe` and `search.exe`. Note that the backslash ending the line allows the macro definition to span two lines.

```
NMAKE @update
```

16.2 NMAKE Options

NMAKE accepts a number of command-line options, which are listed below. You may specify options in uppercase or lowercase and use either a slash or dash. For example, –B, /B, –b, and /b all represent the same option.

Option	Action
/A	Executes commands to build all the targets requested even if they are not out of date.
/C	Suppresses the NMAKE copyright message and prevents nonfatal error or warning messages from being displayed.
/D	Displays the modification date of each file when the date is checked.
/E	Causes environment variables to override macro definitions within description files.
/F *filename*	Specifies *filename* as the name of the description file to use. If a dash (-) is entered instead of a file name, NMAKE accepts input from the standard input device instead of using a description file. If /F is not specified, NMAKE uses the file named MAKEFILE as the description file. If MAKEFILE does not exist, NMAKE uses the first string on the command line that is not an option or macro definition as the name of the file, provided the extension is not listed in the .SUFFIXES list (see Section 16.3.5).

Option	Action
/I	Ignores exit codes (also called return or "errorlevel" codes) returned by programs called from the NMAKE description file. NMAKE continues executing the rest of the description file despite the errors.
/N	Displays the commands from the description file that NMAKE would execute but does not execute these commands. This option is useful for checking which targets are out of date and for debugging description files.
/P	Prints all macro definitions and target descriptions.
/Q	Returns a zero status code if the target is up to date and a nonzero status code if it is not. This option is useful when invoking NMAKE from within a batch file.
/R	Ignores inference rules and macros contained in the TOOLS.INI file.
/S	Does not display commands as they are executed.
/T	Changes the modification dates for out-of-date target files to the current date. The file contents are not modified.
/X *filename*	Sends all error output to *filename*, which can be either a file or a device. If a dash (-) is entered instead of a file name, the error output is sent to the standard output device.

Examples

```
NMAKE /f quick /c f1 f2
```

The example above causes NMAKE to execute the commands in the description file `quick` to update the targets `f1` and `f2`. The /c option prevents NMAKE from displaying nonfatal error messages and warnings.

```
NMAKE  /D /N f1 f1.mak
```

In the example above, NMAKE updates the target `f1`. If the current directory does not contain a file named MAKEFILE, NMAKE reads the file `f1.mak` as the description file. The /D option displays the modification date of each file and the /N option displays the commands without executing them.

16.3 Description Files

NMAKE reads a description file to determine what to do. The description file may contain any number of description blocks, along with macros, inference rules, and directives. These can be in any order.

When NMAKE runs, it builds the first target in the description file by default. You can override this default by specifying on the command line the names of the targets to build.

The sections that follow describe the elements of a description file.

16.3.1 Description Blocks

An NMAKE description file contains one or more description blocks. Each has the following form:

target... : [[*dependent...*]] [[;*command*]][[#*comment*]]
    [[*command*]]
[[#*comment*]]
[[#*comment*]] | [[*command*]]
    .
    .
    .

The file to be updated is *target*; *dependent* is a file upon which *target* depends; *command* is a command used to update *target*; and *comment* documents what is happening. The line containing *target* and *dependent* is called the dependency line because *target* depends on *dependent*.

Each component of a description block is discussed below.

The Target Field

The *target* field specifies the name of one or more files to update. If you specify more than one file, separate the file names by a space. The first target name must start in the first column of the line; it may not be preceded by any tabs or spaces. Note that the target need not be a file; it may be a pseudotarget, as described in Section 16.3.5. A target name can have a complete path specification, i.e., drive: path filename.ext. If a target name is a single letter, then a space must be inserted before the ":" to avoid confusion with a path name, such as "a:".

The Dependent Field

The *dependent* field lists one or more files on which the target depends. If you specify more than one file, separate the file names by a space. You can specify directories for NMAKE to search for the dependent files by using the following form:

target : {*directory1*; *directory2...*}*dependent*

NMAKE searches the current directory first, then *directory1*, *directory2*, and so on. If *dependent* cannot be found in any of these directories, NMAKE looks for an inference rule to create the dependent in the current directory. See Section 16.3.3 for more information on inference rules.

In the following example, NMAKE first searches the current directory for `pass.obj`, then the `\src\alpha` directory, and finally the `d:\proj` directory:

```
forward.exe : {\src\alpha;d:\proj}pass.obj
```

The Command Field

The *command* is used to update the target. This can be any command that can be issued on the DOS command line. A semicolon must precede the command if it is given on the same line as the target and dependent files. Commands may be placed on separate lines following the dependency line, but each line must start with at least one space or tab character. Blank lines may be intermixed with commands. A long command may span several lines if each line ends with a backslash (\). If no commands are specified, NMAKE looks for an inference rule to build the target.

The Comment Field

NMAKE considers any text between a number sign (#) and a new-line character to be a comment and ignores it. You may place a comment on a line by itself or at the end of any line except a command line. In the command section of the description file, comments must start in the first column.

Wild-Card Characters

You can use the DOS wild-card characters (* and ?) when specifying target- and dependent-file names. NMAKE expands wild cards in target names when it reads the description file. It expands wild cards in the dependent names when it builds the target. For example, the following description block compiles all source files with the .C extension:

```
astro.exe : *.c
        QCL *.c
```

Escape Character

You can use a caret (^) to escape any DOS or OS/2 file-name character in a description file, so that the character takes on its literal meaning and does not have any special significance to NMAKE. Specifically, the caret escapes the following characters:

```
# ( ) $ ^ \ { } ! @ -
```

For example, NMAKE interprets the specification

```
big^#.c
```

as the file name

```
big#.c
```

Using the caret, you can include a literal new-line character in a description file. This capability is primarily useful in macro definitions, as in the following example:

```
XYZ=abc^
def
```

NMAKE interprets this example as if you had assigned to the XYZ macro the C-style string `abc\ndef`. Note that this effect differs from the use of the backslash (\) to continue a line. A new-line character that follows a backslash is replaced with a space.

NMAKE ignores a caret that is not followed by any of the characters mentioned above, as in the following:

```
mno ^: def
```

In this case, NMAKE ignores the caret and treats the line as

```
mno : def
```

Carets that appear within quotation marks are not treated as escape characters.

16.3.1.1 Modifying Commands

Three different characters may be placed in front of a command to modify its effect. The character must be preceded by at least one space, and spaces may separate the character from the command. You may use more than one character to modify a single command. The characters are listed below.

Character	Action

Dash (-)	Turns off error checking for the command. If the dash is followed by a number, NMAKE halts only if the error level returned by the command is greater than the number. In the following example, if the program `flash` returned an error code NMAKE would not halt, but would continue to execute commands:
	`light.lst:light.txt`
	`  -flash light.txt`

At sign(@)	Prevents NMAKE from displaying the command as it executes. In the example below, NMAKE does not display the ECHO command line:
	`sort.exe:sort.obj`
	`  @ECHO sorting`
	The output of the ECHO command, however, appears as usual. (This modifier does not work with DOS 2.1.)

Exclamation point (!)	Causes the command to be executed for each dependent file if the command uses one of the special macros $? or $**. The $? macro refers to all dependent files that are out of date with respect to the target, while $** refers to all dependent files in the description block. (See Section 16.3.2 for more information on macros.) For example,
	`print: hop.asm skip.bas jump.c`
	`  !print $** lpt1:`
	causes the following three commands to be generated:
	`print hop.asm lpt1:`
	`print skip.bas lpt1:`
	`print jump.c lpt1:`

16.3.1.2 Specifying a Target in Multiple Description Blocks

You can specify more than one description block for the same target by using two colons (::) as the separator instead of one. For example:

```
target.lib :: a.asm b.asm c.asm
   ML a.asm b.asm c.asm
   LIB target -+a.obj -+b.obj -+c.obj;
target.lib :: d.c e.c
   QCL /c d.c e.c
   LIB target -+d.obj -+e.obj;
```

These two description blocks both update the library named `target.lib`. If any of the assembly-language files have changed more recently than the library file, NMAKE executes the commands in the first block to assemble the source files and update the library. Similarly, if any of the C-language files have changed, NMAKE executes the second group of commands that compile the C files and then update the library.

If you use a single colon in the above example, NMAKE issues an error message. It is legal, however, to use single colons if commands are listed in only one block. In this case, dependency lines are cumulative. For example,

```
target: jump.bas
target: up.c
   commands
```

is equivalent to

```
target: jump.bas up.c
   commands
```

16.3.2 Macros

Macros provide a convenient way to replace a string in the description file with another string. The text is automatically replaced each time NMAKE is invoked. This feature makes it easy to change text used throughout the description file without having to edit every line that uses the text.

Macros can be used in a variety of situations, including the following:

- To create a standard description file for several projects. The macro represents the file names used in commands. These file names are then defined when you run NMAKE. When you switch to a different project, changing the macro changes the file names NMAKE uses throughout the description file.
- To control the options that NMAKE passes to the compiler, assembler, or linker. When you use a macro to specify the options, you can quickly change the options used throughout the description file in one easy step.

16.3.2.1 Macro Definitions

A macro definition uses the following form:

macroname = *string*

The *macroname* may be any combination of alphanumeric characters and the underscore (_) character. The *string* may be any valid string.

You can define macros on the NMAKE command line or in the description file. Because of the way DOS parses command lines, the rules for the two methods are slightly different.

Defining Macros in Description Files

In NMAKE description files, define each macro on a separate line. The first character of the macro name must be the first character on the line. NMAKE ignores spaces following *macroname* or preceding *string*. The *string* may be a null string and may contain embedded spaces. Do not enclose *string* in quotation marks; NMAKE will consider them part of the string.

Defining Macros on the NMAKE Command Line

On the command line, no spaces may surround the equal sign. Spaces cause DOS to treat *macroname* and *string* as separate tokens. Strings that contain embedded spaces must be enclosed in double quotation marks. Alternatively, you can enclose the entire macro definition—*macroname* and *string*—in quotation marks. The *string* may be a null string. C command line macro definitions override definitions of the same macro in the description file.

After you have defined a macro, use the following to include it in a dependency line or command:

$(*macroname*)

The parentheses are not required if *macroname* is only one character long. If you want to use a dollar sign ($) in the file but do not want to invoke a macro, enter two dollar signs ($$), or use the caret (^) as an escape character preceding the dollar sign.

When NMAKE runs, it replaces all occurrences of $(*macroname*) with *string*. If the macro is undefined—that is, if its name does not appear to the left of an equal sign in the file or on the NMAKE command line, NMAKE treats it as a null string. Once a macro is defined, the only way to cancel its definition is to use the **!UNDEF** directive (see Section 16.3.4).

Example

Assume the following text is in a file named MAKEFILE:

```
program = flash
c = LINK
options =

$(program).exe : $(program).obj
   $c  $(options)  $(program).obj;
```

When you invoke NMAKE, it interprets the description block as the following:

```
flash.exe : flash.obj
   LINK    flash.obj;
```

16.3.2.2 Macro Substitutions

Just as macros allow you to substitute text in a description file, you can also substitute text within a macro itself. Use the following form:

$(*macroname*:*string1* = *string2*)

Every occurrence of *string1* is replaced by *string2* in the macro *macroname*. Spaces between the colon and *string1* are considered part of *string1*. Any spaces following *string1* or preceding *string2* are ignored. If *string2* is a null string, all occurrences of *string1* are deleted from the *macroname* macro.

Example

```
SRCS = prog.c sub1.c sub2.c

DUP : $(SRCS)
    echo $(srcs)
    echo $(srcs:.c=.obj)
```

Note that the special macro $** stands for the names of all the dependent files (see Section 16.3.2.3). If the description file above is invoked with a command line that specifies both targets, NMAKE will execute the following commands:

```
    echo prog.c sub1.c sub2.c
prog.c sub1.c sub2.c
    echo prog.obj sub1.obj sub2.obj
prog.obj sub1.obj sub2.obj
```

The macro substitution does not alter the definition of the macro `SRCS`, but replaces the listed characters. When NMAKE builds the target `prog.exe`, it picks up the definition for the special macro $** (that is, the list of dependents) from the dependency line, which specifies the macro substitution in `SRCS`. The same is true for the second target, `DUP`. In this case, however, no macro substitution is requested, so `SRCS` retains its original value, and $** represents the names of the C source files.

16.3.2.3 Special Macros

Several macros have special meaning. These macros are listed below with their values:

Macro	Value
$*	The target name with the extension deleted.
$@	The full name of the current target.
$**	The complete list of dependent files.
$<	The dependent file that is out of date with respect to the target (evaluated only for inference rules).
$?	The list of dependents that are out of date with respect to the target.
$$@	The target NMAKE is currently evaluating. This is a dynamic dependency parameter that can be used only in dependency lines. See "Examples," below, for a typical use of this macro.
$(CC)	The command to invoke the C compiler. By default, NMAKE predefines this macro as `CC = cl`, which invokes the Microsoft C Optimizing Compiler. To redefine the macro to invoke the QuickC compiler, use `CC = qcl` You might want to place the above definition in your TOOLS.INI file to avoid having to redefine it for each description file.
$(AS)	The command that invokes the Microsoft Macro Assembler. NMAKE predefines this macro as `AS = masm`.

$(MAKE)	The name with which the NMAKE utility was invoked. This macro is used to invoke NMAKE recursively. It causes the line on which it appears to be executed even if the /N option is on. You may redefine this macro if you want to execute another program.
$(MAKEFLAGS)	The NMAKE options currently in effect. If you invoke NMAKE recursively, you should use the command: `$(MAKE)`. You cannot redefine this macro.

You can append characters to any of the first six macros in the above list to modify its meaning. Appending a D specifies the directory part of the file name only, an F specifies the file name, a B specifies just the base name, and an R specifies the complete file name without the extension. If you add one of these characters, you must enclose the macro name in parentheses. (The special macros $$@ and $** are the only exceptions to the rule that macro names more than one character long must be enclosed in parentheses.)

For example, assume that $@ has the value `C:\SOURCE\PROG\SORT.OBJ`. The list below shows the effect the special characters have when combined with $@:

Macro	Value
$(@D)	`C:\SOURCE\PROG`
$(@F)	`SORT.OBJ`
$(@B)	`SORT`
$(@R)	`C:\SOURCE\PROG\SORT`

Examples

```
trig.lib : sin.obj cos.obj arctan.obj
        !LIB trig.lib -+$?;
```

In the example above, the macro $? represents the names of all dependents that are more recent than the target. The exclamation point causes NMAKE to execute the LIB command once for each dependent in the list. As a result of this description, the LIB command is executed up to three times, each time replacing a module with a newer version.

```
# Include files depend on versions in current directory
DIR=c:\include
$(DIR)\globals.h : globals.h
        COPY globals.h $@
$(DIR)\types.h : types.h
        COPY types.h $@
$(DIR)\macros.h : macros.h
        COPY macros.h $@
```

This example shows the use of NMAKE to update a group of include files. In the description file above, each of the files `globals.h`, `types.h`, and `macros.h` in the directory `c:\include` depends on its counterpart in the current directory. If one of the include files is out of date, NMAKE replaces it with the file of the same name from the current directory.

The description file below, which uses the special macro $$@, is equivalent.

```
# Include files depend on versions in current directory
DIR=c:\include
$(DIR)\globals.h $(DIR)\types.h $(DIR)\macros.h: $$(@F)
        !COPY $? $@
```

In this example, the special macro $$(@F) signifies the file name (without the directory) of the current target.

When NMAKE executes the description, it evaluates the three targets, one at a time, with respect to their dependents. Thus, NMAKE first checks whether `c:\include\globals.h` is out of date compared with `globals.h` in the current directory. If so, it executes the command to copy the dependent file `globals.h` to the target. NMAKE repeats the procedure for the other two targets. Note that in the command line, the macro $? refers to the dependent for this target. The macro $@ means the full name of the target.

16.3.2.4 Precedence of Macro Definitions

If the same macro is defined in more than one place, the rule with the highest priority is used. The priority from highest to lowest is as follows:

1. Definitions on the command line
2. Definitions in the description file or in an include file
3. Definitions by an environment variable
4. Definitions in the TOOLS.INI file
5. Predefined macros such as CC and AS

If NMAKE is invoked with the /E option, which causes environment variables to override macro definitions, macros defined by environment variables take precedence over those defined in a description file.

16.3.3 Inference Rules

Inference rules are templates that NMAKE uses to generate files with a given extension. When NMAKE encounters a description block with no commands, it looks for an inference rule that specifies how to create the target from the dependent files, given the two file extensions. Similarly, if a dependent file does

not exist, NMAKE looks for an inference rule that specifies how to create the dependent from another file with the same base name.

The use of inference rules eliminates the need to put the same commands in several description blocks. For example, you can use inference rules to specify a single QCL command that changes any C source file (which has an extension of .C) to an object file (which has an extension of .OBJ).

Inference rules have the following form:

.fromext.toext:
    command
    [[*command*]]
    .
    .
    .

In this format, *command* specifies one of the commands involved in converting a file with the extension *fromext* to a file with the extension *toext*. Using the earlier example of converting C source files to object files, the inference rule looks as follows:

```
.C.OBJ:
   QCL -c $<;
```

The special macro $< represents the name of a dependent that is out of date relative to the target.

Path Specifications

You can specify a single path for each of the extensions, using the following form:

{*frompath*}.*fromext*{*topath*}.*toext*
    commands

NMAKE takes the files with the *fromext* extension it finds in the directory specified by *frompath* and uses *commands* to create files with the *toext* extension in the directory specified by *topath*.

If NMAKE finds a description block without commands, it looks for an inference rule that matches both extensions. NMAKE searches for inference rules in the following order:

1. In the current description file.
2. In the tools-initialization file, TOOLS.INI. NMAKE first looks for the TOOLS.INI file in the current working directory and then in the directory

indicated by the INIT environment variable. If it finds the file, NMAKE looks for the inference rules following the line that begins with the tag [nmake]. This begins a section that can contain all default macros, .SUFFIXES lists, and inference rules.

NOTE *NMAKE applies an inference rule only if the base name of the file it is trying to create matches the base name of a file that already exists.*

In effect, this means that inference rules are useful only when there is a one-to-one correspondence between the files with the "from" extension and the files with the "to" extension. You cannot, for example, define an inference rule that inserts a number of modules into a library.

Predefined Inference Rules

NMAKE uses three predefined inference rules, summarized in Table 16.1. Note that these rules use the macro CC, which invokes the Microsoft C Optimizing Compiler by default. If you plan to rely on inference rules to build your targets, you should redefine CC to invoke the QuickC compiler, as shown in the list in Section 16.3.2.3.

Table 16.1 Predefined Inference Rules

Inference Rule	Command	Default Action
.c.obj	$(CC) $(CFLAGS) /c $*.c	CL /c $*.c
.c.exe	$(CC) $(CFLAGS) $*.c	CL $*.c
.asm.obj	$(AS) $(AFLAGS) $*;	masm $*;

Example

```
.OBJ.EXE:
   LINK $<;

EXAMPLE1.EXE: EXAMPLE1.OBJ

EXAMPLE2.EXE: EXAMPLE2.OBJ
   LINK /CO EXAMPLE2,,,LIBV3.LIB
```

In the sample description file above, the first line defines an inference rule that executes the LINK command on the second line to create an executable file whenever a change is made in the corresponding object file. The file name in the inference rule is specified with the special macro $< so that the rule applies to any .OBJ file that has an out-of-date executable file.

When NMAKE does not find any commands in the first description block, it checks for a rule that may apply and finds the rule defined on the first two lines of the description file. NMAKE applies the rule, replacing the $< macro with `EXAMPLE1.OBJ` when it executes the command, so that the LINK command becomes

```
LINK EXAMPLE1.OBJ;
```

NMAKE does not search for an inference rule when examining the second description block because a command is explicitly given.

16.3.4 Directives

Using directives, you can construct description files that are similar to batch files. NMAKE provides directives that specify conditional execution of commands, display error messages, include the contents of other files, and turn on or off some of NMAKE's options.

Each directive begins with an exclamation point (!) in the first column of the description file. Spaces can be placed between the exclamation point and the directive keyword. The list below describes the directives.

Directive	Description
!IF *expression*	Executes the statements between the **!IF** keyword and the next **!ELSE** or **!ENDIF** directive if *constantexpression* evaluates to a nonzero value.
!ELSE	Executes the statements between the **!ELSE** and **!ENDIF** directives if the statements preceding the **!ELSE** directive were not executed.
!ENDIF	Marks the end of the **!IF**, **!IFDEF**, or **!IFNDEF** block of statements.
!IFDEF *macroname*	Executes the statements between the **!IFDEF** keyword and the next **!ELSE** or **!ENDIF** directive if *macroname* is defined in the description file. NMAKE considers a macro with a null value to be defined.
!IFNDEF *macroname*	Executes the statements between the **!IFNDEF** keyword and the next **!ELSE** or **!ENDIF** directive if *macroname* is not defined in the description file.
!UNDEF *macroname*	Marks *macroname* as being undefined in NMAKE's symbol table.
!ERROR *text*	Causes *text* to be printed and then stops execution.

!INCLUDE *filename*	Reads and evaluates the file *filename* before continuing with the current description file. If *filename* is enclosed by angle brackets (< >), NMAKE searches for the file in the directories specified by the INCLUDE macro; otherwise it looks in the current directory only. The INCLUDE macro is initially set to the value of the INCLUDE environment variable.
!CMDSWITCHES: {+|–}*opt*...	Turns on or off one of four NMAKE options: /D, /I, /N, and /S. If no options are specified, the options are reset to the way they were when NMAKE was started. Turn an option on by preceding it with a plus sign (+), or turn it off by preceding it with a minus sign (–). Using this directive updates the MAKEFLAGS macro.

The *constantexpression* used with the !IF directive may consist of integer constants, string constants, or program invocations. Integer constants can use the C unary operators for numerical negation (–), one's complement (~), and logical negation (!). They may also use any of the C binary operators listed below:

Operator	Description
+	Addition
–	Subtraction
*	Multiplication
/	Division
%	Modulus
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
&&	Logical AND
||	Logical OR
<<	Left shift
>>	Right shift
==	Equality
!=	Inequality
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

You can use parentheses to group expressions. Values are assumed to be decimal values unless specified with a leading 0 (octal) or leading 0x (hexadecimal). Use the equality (==) operator to compare two strings for equality or the inequality (!=) operator to compare for inequality. Strings are enclosed by quotes. Program invocations must be in square brackets ([ ]).

Example

```
!INCLUDE <infrules.txt>
!CMDSWITCHES +D
winner.exe:winner.obj
!IFDEF debug
!   IF "$(debug)"=="y"
      LINK /CO winner.obj;
!   ELSE
      LINK winner.obj;
!   ENDIF
!ELSE
!   ERROR Macro named debug is not defined.
!ENDIF
```

The **!INCLUDE** directive causes the file `INFRULES.TXT` to be read and evaluated as if it were a part of the description file. The **!CMDSWITCHES** directive turns on the /D option, which displays the dates of the files as they are checked. If `winner.exe` is out of date with respect to `winner.obj`, the **!IFDEF** directive checks to see if the macro `debug` is defined. If it is defined, the **!IF** directive checks to see if it is set to `y`. If it is, then the linker is invoked with the /CO option; otherwise it is invoked without it. If the `debug` macro is not defined, the **!ERROR** directive prints the message and NMAKE stops executing.

16.3.5 Pseudotargets

A "pseudotarget" is a target that is not a file but instead is a name that serves as a "handle" for building a group of files or executing a group of commands. In the following example, `UPDATE` is a pseudotarget.

```
UPDATE: *.*
        !copy $** a:\product
```

When NMAKE evaluates a pseudotarget, it always considers the dependents out of date. In the description above, NMAKE copies each of the dependent files to the specified drive and directory.

The NMAKE utility includes four predefined pseudotargets that provide special rules within a description file. The list below describes these pseudotargets.

Pseudotarget	Action
.SILENT:	Does not display lines as they are executed. Same effect as invoking NMAKE with the /S option.
.IGNORE:	Ignores exit codes returned by programs called from the description file. Same effect as invoking NMAKE with the /I option.
.SUFFIXES:*list*	Lists file suffixes for NMAKE to try if it needs to build a target file for which no dependents are specified. NMAKE searches the current directory for a file with the same name as the target file and a suffix from the list. If NMAKE finds such a file, and if an inference rule applies to the file, then NMAKE treats the file as a dependent of the target. The order of the suffixes in the list defines the order in which NMAKE searches for the files. The list is predefined as follows: `.SUFFIXES: .obj .exe .c .asm` To add suffixes to the list, specify .SUFFIXES: followed by the new suffixes. To clear the list, specify .SUFFIXES:

PRECIOUS: *target...*	Tells NMAKE not to delete *target* if the commands that build it are quit or interrupted. Using this pseudotarget overrides the NMAKE default. By default, NMAKE deletes the target if it cannot be sure the target was built successfully. For example:
	`.PRECIOUS: tools.lib`
	`tools.lib : a2z.obj z2a.obj`
	`.`
	`.`
	`.`
	If the commands (not shown here) to build `tools.lib` are interrupted, leaving an incomplete file, NMAKE does not delete the partially built `tools.lib` because it is listed with .PRECIOUS.
	Note, however, that .PRECIOUS is useful only in limited circumstances. Most professional development tools, including those provided by Microsoft, have their own interrupt handlers and "clean up" when errors occur.

16.4 Response-File Generation

At times, you may need to issue a command in the description file that has a list of arguments that exceeds the DOS limit of 128 characters. NMAKE can generate response files for use with other programs.

The syntax for creating a response file is

target : *dependents*
  command @<< [[*filename*]]
response-file-text
<<

All of the text between the two sets of double brackets (<<) is placed in a response file and given the name *filename*. The response file can be referred to at a later time using *filename*. If *filename* is not given, NMAKE gives the file a unique name in the directory specified by the TMP environment variable if it is defined; otherwise it creates it in the current directory. Note that the at sign (@) is not part of the NMAKE syntax but is the typical response-file character for utilities such as LIB and LINK.

Example

```
math.lib : add.obj sub.obj mul.obj div.obj
  LIB @<<
math.lib
-+add.obj-+sub.obj-+mul.obj-+div.obj
listing
<<
```

The above example creates a response file and uses it to invoke the Microsoft Library Manager LIB. The response file specifies which library to use, the commands to execute, and the listing file to produce. The response file contains the following:

```
math.lib
-+add.obj-+sub.obj-+mul.obj-+div.obj
listing
```

16.5 Differences between NMAKE and MAKE

NMAKE differs from MAKE in the following ways:

- It accepts command-line arguments from a file.
- It provides more command-line options.
- It no longer evaluates targets sequentially. Instead, it updates the targets you specify when you invoke NMAKE, regardless of their positions in the description file. If no targets are specified, NMAKE updates the first target in the file.
- It provides more special macros.
- It permits substitutions within macros.
- It supports directives placed in the description file.
- It allows you to specify include files in the description file.

MAKE assumed that all targets in the description file would be built. Because NMAKE builds the first target in the file unless you specify otherwise, you may need to change your old description files to work with the new utility.

Description files written for use with MAKE typically list a series of subordinate targets followed by a higher-level target that depends on the subordinates. As MAKE executed, it would build the targets sequentially, creating the highest-level target at the end.

The easiest way to convert these description files is to create a new description block at the top of the file. Give this block a pseudotarget named ALL and set its dependents to all of the other targets in the file. When NMAKE executes the description, it will assume you want to build the target ALL and consequently will build all targets in the file.

Alternatively, if your description file already contains a block that builds a single, top-level target, you can simply make that block the first in the file.

Example

```
one.obj: one.c

two.obj: two.c

three.obj: three.c

prog1.exe: one.obj two.obj three.obj
        link one two three, prog1;

x.obj: x.c

y.obj: y.c

z.obj: z.c

xyz.exe: x.obj y.obj z.obj
        link x y z, xyz;
```

Assume the above is an old MAKE description file named MAKEFILE. Note that it builds two top-level targets, `prog1.exe` and `xyz.exe`. To use this file with the new NMAKE, insert the following as the first line in the file:

```
ALL : prog1.exe xyz.exe
```

With the addition of this line, `ALL` becomes the first target in the file. Since NMAKE, by default, builds the first target, you can invoke NMAKE with

```
NMAKE
```

and it will build both `prog1.exe` and `xyz.exe`.

CHAPTER 17

Using Other Utilities

The following utilities allow you to modify files and change the operating environment:

- Microsoft EXE File Header Utility (EXEMOD)

  Modifies header information in executable files.

- Microsoft Environment Expansion Utility (SETENV)

  Enlarges the DOS environment table in IBM PC-DOS Versions 2.0, 2.1, 3.0, and 3.1. SETENV allows you to use more, larger environment variables.

- Microsoft Debug Information Compactor Utility (CVPACK)

  Compresses executable files by reducing the size of CodeView debugging information within the files.

The following sections explain how to use the EXEMOD, SETENV, and CVPACK utilities.

17.1 Modifying Program Headers with the EXEMOD Utility

The EXEMOD utility allows you to modify fields in the header of an executable file. 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 as hexadecimal numbers.

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* [[*options*]]

EXEMOD expects *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 the following error message:

```
exemod: file not .EXE
```

The EXEMOD options are shown with the forward slash (/) designator, but a dash (-) may also be used. Options can be given in either uppercase or lowercase, but they cannot be abbreviated. The EXEMOD options and their effects are described in the following list:

Option	Effect
/H	Displays the current status of the DOS program header. Its effect is the same as entering EXEMOD with an *executablefile* specification but without options. The /H option should not be used with other options.
/STACK *hexnum*	Allows you to set the size of the stack (in bytes) for your program by setting the initial SP (stack pointer) value to *hexnum*. The minimum allocation value is adjusted upward, if necessary. This option has the same effect as the LINK /STACK option, except it works on files that are already linked.
/MIN *hexnum*	Sets the minimum allocation value (that is, the minimum number of 16-byte paragraphs needed by the program when it is loaded into memory) to *hexnum*. The actual value set may be different from the requested value if adjustments are necessary to accommodate the stack.
/MAX *hexnum*	Sets the maximum allocation value (that is, the maximum number of 16-byte paragraphs used by the program when it is loaded into memory) to *hexnum*. 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.

For each of the options listed above, *hexnum* is a number entered using hexadecimal digits (uppercase or lowercase); no prefix is needed.

NOTE *Use of the /STACK option on programs developed with other than Microsoft compilers or assemblers 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 prints the message

```
packed file
```

then continues 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 has after expansion. If either the /MIN or the /STACK option is used, the value is 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 are after expansion. These values are not the same as the actual values in the header of the packed file.

Example

```
Microsoft (R) EXE File Header Utility  Version 4.02
Copyright (C) Microsoft Corp 1985.  All rights reserved.

TEST.EXE                              (hex)          (dec)

.EXE size (bytes)                      439D          17309
Minimum load size (bytes)              419D          16797
Overlay number                            0              0
Initial CS:IP                     0403:0000
Initial SS:SP                     0000:0000
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 display above shows how EXEMOD would display the current file header for file `TEST.EXE`. Note that `(para)` refers to paragraphs, which are units of 16 bytes. To translate paragraphs to bytes, multiply by 16. The meaning of each field is given below.

`.EXE size` is the size of the file as stored on disk. `Minimum load size` is the total amount of memory that DOS must provide in order for the program to execute.

`Overlay number` is the ordinal number of the overlay as generated by LINK. (If the executable file does not use overlays, there is exactly one overlay module, the root.) Since EXEMOD looks only at the beginning of the file, the overlay number displayed is normally 0.

`Initial CS:IP` and `Initial SS:SP` indicate the initial values of the instruction pointer and the stack pointer, respectively. The values of CS and SS are relative to the beginning of the load module and are changed once the file is actually loaded into memory. The offset address of the stack pointer (SP) indicates the amount of room available for the stack to grow downward before reaching SS. (Some of this room may be needed by other segments, however.) The initial value of SP can be changed with EXEMOD.

`Minimum allocation` indicates the amount of memory that the file requires, in addition to the memory that DOS uses to load the file itself. If DOS is unable to allocate this amount of memory, it does not execute the file. This value can be changed with EXEMOD.

`Maximum allocation` indicates the amount of memory the file requests, in addition to memory used to load the file itself. If the amount specified is not available, DOS allocates all available memory. This value can be changed with EXEMOD.

`Header size` gives the size of all header information, including relocation entries.

`Relocation table offset` indicates the number of bytes from the beginning of the file to the relocation entries.

`Relocation entries` gives the number of relocation entries. Each of these entries is a piece of information used to adjust segment addresses in the load module (the portion of the file that is actually loaded into memory). DOS adds the load address to each segment address so that the segment address refers to a true location in physical memory.

Examples

```
>EXEMOD TEST.EXE
```

The command in the above example generates the display in the previous example for the file `TEST.EXE`.

```
EXEMOD TEST.EXE /STACK FF /MIN FF /MAX FFF
```

The example above uses the EXEMOD command line to modify the header fields in `TEST.EXE`.

```
>EXEMOD TEST.EXE

Microsoft (R) EXE File Header Utility  Version 4.02
Copyright (C) Microsoft Corp 1985.  All rights reserved.

TEST.EXE                                (hex)          (dec)

.EXE size (bytes)                        439D          17309
Minimum load size (bytes)                528D          20877
Overlay number                              0              0
Initial CS:IP                      0403:0000
Initial SS:SP                      0000:00FF            256
Minimum allocation (para)                FF            256
Maximum allocation (para)               FFF           4095
Header size (para)                       20             32
Relocation table offset                  1E             30
Relocation entries                        1              1
```

The last example shows the current status of the header for `TEST.EXE` after being altered by the previous example.

17.2 Enlarging the DOS Environment with the SETENV Utility

The SETENV utility allows you to allocate more operating-environment space to DOS by modifying a copy of COMMAND.COM.

Normally, DOS Versions 2.0 and later allocate 160 bytes (10 paragraphs) for the environment table. This may not be enough space 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 want to set several environment variables.

NOTE *SETENV works with most MS-DOS and PC-DOS operating systems, Versions 2.0 through 3.1. If the SETENV utility does not work with your version of COMMAND.COM, please contact Microsoft Technical Support.*

If you use DOS 3.2 or later, you can set the environment space with the DOS SHELL command. For example, the following command sets the environment size at 3000 bytes when placed in CONFIG.SYS:

```
SHELL = COMMAND.COM /E:3000 /P
```

See your DOS manual for further information.

To enlarge the environment table, you can modify a copy of COMMAND.COM using SETENV. 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 can be renamed. 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 65,520. The specified *environmentsize* is rounded up to the nearest multiple of 16 (the size of a paragraph).

If *environmentsize* is not given, SETENV reports the value currently allocated by the COMMAND.COM file.

After modifying COMMAND.COM, you must reboot so that the environment table is set to the new size.

Examples

```
>SETENV COMMAND.COM

Microsoft (R) Environment Expansion Utility  Version 2.10
Copyright (C) Microsoft Corp 1985,1986.  All rights
reserved.

command.com: Environment allocation = 160
```

In the example above, no environment size is specified, so SETENV reports the current size of the environment table.

```
>SETENV COMMAND.COM 605
```

In the example above, 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 automatically sets an environment table of 608 bytes (38 paragraphs). You must reboot to set the new environment-table size.

17.3 Saving Memory with the CVPACK Utility

After you compile and link a program with CodeView debugging information, you can use the Microsoft Debug Information Compactor Utility (CVPACK) to reduce the size of the executable file. CVPACK compresses information in the file, and allows the CodeView debugger to load larger programs without running out of memory.

The CVPACK utility has the following command line:

CVPACK [[/p]] *exefile*

The /p option results in the most effective possible packing but causes CVPACK to take longer to execute. When the /p option is specified, unused debugging information is discarded, and the packed information is sorted within the file. When the /p option is not specified, packed information is simply appended to the end of the file.

To debug a file that has been altered with CVPACK, you must use Version 2.10 or later of the CodeView debugger.

CHAPTER 18

18 Linking for Windows and OS/2 Systems

This chapter covers concepts important to linking for Windows and OS/2 systems, such as dynamic-linking and import libraries. Section 18.6 describes the IMPLIB utility for creating import libraries.

In most respects, linking a program using the Microsoft Segmented-Executable Linker (LINK) for the OS/2 environment is similar to linking a program for the DOS 3.x environment. The principal difference is that most programs created for the DOS 3.x environment run as stand-alone applications, whereas programs that run under OS/2 protected mode generally call one or more "dynamic-link libraries."

18.1 Dynamic-Link Libraries

A dynamic-link library contains executable code for common functions, just as an ordinary library does. Yet code for dynamic-link functions is not linked into the executable (.EXE) file. Instead, the library itself is loaded into memory at run time, along with the .EXE file.

Each .DLL file (dynamic-link library) must use "export definitions" to make its functions directly available to other modules. At run time, functions not exported can only be called from within the same file. Each export definition specifies a function name.

Conversely, the .EXE file must use "import definitions" that tell where each dynamic-link function can be found. Otherwise, OS/2 would not know what dynamic-link libraries to load when the program is run. Each import definition specifies a function name and the .DLL file where the function resides.

Assume the simplest case, in which you create one application and one dynamic-link library. The linker requires export and import definitions for dynamic-link

function calls. The OS/2 operating system provides two ways for you to supply these definitions:

- You create one module-definition file (.DEF extension) with export definitions for the .DLL file and another module-definition file with import definitions for the .EXE file. The module-definition files provide these definitions in an ASCII format.
- You create one module-definition file (.DEF extension) for the .DLL file and then generate an import library to be linked to the .EXE file.

The next two sections consider each of these methods in turn. Chapter 19, "Using Module-Definition Files," gives a complete description of module-definition files.

18.2 Linking without an Import Library

Figure 18.1 illustrates the first way to supply definitions for dynamic-link function calls, in which each of the two files—the .DLL file and the .EXE file—has a corresponding module-definition file. (A module-definition file has a .DEF default extension.)

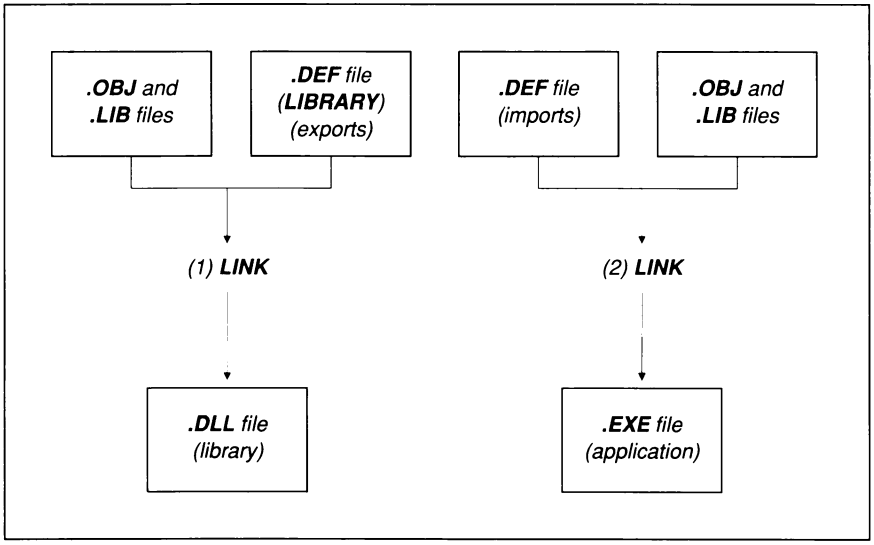


Figure 18.1 Linking without an Import Library

The two major steps are described below.

1. Object files (and possibly standard-library files) are linked together with a module-definition file to create a dynamic-link library. A module-definition file for a dynamic-link library has at least two statements. The first is a **LIBRARY** statement, which directs the linker to create a .DLL rather than an .EXE file. The second statement is a list of export definitions.

2. Object files (and possibly standard-library files) are linked together with a module-definition file to create an application. The module-definition file for this application contains a list of import definitions. Each definition in this list contains both a function name and the name of a dynamic-link library.

18.3 Linking with an Import Library

Figure 18.2 illustrates the second way to supply definitions for dynamic-link function calls, in which a module-definition file is supplied for the dynamic-link library and an import library is supplied for the application.

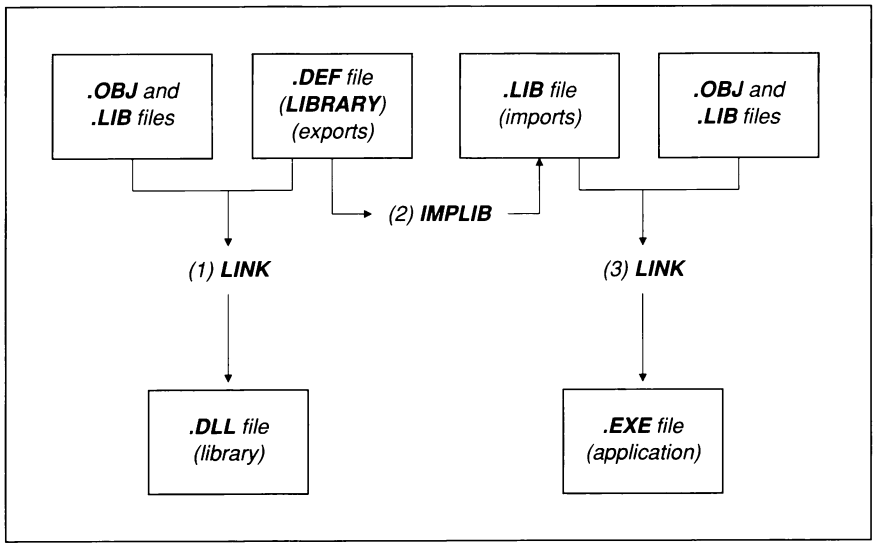


Figure 18.2 Linking with an Import Library

The three major steps are explained below.

1. Object files are linked to produce a .DLL file. This step is identical to the first step in the section above. Note that the module-definition file contains export definitions.
2. The IMPLIB utility is used to generate an import library. IMPLIB takes as input the same module-definition file used in the first step. IMPLIB knows the name of the library module (which by default has the same base name as the .DEF file), and it determines the name of each exported function by examining export definitions. For each export definition in the .DEF file, IMPLIB generates a corresponding import definition.
3. The .LIB file generated by IMPLIB is used as input to LINK, which creates an application. This .LIB file does not use the same file format as a .DEF file, but it fulfills the same purpose: to provide the linker with information about imported dynamic-link functions.

The .LIB file generated by IMPLIB is called an import library. Import libraries are similar in most respects to ordinary libraries; you specify import libraries and ordinary libraries in the same command-line field of LINK, and you can append the two kinds of libraries together (by using the Library Manager). Furthermore, both kinds of libraries resolve external references at link time. The only difference is import libraries do not contain executable code, merely records that describe where the executable code can be found at run time.

The cases considered in this section have been simple ones. Dynamic linking is flexible and supports more complicated cases. An application can make calls to more than one dynamic-link library. Furthermore, module-definition files for libraries can import functions as well as export them. It is possible for a .DLL file to call another .DLL file, and so on, to any level of complexity; the result may be a situation in which many files are loaded at run time.

18.4 Why Use Import Libraries?

At first glance, it may seem easier to create programs without import libraries since import libraries add an extra step to the linking process. However, it is easier to use import libraries for two reasons.

First, the IMPLIB utility automates much of the program-creation process for you. To run IMPLIB, you specify the .DEF file that you already created for the dynamic-link library. Operation of IMPLIB is simple. If you do not use an import library generated by IMPLIB, you must use an ASCII text editor to create a second .DEF file where you explicitly give all needed import definitions.

Second, the first two steps in the linking process described above (creation of the .DLL file and creation of the import library) may be carried out only by the author of the dynamic-link library. The libraries may then be given to an applications programmer, who focuses on linking the application (third step). An applications programmer's task is simplified by linking with the import library because then it is not necessary to edit the .DEF file. The import library comes ready to link.

A good example of a useful import library is the file DOSCALLS.LIB. Generally, protected-mode applications need to call one of the dynamic-link system libraries released with OS/2; the DOSCALLS.LIB file contains import definiions for all calls to these system libraries. It is much easier to link with DOSCALLS.LIB than to create a .DEF file for every OS/2 program you link.

18.5 Advantages of Dynamic Linking

Why use dynamic-link libraries at all? Dynamic-link libraries serve much the same purpose that standard libraries do but they also give you the following advantages:

1. Link applications faster.

   With dynamic linking, the executable code for a dynamic-link function is not copied into the application's .EXE file. Instead, only an import definition is copied.

2. Save significant disk space.

   Suppose you create a library function called `printit`, and this function is called by many different programs. If `printit` is in a standard library, the function's executable code must be linked into each .EXE file that calls the function. In other words, the same code resides on your disk in many different files. But if `printit` is stored in a dynamic-link library, the executable code resides in just one file—the library itself.

3. Make libraries and applications more independent.

   Dynamic-link libraries can be updated any number of times without relinking the applications that use them. If you are a user of third-party libraries, this is particularly convenient. You receive the updated .DLL file from the third-party developers, and you need only copy the new library onto your disk. At run time, your applications automatically call the updated library functions.

4. Utilize shared code and data segments.

   Code and data segments loaded in from a dynamic-link library can be shared. Without dynamic linking, this sharing is not possible because each file has its own copy of all the code and data it uses. By sharing segments with dynamic linking, you can use memory more efficiently.

18.6 Creating Import Libraries with IMPLIB

This section summarizes the use of the Microsoft Import Library Manager (IMPLIB), and assumes you are familiar with the concepts of import libraries, dynamic linking, and module-definition files discussed in Section 18.2.

You can create an import library for use by other programmers in resolving external references to your dynamic-link library. The IMPLIB command creates an import library, which is a file with a .LIB extension that can be read by the OS/2 linker. The .LIB file can be specified in the LINK command line with other libraries. Import libraries are recommended for all dynamic-link libraries. Without the use of import libraries, external references to dynamic-link routines must be declared in an **IMPORTS** statement in the module-definition file for the application being linked. IMPLIB is supported only in protected mode.

The IMPLIB command-line format is as follows:

IMPLIB *implibname mod-def-file* [[*mod-def-file*...]]

The *implibname* is the name you wish the new import library to have.

The *mod-def-file* is the name of a module-definition file for the dynamic-link module. You may enter more than one.

Example

The following command creates the import library named MYLIB.LIB from the module-definition file MYLIB.DEF:

```
IMPLIB mylib.lib mylib.def
```

CHAPTER 19

19 Using Module-Definition Files

A module-definition file describes the name, attributes, exports, imports, and other characteristics of an application or library for OS/2 or Microsoft Windows. This file is required for Windows applications and libraries and is also required for dynamic-link libraries that run under OS/2.

19.1 Module Statements

A module-definition file contains one or more "module statements." Each module statement defines an attribute of the executable file, such as its module name, the attributes of program segments, and the number and names of exported and imported functions. The module statements and the attributes they define are listed below.

Module Statements	Attribute Defined
NAME	Names application (no library created)
LIBRARY	Names dynamic-link library (no application created)
DESCRIPTION	Describes the module in one line
CODE	Gives default attributes for code segments
DATA	Gives default attributes for data segments
SEGMENTS	Gives attributes for specific segments
STACKSIZE	Specifies local-stack size in bytes
EXPORTS	Defines exported functions
IMPORTS	Defines imported functions
STUB	Adds a DOS 3.x executable file to the beginning of the module, usually to terminate the program when run in real mode
HEAPSIZE	Specifies local heap size in bytes

PROTMODE	Specifies that the module runs only in DOS protected mode
OLD	Preserves import information from a previous version of the library
REALMODE	Relaxes some restrictions that the linker imposes for protected-mode programs
EXETYPE	Identifies operating system

The following rules govern the use of module statements in a module-definition file:

- If you use either a **NAME** or a **LIBRARY** statement, it must precede all other statements in the module-definition file.
- You can include source-level comments in the module-definition file by beginning a line with a semicolon (;). The OS/2 utilities ignore each such comment line.
- All module-definition keywords (such as **NAME**, **LIBRARY**, and **OLD**) must be entered in uppercase letters.

The sample module-definition file below gives module definitions for a dynamic-link library. This sample file includes one source-level comment and five statements.

```
; Sample module-definition file

LIBRARY

DESCRIPTION 'Sample .DEF file for a dynamic-link library'

CODE        PRELOAD

STACKSIZE   1024

EXPORTS
    Init    @1
    Begin   @2
    Finish  @3
    Load    @4
    Print   @5
```

The sections below explain the meaning of these statements, as well as others, giving syntax and examples.

19.2 The NAME Statement

The **NAME** statement identifies the executable file as an application and optionally defines the name.

Syntax

NAME[[*appname*]] [[*apptype*]]

Remarks

If *appname* is given, it becomes the name of the application as it is known by OS/2. This name can be any valid file name. If *appname* is not given, the name of the module-definition file—with the extension removed—becomes the name of the application.

The *apptype* field will be used by a future version of OS/2 and should be declared for compatibility with this future version.

If *apptype* is given, it defines the type of application being linked. This information is kept in the executable-file header. You do not need to use this field unless you may be using your application in a Windows environment. The *apptype* field may have one of the following values:

Keyword	Meaning
WINDOWAPI	Real-mode Presentation Manager application. The application uses the API provided by the Presentation Manager and must be executed in the Presentation Manager environment.
WINDOWCOMPAT	Presentation Manager-compatible application. The application can run inside the Presentation Manager, or it can run in a separate screen group An application can be of this type if it uses the proper subset of OS/2 video, keyboard, and mouse functions supported in the Presentation Manager applications.
NOTWINDOWCOMPAT	Application is not compatible with the Presentation Manager and must operate in a separate screen group from the Presentation Manager.

If the **NAME** statement is included in the module-definition file, the **LIBRARY** statement cannot appear. If neither a **NAME** statement nor a **LIBRARY** statement appears in a module-definition file, the default is **NAME**—that is, the linker acts as though a **NAME** statement were included, and thus creates an application rather than a library.

Example

The example below assigns the name `calendar` to the application being defined:

```
NAME calendar WINDOWCOMPAT
```

19.3 The LIBRARY Statement

The **LIBRARY** statement identifies the executable file as a dynamic-link library and it can specify the name of the library or the type of library-module initialization required.

Syntax

LIBRARY [[*libraryname*]] [[*initialization*]]

Remarks

If *libraryname* is given, it becomes the name of the library as it is known by OS/2. This name can be any valid file name. If *libraryname* is not given, the name of the module-definition file—with the extension removed—becomes the name of the library.

The *initialization* field is optional and can have one of the two values listed below. If neither is given, then the *initialization* default is **INITGLOBAL**.

Keyword	Meaning
INITGLOBAL	The library-initialization routine is called only when the library module is initially loaded into memory
INITINSTANCE	The library-initialization routine is called each time a new process gains access to the library

If the **LIBRARY** statement is included in a module-definition file, **NAME** cannot appear. If no **LIBRARY** statement appears in a module-definition file, the linker assumes that the module-definition file is defining an application.

Example

The following example assigns the name `calendar` to the dynamic-link module being defined, and specifies that library initialization is performed each time a new process gains access to `calendar`:

```
LIBRARY calendar INITINSTANCE
```

19.4 The DESCRIPTION Statement

The **DESCRIPTION** statement inserts the specified *text* into the application or library. This statement is useful for embedding source-control or copyright information into an application or library.

Syntax

DESCRIPTION *'text'*

Remarks

The *text* is a one-line string enclosed in single quotation marks. Use of the **DESCRIPTION** statement is different from the inclusion of a comment because comments—lines that begin with a semicolon (;)—are not placed in the application or library.

Example

The following example inserts the text `Template Program` into the application or library being defined:

```
DESCRIPTION 'Template Program'
```

19.5 The CODE Statement

The **CODE** statement defines the default attributes for code segments within the application or library.

Syntax

CODE[[*attribute...*]]

Remarks

Each *attribute* must correspond to one of the following attribute fields. Each field can appear at most one time, and order is not significant. The attribute fields are presented below, along with legal values. In each case, the default value is listed last. The last three fields have no effect on OS/2 code segments and are included for use with Microsoft Windows.

Field	Values
load	**PRELOAD, LOADONCALL**
executeonly	**EXECUTEONLY, EXECUTEREAD**
iopl	**IOPL, NOIOPL**
conforming	**CONFORMING, NONCONFORMING**
shared	**SHARED, NONSHARED**
movable	**MOVABLE, FIXED**
discard	**NONDISCARDABLE, DISCARDABLE**

The *load* field determines when a code segment is to be loaded. This field contains one of the following keywords:

Keyword	Meaning
PRELOAD	The segment is loaded automatically at the beginning of the program
LOADONCALL	The segment is not loaded until accessed (the default)

The *executeonly* field determines whether a code segment can be read as well as executed. This field contains one of the following keywords:

Keyword	Meaning
EXECUTEONLY	The segment can only be executed
EXECUTEREAD	The segment can be both executed and read (the default)

The *iopl* field determines whether or not a segment has I/O privilege (that is, whether it can access the hardware directly). This field contains one of the following keywords:

Keyword	Meaning
IOPL	The code segment has I/O privilege
NOIOPL	The code segment does not have I/O privilege (the default)

The *conforming* field specifies whether a code segment is a 286 "conforming" segment. The concept of a conforming segment deals with privilege level (the range of instructions that the process can execute) and is relevant only to users writing device drivers and system-level code. A conforming segment can be called from either Ring 2 or Ring 3, and the segment executes at the caller's privilege level. This field contains one of the following keywords:

Keyword	Meaning
CONFORMING	The segment is conforming
NONCONFORMING	The segment is nonconforming (the default)

The *shared* field determines whether all instances of the program can share a given code segment. This field is ignored by OS/2, but is provided for use with real-mode Windows. Under OS/2, all code segments are shared. The *shared* field contains one of the following keywords: **SHARED** or **NONSHARED** (the default for Windows).

The *movable* field determines whether a segment can be moved around in memory. This field is ignored by OS/2, but is provided for use with real-mode Windows. Under OS/2, all segments are movable. The *movable* field contains one of the following keywords: **MOVABLE** or **FIXED** (the default for Windows).

The *discard* field determines whether a segment can be swapped out to disk by the operating system when not currently needed. This attribute is ignored by OS/2, but is provided for use with real-mode Windows. Under OS/2 systems, all segments can be swapped as needed. The *shared* attribute contains one of the following keywords: **DISCARDABLE** or **NONDISCARDABLE** (the default for Windows).

Example

The following example sets defaults for the module's code segments so they are not loaded until accessed and have I/O hardware privilege:

```
CODE LOADONCALL IOPL
```

19.6 The DATA Statement

The **DATA** statement defines the default attributes for the data segments within the application or module.

Syntax

DATA [[*attribute...*]]

Remarks

Each *attribute* must correspond to one of the following attribute fields. Each field can appear at most one time, and order is not significant. The attribute fields are present below, along with legal values. In each case, the default value is listed last. The last two fields have no effect on OS/2 data segments, but are included for use with Microsoft Windows.

Field	Values
load	**PRELOAD, LOADONCALL**
readonly	**READONLY, READWRITE**
instance	**NONE, SINGLE, MULTIPLE**
iopl	**IOPL, NOIOPL**
shared	**SHARED, NONSHARED**
movable	**MOVABLE, FIXED**
discard	**NONDISCARDABLE, DISCARDABLE**

The *load* field determines when a segment will be loaded. This field contains one of the following keywords:

Keyword	Meaning
PRELOAD	The segment is loaded when the program begins execution
LOADONCALL	The segment is not loaded until it is accessed (the default)

The *readonly* field determines the access rights to a data segment. This field contains one of the following keywords:

Keyword	Meaning
READONLY	The segment can only be read
READWRITE	The segment can be both read and written to (the default)

The *instance* field affects the sharing attributes of the automatic data segment, which is the physical segment represented by the group name DGROUP. (This segment group makes up the physical segment which contains the local stack and heap of the application.) The *instance* field contains one of the following keywords:

Keyword	Meaning
NONE	No automatic data segment is created.
SINGLE	A single automatic data segment is shared by all instances of the module. In this case, the module is said to have "solo" data. This keyword is the default for dynamic-link libraries.
MULTIPLE	The automatic data segment is copied for each instance of the module. In this case, the module is said to have "instance" data. This keyword is the default for applications.

The *iopl* field determines whether or not data segments have I/O privilege (that is, whether or not they can access the hardware directly). This field contains one of the following keywords:

Keyword	Meaning
IOPL	The data segments have I/O privilege
NOIOPL	The data segments do not have I/O privilege (the default)

The *shared* field determines whether all instances of the program can share a **READWRITE** data segment. Under OS/2, this field is ignored by the linker if the segment has the attribute **READONLY**, since **READONLY** data segments are always shared. The *shared* field contains one of the following keywords:

Keyword	Meaning
SHARED	One copy of the data segment will be loaded and shared among all processes accessing the module. This keyword is the default for dynamic-link libraries
NONSHARED	The segment cannot be shared and must be loaded separately for each process. This keyword is the default for applications

The *movable* field determines whether a segment can be moved around in memory. This field is ignored by OS/2, but is provided for use with real-mode Windows. Under OS/2, all segments are movable. The *movable* field contains one of the following keywords: **MOVABLE** or **FIXED** (the default for Windows).

The optional *discard* field determines whether a segment can be swapped out to disk by the operating system when not currently needed. This attribute is ignored by OS/2, but is provided for use with real-mode Windows. Under OS/2 systems, all segments can be swapped as needed. The *discard* attribute contains one of the following keywords: **DISCARDABLE** or **NONDISCARDABLE** (the default for Windows).

Note that the linker makes the automatic-data-segment attribute (specified by an instance value of **SINGLE** or **MULTIPLE**) match the sharing attribute of the automatic data segment (specified by a shared value of **SHARED** or **NONSHARED**). Solo data (specified by **SINGLE**) force shared data segments by default. Instance data (specified by **MULTIPLE**) force nonshared data by default. Similarly, **SHARED** forces solo data, and **NONSHARED** forces instance data.

If you give a contradictory **DATA** statement such as `DATA SINGLE NON-SHARED`, all segments in DGROUP are shared, and all other data segments are nonshared by default. If a segment that is a member of DGROUP is defined with a sharing attribute that conflicts with the automatic data type, a warning about the bad segment is issued, and the segment's flags are converted to a consistent sharing attribute. For example, the following

```
DATA SINGLE
SEGMENTS
_DATA CLASS 'DATA' NONSHARED
```

is converted to

```
_DATA CLASS 'DATA' SHARED
```

Example

The example below defines the application's data segment so it is loaded only when it is accessed and cannot be shared by more than one copy of the program.

```
DATA LOADONCALL NONSHARED
```

By default, the data segment can be read and written, the automatic-data segment is copied for each instance of the module, and the data segment has no I/O privilege.

19.7 The SEGMENTS Statement

The **SEGMENTS** statement defines the attributes of one or more segments in the application or library on a segment-by-segment basis. The attributes specified by this statement override defaults set in **CODE** and **DATA** statements.

Syntax

SEGMENTS
segmentdefinitions

Remarks

The **SEGMENTS** keyword marks the beginning of the segment definitions. This keyword can be followed by one or more segment definitions, each on a separate line (limited by the number set by the linker's /SEGMENTS option, or 128 if the option is not used). The syntax for each segment definition is as follows:

[[']]*segmentname*[[']][[**CLASS** ' *classname*']][[*attribute*...]]

Each segment definition begins with a *segmentname*, which can be placed in optional single quotation marks ('). The quotation marks are required if *segmentname* conflicts with a module-definition keyword, such as **CODE** or **DATA**.

The **CLASS** keyword specifies the class of the segment. Single quotation marks (') are required around *classname*. If you do not use the **CLASS** argument, the linker assumes that the class is **CODE**.

Each *attribute* must correspond to one of the following attribute fields. Each field can appear at most one time, and order is not significant. The attribute fields are presented below, along with legal values. In each case, the default value is listed last.

Field	Values
load	**PRELOAD, LOADONCALL**
readonly	**READONLY, READWRITE**
executeonly	**EXECUTEONLY, EXECUTEREAD**
iopl	**IOPL, NOIOPL**
conforming	**CONFORMING, NONCONFORMING**
shared	**SHARED, NONSHARED**
movable	**MOVABLE, FIXED**
discard	**NONDISCARDABLE, DISCARDABLE**

The *load* field determines when a segment is to be loaded. This field contains one of the following keywords:

Keyword	Meaning
PRELOAD	The segment is loaded automatically at the beginning of the program
LOADONCALL	The segment is not loaded until accessed (the default)

The *readonly* field determines the access rights to a data segment. This field contains one of the following keywords:

Keyword	Meaning
READONLY	The segment can only be read
READWRITE	The segment can be both read and written to (the default)

The *executeonly* field determines whether a code segment can be read as well as executed. (The attribute has no effect on data segments.) This field contains one of the following keywords:

Keyword	Meaning
EXECUTEONLY	The segment can only be executed
EXECUTEREAD	The segment can be both executed and read (the default)

The *iopl* field determines whether or not a segment has I/O privilege (that is, whether it can access the hardware directly). This field contains one of the following keywords:

Keyword	Meaning
IOPL	The segments have I/O privilege
NOIOPL	The segments do not have I/O privilege (the default)

The *conforming* field specifies whether a code segment is a 286 "conforming" segment. The concept of a conforming segment deals with privilege level (the range of instructions that the process can execute) and is relevant only to users writing device drivers and system-level code. A conforming segment can be called from either Ring 2 or Ring 3, and the segment executes at the caller's privilege level. (The attribute has no effect on data segments.) This field contains one of the following keywords:

Keyword	Meaning
CONFORMING	The segment is conforming
NONCONFORMING	The segment is nonconforming (the default)

The *shared* field determines whether all instances of the program can share a **READWRITE** segment. For code segments and **READONLY** data segments, this field is ignored by OS/2, but is provided for use with real-mode Windows. Under OS/2, all code segments and all **READONLY** data segments are shared. The *shared* field contains one of the following keywords: **SHARED** or **NONSHARED** (the default).

The *movable* field determines whether a segment can be moved around in memory. This field is ignored by OS/2, but is provided for use with real-mode Windows. Under OS/2, all segments are movable. The *movable* field contains one of the following keywords: **MOVABLE** or **FIXED** (the default for Windows).

The optional *discard* field determines whether a segment can be swapped out to disk by the operating system, when not currently needed. This attribute is ignored by OS/2, but is provided for use with real-mode Windows. Under OS/2 systems, all segments can be swapped as needed. The *shared* attribute contains one of the following keywords: **DISCARDABLE** or **NONDISCARDABLE** (the default for Windows).

Example

The following example specifies segments named `cseg1`, `cseg2`, and `dseg`. The first segment is assigned class `mycode` and the second is assigned **CODE** by default. Each segment is given different attributes.

```
SEGMENTS
    cseg1 CLASS 'mycode' IOPL
    cseg2 EXECUTEONLY PRELOAD CONFORMING
    dseg  CLASS 'data' LOADONCALL READONLY
```

19.8 The STACKSIZE Statement

The **STACKSIZE** statement performs the same function as the /STACKSIZE linker option. It overrides the size of any stack segment defined in an application. (The **STACKSIZE** statement overrides the /STACKSIZE option).

Syntax

STACKSIZE *number*

Remarks

The *number* must be an integer; it is considered to be in decimal format by default, but you can use C notation to specify hexadecimal or octal.

Example

The following example allocates 4,096 bytes of local-stack space:

```
STACKSIZE 4096
```

19.9 The EXPORTS Statement

The **EXPORTS** statement defines the names and attributes of the functions exported to other modules and of the functions that run with I/O privilege. The term "export" refers to the process of making a function available to other run-time modules. By default, functions are hidden from other modules at run time.

Syntax

EXPORTS
    exportdefinitions

Remarks

The **EXPORTS** keyword marks the beginning of the export definitions. It may be followed by up to 3,072 export definitions, each on a separate line. You need to give an export definition for each dynamic-link routine you want to make available to other modules. The syntax for an export definition is as follows:

entryname[[=*internalname*]] [[**@***ord*[[**RESIDENTNAME**]]] [[*pwords*]] [[**NODATA**]]

The *entryname* specification defines the function name as it is known to other modules. The optional *internalname* defines the actual name of the export function as it appears within the module itself; by default, this name is the same as *entryname*.

The optional *ord* field defines the function's ordinal position within the module-definition table. If this field is used, the function's entry point can be invoked by name or by ordinal. Use of ordinal positions is faster and may save space.

The optional keyword **RESIDENTNAME** specifies that the function's name be kept resident in memory at all times. This keyword is applicable only if the *ord* option is used because if the *ord* option is not used, OS/2 automatically keeps the names of all exported functions resident in memory anyway.

The *pwords* field specifies the total size of the function's parameters, as measured in words (the total number of bytes divided by two). This field is required only if the function executes with I/O privilege. When a function with I/O privilege is called, OS/2 consults the *pwords* field to determine how many words to copy from the caller's stack to the I/O-privileged function's stack.

The optional keyword **NODATA** is ignored by OS/2, but is provided for use by real-mode Windows.

Normally, the **EXPORTS** statement is only meaningful for functions within dynamic-link libraries and for functions that execute with I/O privilege.

Example

The following **EXPORTS** statement defines the three export functions `SampleRead`, `StringIn`, and `CharTest`. The first two functions can be accessed either by their exported names or by an ordinal number. Note that in the module's own source code, these functions are defined as `read2bin` and `str1`, respectively. The last function runs with I/O privilege and therefore is given with the total size of the parameters: six words.

```
EXPORTS
       SampleRead = read2bin  @8
       StringIn = str1        @4  RESIDENTNAME
       CharTest   6
```

19.10 The IMPORTS Statement

The **IMPORTS** statement defines the names of the functions that will be imported for the application or library. The term "import" refers to the process of declaring that a symbol is defined in another run-time module (a dynamic-link library). Typically, LINK uses an import library (created by the IMPLIB utility) to resolve external references to dynamic-link symbols. However, the **IMPORTS** statement provides an alternative for resolving these references within a module.

Syntax

IMPORTS
    importdefinitions

Remarks

The **IMPORTS** keyword marks the beginning of the import definitions. This keyword is followed by one or more import definitions, each on a separate line. The only limit on the number of import definitions is that the total amount of space required for their names must be less than 64K. Each import definition corresponds to a particular function. The syntax for an import definition is as follows:

[[*internalname*=]]*modulename*.*entry*

The *internalname* specifies the name that the importing module actually uses to call the function. Thus, *internalname* appears in the source code of the importing module, though the function may have a different name in the module where it is defined. By default, *internalname* is the same as the name given in *entry*.

The *modulename* is the name of the application or library that contains the function.

The *entry* field determines the function to be imported and can be a name or an ordinal value. (Ordinal values are set in an **EXPORTS** statement.) If an ordinal value is given, the *internalname* field is required.

NOTE *A given function has a name for each of three different contexts. The function has a name used by the exporting module (where it is defined), a name used as an entry point between modules, and a name as it is used by the importing module (where it is called). If neither module uses the optional internalname field, the function has the same name in all three contexts. If either of the modules use the internalname field, the function may have more than one distinct name.*

Example

The following **IMPORTS** statement defines three functions to be imported: `SampleRead`, `SampleWrite`, and a function that has been assigned an ordinal value of 1. The functions are found in the modules `Sample`, `SampleA`, and `Read`, respectively. The function from the `Read` module is referred to as `ReadChar` in the importing module; the original name of the function, as it is defined in the `Read` module, may or may not be known.

```
IMPORTS
    Sample.SampleRead
    SampleA.SampleWrite
    ReadChar = Read.1
```

19.11 The STUB Statement

The **STUB** statement adds a DOS 3.x executable file to the beginning of the application or library being created. The stub is invoked whenever the module is executed under DOS 2.x or DOS 3.x. Typically, the stub displays a message and terminates execution. (By default, the linker adds its own standard stub for this purpose.)

Syntax

STUB *'filename'*

Remarks

The filename specifies the DOS executable file to be added. If the linker does not find *filename* in the current directory, it searches in the list of directories specified in the PATH environment variable.

Example

The following example appends the DOS executable file STOPIT.EXE to the beginning of the module:

```
STUB 'STOPIT.EXE'
```

The file STOPIT.EXE is executed when you attempt to run the module under DOS.

19.12 The HEAPSIZE Statement

The **HEAPSIZE** statement defines the size of the application's local heap in bytes. This value affects the size of the automatic data segment.

Syntax

HEAPSIZE *bytes* | **MAXVAL**

Remarks

The *bytes* field is an integer number considered decimal by default. However, hexadecimal and octal numbers can be entered by using C notation.

MAXVAL is an optional keyword that may be substituted for *bytes* to set the field parameter. **MAXVAL** sets the heap size to the value of DGROUP–64K. DGROUP is the automatic or default data segment. The effect is that the loader allocates exactly 64K for DGROUP. This may be useful in bound applications in which you want to force a 64K requirement for DGROUP on the program in DOS. The bound program fails to load if 64K minus the size of DGROUP is less than zero.

Examples

```
HEAPSIZE 4000

HEAPSIZE MAXVAL
```

19.13 The PROTMODE Statement

The **PROTMODE** statement specifies that the module runs only in protected mode and not in Windows or dual mode. This statement is always optional, and permits a protected-mode-only application to omit some information from the executable-file header.

Syntax

PROTMODE

Remarks

If this statement is not included in the module-definition file, the linker assumes the application can be run in either real or protected mode.

19.14 The OLD Statement

The **OLD** statement directs the linker to search another dynamic-link module for export ordinals. For more information on ordinals, see the sections above on the **EXPORTS** and **IMPORTS** statements. Exported names in the current module that match exported names in the **OLD** module are assigned ordinal values from that module unless one of the following conditions is in effect: the name in the **OLD** module has no ordinal value assigned, or an ordinal value is explicitly assigned in the current module.

Syntax

OLD *'filename'*

Remarks

This statement is useful for preserving export ordinal values throughout successive versions of a dynamic-link module. The **OLD** has no effect on application modules.

19.15 The REALMODE Statement

The **REALMODE** statement is analogous to the **PROTMODE** statement and is provided for use with real-mode Windows applications.

Syntax

REALMODE

Remarks

REALMODE specifies that the application runs only in real mode. With this statement, the linker relaxes some of the restrictions that it imposes on programs running in protected mode.

19.16 The EXETYPE Statement

The **EXETYPE** statement specifies in which operating system the application (or dynamic-link library) is to run. This statement is optional and provides an additional degree of protection against the program being run in an incorrect operating system.

Syntax

EXETYPE [[OS2 | WINDOWS | DOS4]]

Remarks

The **EXETYPE** keyword must be followed by a descriptor of the operating system, either **OS2** (for OS/2 applications and dynamic-link libraries), **WINDOWS**, or **DOS4**. If no **EXETYPE** statement is given, **EXETYPE OS2** is assumed by an operating system that is loading the program.

The effect of **EXETYPE** is to set bits in the header which identify operating-system type. Operating-system loaders may check these bits.

CHAPTER 20

20 Creating Dual-Mode Programs with BIND

The Microsoft Operating System/2 Bind (BIND) utility converts protected-mode programs so they can run in both real mode and protected mode. Not every protected-mode program can readily be converted. Programs you wish to convert should make no system calls other than calls to the functions listed in the Family API. (The Family API, see the *Microsoft Operating System/2 Programmer's Reference*, is a subset of the API functions.)

NOTE *The BIND utility will not work on BASIC programs.*

The BIND utility must "bind" dynamic-link functions—that is, the utility brings an application program together with libraries and links everything into a single stand-alone file that can run in real mode. The BIND utility also alters the executable-file format of the program so it is recognized as a standard executable file in both real mode and protected mode.

If you are unable to create a bound version of your program, you can build a dual-mode version, as explained at the end of the chapter.

There are three components to the BIND utility:

- BIND. This utility merges the executable file with the appropriate libraries as described above.
- Loader. This tool loads the OS/2 executable file when running DOS 2.x or 3.x and simulates the OS/2 startup conditions in a DOS environment. The loader consists of code that is stored in BIND.EXE and copied into files as needed.
- API.LIB. This library simulates the OS/2 API in a DOS environment.

20.1 Binding Library Routines

The BIND utility replaces Family-API calls with simulator routines from the standard (object-code) library API.LIB. However, your program may also make dynamic-link calls to functions outside the API (that is, you can make dynamic-link calls that are not system calls). This section explains how BIND can accommodate these calls.

If your program makes dynamic-link calls to functions outside the API, use the *linklibs* field described in Section 20.3, "The BIND Command Line." BIND searches each of the libraries and files specified in *linklibs* for object code corresponding to the imported functions. In addition, if you are using import definitions with either the ordinal or the internal-name option, you need to specify import libraries so the functions you call can be identified correctly. (For a discussion of various options within import definitions, see Chapter 19, "Using Module-Definition Files.")

20.2 Binding Functions as Protected Mode Only

If your program freely makes non-Family-API calls without regard to which operating system is in use, the program cannot be converted for use in real mode. However, you may choose to write a program so it first checks the operating system and then restricts system calls (to the Family API) when running in real mode. The BIND utility supports conversion of these programs.

By using the /n command-line option described below you can specify a list of functions supported in protected mode only. If your program ever attempts to call one of these functions when running in real mode, the BadDynLink system function is called and aborts your program. The advantage of this option is that it helps resolve external references, yet it remains the responsibility of your program to check the operating-system version and ensure that not one of these functions is ever called in real mode.

If your program makes calls (either directly or indirectly) to non-Family-API system calls, but you do not use the /n option, then BIND fails to convert your program.

NOTE *BIND Version 1.0 does not work with files linked with the /EXEPACK option.*

20.3 The BIND Command Line

Invoke BIND with the following command line:

BIND *infile* [[*implibs*]] [[*linklibs*]] [[/o *outfile*]] [[/n @*file*]] [[/n *names*]] [[/m *mapfile*]]

The meaning of each command-line field and option is explained below:

The *infile* field contains the name of the OS/2 application. The file name may contain a complete path name. The file extension is optional; if you provide no extension, .EXE is assumed.

The *implibs* field contains the name of one or more import libraries. As explained above, use this field if your program uses an import definition with either the *ordinal* or *internalname* fields.

NOTE *If you want to specify a 64K default data segment when running in real mode, specify the file APILMR.OBJ, which guarantees a 64K stack. The reason this object file may be necessary is that a protected-mode application is not automatically given a 64K default data segment; a protected-mode application is only allocated the space it specifically requests. If you do not specify the file APILMR.OBJ, you may not have the local heap area needed when you run in real mode.*

The *linklibs* field contains the name of one or more standard libraries and object files. Use this field to supply object code needed to resolve dynamic-link calls. Include DOSCALLS.LIB in this field. You must give the complete path name to DOSCALLS.LIB if it is not in the current directory. If you specify a second library, API.LIB, you need to give the complete path name for it also. If you do not specify API.LIB in this field, BIND automatically searches for API.LIB by looking in directories listed in the LIB environment variable. For example, the following command line successfully uses BIND if API.LIB is located in a directory listed in the LIB environment variable:

```
BIND FOO.EXE \LIB\DOSCALLS.LIB
```

The /o option specifies the name for the bound application, *outfile*, and may contain a full path name. The default value for this field is the name of the file that was given as *infile*.

The /n option provides a way of listing functions that are supported in protected mode only. As explained above, if any of these functions are ever called in real

mode, the BadDynLink function is called to abort the program. The /n option can be used either with a list of one or more *names* (separated by spaces), or Module Statements Attribute Defined with a *file* specification preceded by the @ sign. The specified file should consist of a list of functions, one per line.

The /m option causes a link map to be generated for the DOS 3.x environment of the .EXE file. The *mapfile* is the destination of the link map. If *mapfile* is not specified with the /m option, the destination of the link map is standard output.

20.4 BIND Operation

BIND produces a single executable file that can run in either real mode or protected mode. To complete this task, BIND executes three major steps:

1. Reads in the dynamic-link entry points (for imported functions) from the OS/2 executable file and outputs to a temporary object file the EXTDEF object records for each imported item. Each EXTDEF record tells the linker of an external reference that needs to be resolved through ordinary linking.
2. Invokes LINK, giving the executable file, the temporary object file, the API.LIB file, and any other libraries specified on the BIND command line. By linking in the loader and the API-simulator routines, LINK produces an executable file that can run in real mode.
3. Merges the protected-mode and real-mode executable files to produce a single file that can run in either mode.

NOTE *A dual-mode executable file produced with BIND can be run in both DOS 2.x and 3.x environments. However, if you change the name of an executable file produced by BIND, then it will not run under DOS 2.x.*

20.5 Executable-File Layout

OS/2 executable files have two headers. The first header has a DOS 3.x format. The second header has the OS/2 format. When the executable file is run on an OS/2 system, it ignores the first header and uses the OS/2 format. When run under DOS 3.x, the old header is used to load the file. Figure 20.1 shows the arrangement of the merged headers.

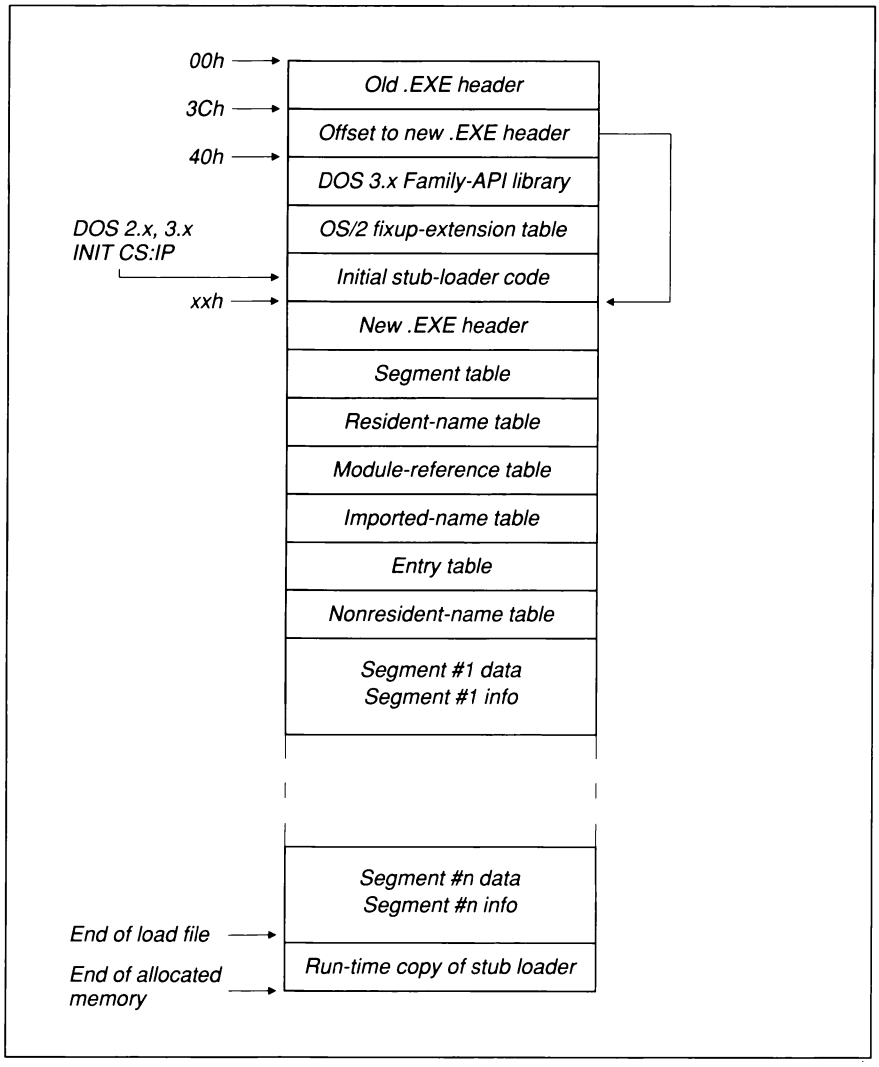


Figure 20.1 OS/2 Executable-File Header

20.6 How to Build a Dual-Mode Application

If you cannot create a bound application out of your program, you may want to create a dual-mode executable (.EXE) file instead. A dual-mode .EXE file is similar to a bound file; both types of files run in either real mode or protected mode. However, the two types of files have a different internal structure.

A bound file has common executable code that actually runs in both modes. System calls are specific to real mode or protected mode, but all system calls are modified at load time.

In contrast, a dual-mode file has two separate programs contained in one file. One of these programs is real-mode-only and the other is protected-mode-only. All the code in a dual-mode executable file runs in either one mode or the other.

To create a dual-mode application:

1. Link a real-mode version of your program.
2. Create a module-definition file that contains the statement

   ```
   STUB 'PROGR.EXE'
   ```

   in which you substitute the name of your real-mode program for PROGR.EXE.
3. Link the protected-mode version of your program with this module-definition file. The protected-mode version and the real-mode version should have different names.

CHAPTER 21
Using EXEHDR

The Microsoft Segmented-EXE Header Utility (EXEHDR) displays the contents of an executable-file header. You can use EXEHDR with OS/2 or Windows, and you can use it with an application or dynamic-link library. (With a Windows file, some of the meanings of the executable-file-header fields change; see your Windows documentation for more information.) The principal uses of EXEHDR include the following:

- Determining whether a file is an application or a dynamic-link library
- Viewing the attributes set by the module-definition file
- Viewing the number and size of code and data segments

Except where noted otherwise, the special terms and keywords mentioned in this section are explained in Chapter 19, "Using Module-Definition Files."

21.1 The EXEHDR Command Line

To invoke the EXEHDR utility, use the syntax

EXEHDR [[/v]] *file*

in which *file* is an application or dynamic-link library for either the OS/2 or Windows environment. The /v option specifies output in verbose mode.

Section 21.2 presents sample output and explains the meaning of each field of the output. Section 21.3 describes additional output that EXEHDR produces when it is run in verbose mode.

21.2 EXEHDR Output

This section discusses the meaning of each field in the output below—output produced when `EXEHDR LINK.EXE` is specified on the OS/2 command line. The first six fields list the contents of the segmented-executable-file header. The rest of the output lists each physical segment in the file. (The term "physical segment" is defined in Chapter 14, "Incremental Linking with ILINK.")

```
Module:                     LINK
Description:                Microsoft Segmented-Executable
Linker
Data:                       NONSHARED
Initial CS:IP:              seg   2 offset 3d9c
Initial SS:SP:              seg   4 offset 8e40
DGROUP:                     seg   4

no. type address  file  mem   flags
  1 CODE 00003400 0f208 0f208
  2 CODE 00012e00 05b04 05b04
  3 DATA 00018c00 01c1f 01c1f
  4 DATA 0001aa00 01b10 08e40
```

The `Module` field is the name of the application as specified in the **NAME** statement of the module-definition file. If no module definition was used to create the executable file, this field displays the name assumed by default. If a module definition was used to create the file, but the **LIBRARY** statement appeared instead of the **NAME** statement (thus specifying a dynamic-link library), the name of the library is given and EXEHDR uses the word `Library` instead of `Module`.

The `Description` field gives the contents, if any, of the **DESCRIPTION** statement of the module-definition file used to create the file being examined.

The `Data` field indicates the type of the program's automatic data segment: **SHARED**, **NONSHARED**, or **NONE**. This type can be specified in a module-definition file, but by default is **NONSHARED** for applications and **SHARED** for dynamic-link libraries. In this context, **SHARED** and **NONSHARED** are equivalent to the **SINGLE** and **MULTIPLE** attributes listed in Section 19.6. (The "automatic data segment" is the physical segment corresponding to the group named DGROUP.)

The `Initial CS:IP` field is the program starting address (if an application is being examined) or address of the initialization routine (if a dynamic-link library is being examined).

The `Initial SS:SP` field gives the value of the initial stack pointer.

The `DGROUP` field is the segment number of the automatic data segment. This segment corresponds to the group named DGROUP in the program's object modules. Note that segment numbers start with the number 1.

After the contents of the OS/2 executable header are displayed, the contents of the segment table are listed. The following list describes the meaning of each heading in the table. Note that all values are given in hexadecimal radix except for the segment index number.

Heading	Meaning
`no.`	Segment index number, starting with 1, in decimal radix.
`type`	Identification of the segment as a code or data segment. A code segment is any segment with class name ending in `'CODE'`. All other segments are data segments.
`address`	Location, within the file, of the contents of the segment.
`file`	Size in bytes of the segment, as contained in the file.
`mem`	Size in bytes of the segment as it will be stored in memory. If the value of this field is greater than the value of the `file` field, OS/2 pads the additional space with zero values at load time.
`flags`	Segment attributes, as defined in Chapter 19, "Using Module-Definition Files." If the /v option is not used, only nondefault attributes are listed. Attributes are given in the form specified in Chapter 19: **READWRITE**, **PRELOAD**, and so forth. Attributes that are meaningful to Windows but not to OS/2 are displayed as lowercase and in parentheses, [e.g., `(movable)`].

21.3 Output in Verbose Mode

When you specify the /v mode, the EXEHDR utility gives all the information discussed in Section 21.2, as well as some additional information. Specifically, when running in verbose mode, EXEHDR displays the following information in this order:

1. DOS 3.x header information. All OS/2 executable files have a DOS 3.x header, whether bound or not. If the program is not bound, the DOS 3.x portion typically consists of a stub that terminates the program. For a description of DOS executable-file-header fields, see the *Microsoft MS-DOS Programmer's Reference*, or see Chapter 17 in this manual, "Using Other Utilities," for information on the Microsoft EXE File Header Utility (EXEMOD).

2. OS/2-specific header fields. This output includes the fields described in Section 21.2 except for the segment table. (The segment-table display for verbose mode is described below.)

3. File addresses and lengths of the various tables in the executable file, as in the following example:

```
Resource Table:             00003273 length 0000 (0)
Resident Names Table:       00003273 length 0008 (8)
Module Reference Table:     0000327b length 0006 (6)
Imported Names Table:       00003281 length 0033 (51)
Entry Table:                000032b4 length 0002 (2)
Non-resident Names Table:   000032b6 length 0029 (41)
Movable entry points:       0
Segment sector size:        512
```

   The first field in each row gives the name of the table, the second field gives the address of the table within the file, the third field gives the length of the table in hexadecimal radix, and the last field gives the length of the table in decimal radix. See the *Microsoft Operating System/2 Programmer's Reference* for an explanation of each table.

4. Segment table with complete attributes, not just the nondefault attributes. In addition to the attributes described in Section 21.2, verbose mode also displays these two additional attributes:

   - The `relocs` attribute is displayed for each segment that has address relocations. Relocations occur in each segment that references objects in other segments or makes dynamic-link references.
   - The `iterated` attribute is displayed for each segment that has iterated data. Iterated data consist of a special code that packs repeated bytes; for example, OS/2 executables produced with the /EXEPACK option of LINK have iterated data.

5. Run-time relocations and fixups. See the object-module information in the *Microsoft Operating System/2 Programmer's Reference* for more information.

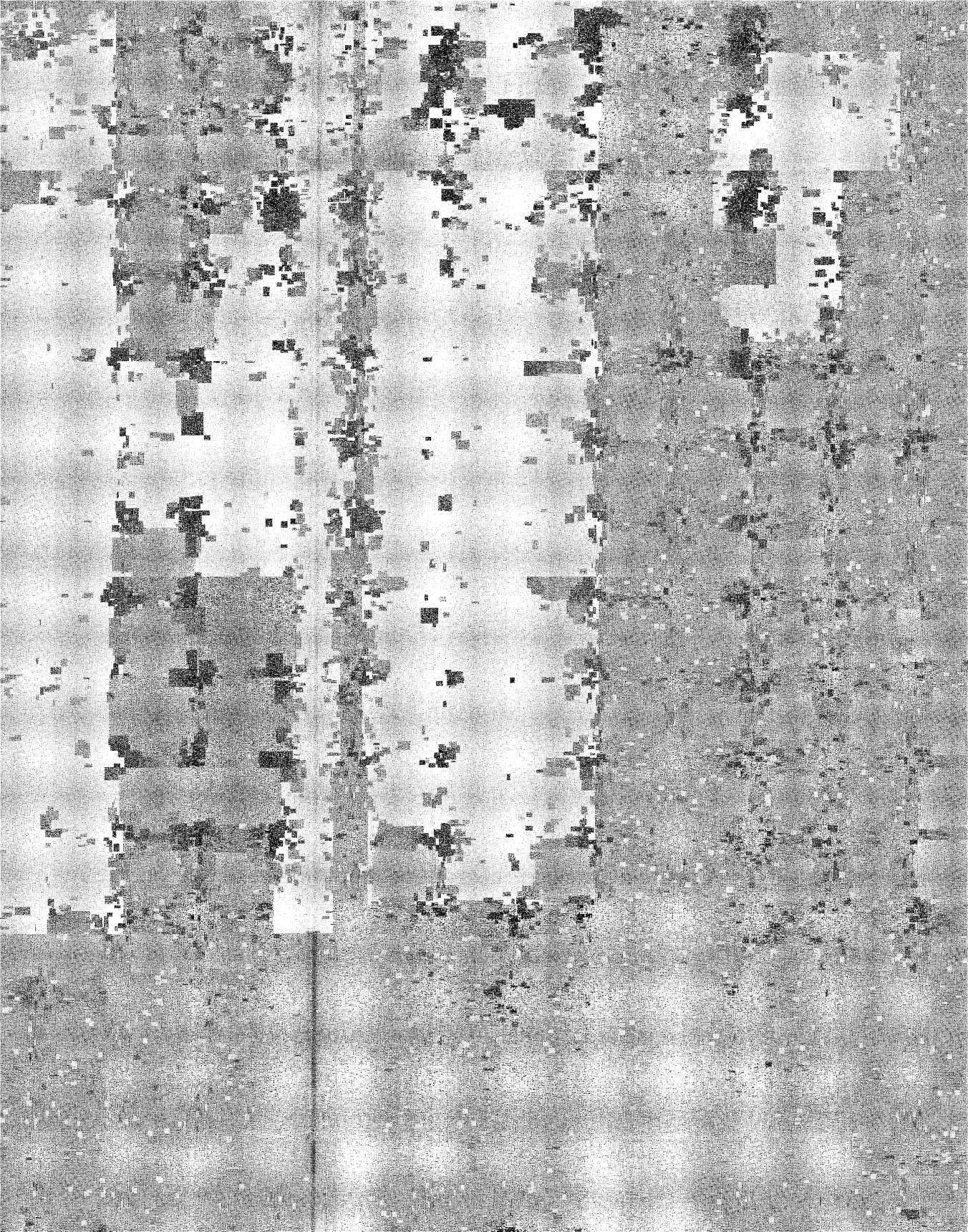
6. Finally, EXEHDR lists all exported entry points. Exports are discussed in Chapter 18, "Linking for Windows and OS/2 Systems," and in Section 19.9, "The EXPORTS Statement."

Appendixes

A Regular Expressions 353

B Exit Codes 357

C Error Messages 361



Appendix A

Regular Expressions

Regular expressions are used to specify text patterns in searches for variable text strings. Special characters can be used within regular expressions to specify groups of characters to be searched for.

This appendix explains all of the special characters you can use to form regular expressions, but you do not need to learn them all to use the CodeView Search commands. The simplest form of regular expression is simply a text string. For example, if you want to search for all instances of the symbol `COUNT`, you can specify `COUNT` as the string to be found.

If you want to search only for simple strings, you do not need to read this entire appendix, but you should know how to search for strings containing the special characters used in regular expressions. See Section A.2 for more information.

A.1 Special Characters in Regular Expressions

The following characters have special meanings in regular expressions:

Character	Purpose
Asterisk (*)	Matches any number of repetitions of the previous character.
Backslash (\)	Removes the special characteristics of the following characters: backslash (\), period (.), caret (^), dollar sign ($), asterisk (*), and left bracket ([).
Brackets ([ ])	Matches characters specified within the brackets. The following special characters may be used:

Character	Purpose
Caret (^)	Reverses the function of the brackets. That is, the caret matches any character except those specified within the brackets.
Dash (–)	Matches characters in ASCII order between (inclusive of) the characters on either side of the dash.
Caret (^)	Matches beginning of line.
Dollar sign ($)	Matches end of line.
Period (.)	Matches any character.

A.2 Searching for Special Characters

If you need to match one of the special characters used in regular expressions, precede it with a backslash when you specify a search string. The special characters are the asterisk (*), backslash (\), left bracket ([), caret (^), dollar sign ($), and period (.).

For example, the regular expression `I*J` matches such combinations as `J`, `IJ`, `IIJ`, and `IIIJ`. The regular expression `I\*J` matches only `I*J`. The backslash is necessary because the asterisk (*) is a special character in regular expressions.

A.3 Using the Period

A period in a regular expression matches any single character. This corresponds to the question mark (?) used in specifying DOS file names.

For example, you could use the regular expression `AMAX.` to search for either of the intrinsic functions `AMAX0` and `AMAX1`. You could use the expression `X.Y` to search for strings such as `X+Y`, `X-Y`, or `X*Y`. If your programming style is to put a space between variables and operators, you could use the regular expression `X.Y` for the same purpose.

Note that when you use the period as a wild card, you will find the strings you are looking for, but you may also find other strings that you aren't interested in. You can use brackets to be more exact about the strings you want to find.

A.4 Using Brackets

You can use brackets to specify a character or characters you want to match. Any of the characters listed within the brackets is an acceptable match. This method is more exact than using a period to match any character.

For example, the regular expression `x[-+/*]y` matches `x+y`, `x-y`, `x/y`, or `x*y`, but not `x=y` or `xzy`. The regular expression `COUNT[12]` matches `COUNT1` and `COUNT2`, but not `COUNT3`.

Most regular-expression special characters have no special meaning when used within brackets. The only special characters within brackets are the caret (^), dash (–), and right bracket (]). Even these characters only have special meanings in certain contexts, as explained in Sections A.4.1–A.4.3.

A.4.1 Using the Dash within Brackets

The dash (minus sign) can be used within brackets to specify a group of sequential ASCII characters. For example, the regular expression `[0-9]` matches any digit; it is equivalent to `[0123456789]`. Similarly, `[a-z]` matches any lowercase letter, and `[A-Z]` matches any uppercase letter.

You can combine ASCII ranges of characters with other listed characters. For example, `[A-Za-z ]` matches any uppercase or lowercase letter or a space.

The dash has this special meaning only if you use it to separate two ASCII characters. It has no special meaning if used directly after the starting bracket or directly before the ending bracket. This means you must be careful where you place the dash (minus sign) within brackets.

For example, you might use the regular expression `[+-/*]` to match the characters `+`, `-`, `/`, and `*`. However, this does not give the intended result. Instead it matches the characters between `+` and `/` and also the character `*`. To specify the intended characters, put the dash first or last in the list: `[-+/*]` or `[+/*-]`.

A.4.2 Using the Caret within Brackets

If used as the first character within brackets, the caret (`^`) reverses the meaning of the brackets. That is, any character except the ones in brackets is matched. For example, the regular expression `[^0-9]` matches any character that is not a digit. Specifying the characters to be excluded is often more concise than specifying the characters you want to match.

If the caret is not in the first position within the brackets, it is treated as an ordinary character. For example, the expression `[0-9^]` matches any digit or a caret.

A.4.3 Matching Brackets within Brackets

Sometimes you may want to specify the bracket characters as characters to be matched. This is no problem with the left bracket; it is treated as a normal character. However, the right bracket is interpreted as the end of the character list rather than as a character to be matched.

If you want the right bracket to be matched, you must make it the first character after the initial left bracket. For example, the regular expression `[]#![@%]` matches either bracket character or any of the other characters listed within the brackets. However, if you changed the order of just one of the characters (to `[#]![@%]`), the meaning would be changed so that you would be specifying two groups of characters in brackets: `[#]` and `[@%]`.

A.5 Using the Asterisk

The asterisk matches zero or more occurrences of the character preceding the asterisk.

For example, the regular expression `IF * TEST` matches any number of repetitions of the space character that follow the word "`if`."

```
IF TEST
IF        TEST
IF TEST
IFTEST
```

Notice that the last example contains zero repetitions of the space character.

The asterisk is convenient if the text you are searching for might contain some spaces, but you don't know the exact number. (Be careful in this situation: you can't be sure if the text contains a series of spaces or a tab.)

You might also use the asterisk to search for a symbol when you aren't sure of the spelling. For example, you could use `first*ime` if you aren't sure if the identifier you are searching for is spelled `firsttime` or `firstime`.

One particularly powerful use of the asterisk is to combine it with the period (.*). This combination searches for any group of characters. It is similar to the asterisk used in specifying DOS file names. For example, the expression `(.*)` matches `(test)`, `(response .EQ. 'Y')`, `(x=0;x .LE. 20;x=x+1)`, or any other string that starts with a left parenthesis and ends with a right parenthesis.

You can use brackets with the asterisk to search for a sequence of repeated characters of a given type. For example, `\[[0-9]*]` matches number strings within brackets (`[1353]` or `[3]`), but does not match character strings within brackets (`[count]`). Empty brackets (`[ ]`) are also matched since the characters in the brackets are repeated zero times.

A.6 Matching the Start or End of a Line

In regular expressions, the caret (^) matches the start of a line, and the dollar sign ($) matches the end of a line.

For example, the regular expression `^C` matches any uppercase C that starts a line. Similarly, `)$` matches a right parenthesis at the end of a line, but not a right parenthesis within a line.

You can combine both symbols to search for entire lines. For example, `^{$` matches any line consisting of only a left curly brace in the left margin and `^$` matches blank lines.

Appendix B

Using Exit Codes

Most utilities return an exit code (sometimes called an "errorlevel" code) used by DOS batch files or other programs. If the program finishes without errors, it returns an exit code 0. The code varies if the program encounters an error.

B.1 Exit Codes with NMAKE

The NMAKE stops execution if a program executed by one of the commands in the NMAKE description file encounters an error. (Invoke NMAKE with the /I option to disable this behavior for the entire description file, or place a minus sign (–) in front of a command to disable it only for that command.) The exit code returned by the program is displayed as part of the error message.

Assume the NMAKE description file `TEST` contains the following lines:

```
TEST.OBJ :      TEST.FOR
      FL /c TEST.FOR
```

If the source code in `TEST.FOR` contains a program error (but not if it contains a warning error), you would see the following message the first time you use NMAKE with the NMAKE description file `TEST`:

```
"nmake: CL /c TEST.FOR - error 2"
```

This error message indicates that the command `CL /c TEST.FOR` in the NMAKE description file returned exit code 2.

You can also test exit codes in NMAKE description files with the **!IF** directive.

B.2 Exit Codes with DOS Batch Files

If you prefer to use DOS batch files instead of NMAKE description files, you can test the code returned with the IF command. The following sample batch file, called `COMPILE.BAT`, illustrates how to do this:

```
CL /c %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.C
```

DOS then executes the first line of the batch file, substituting `TEST.C` for the parameter `%1`, as in the command line below.

```
CL /c TEST.C
```

It returns an exit code 0 if the compilation is successful or a higher code if the compiler encounters an error. In the second line, 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), DOS executes the following command:

```
LINK TEST;
```

LINK also returns a code that is tested by the third line.

B.3 Exit Codes for Programs

An exit code 0 always indicates execution of the program with no fatal errors. Warning errors also return exit code 0. NMAKE can return several codes indicating different kinds of errors; other programs return only one code. The exit codes for each program are listed in Sections B.3.1–B.3.4.

B.3.1 LINK Exit Codes

Code	Meaning
0	No error.
2	Program error. Commands or files given as input to the linker produced the error.
4	System error. The linker encountered one of the following problems: 1) ran out of space on output files; 2) was unable to reopen the temporary file; 3) experienced an internal error; 4) was interrupted by the user.

B.3.2 LIB Exit Codes

Code	Meaning
0	No error.
2	Program error. Commands or files given as input to the utility produced the error.
4	System error. The library manager encountered one of the following problems: 1) ran out of memory; 2) experienced an internal error; 3) was interrupted by the user.

B.3.3 NMAKE Exit Codes

Code	Meaning
0	No error
2	Program error
4	System error—out of memory

If a program called by a command in the NMAKE description file produces an error, the exit code is displayed in the NMAKE error message.

B.3.4 EXEMOD and SETENV Exit Codes

Code	Meaning
0	No error.
2	Program error. Commands or files given as input to the utility produced the error.
4	System error. The utility encountered one of the following problems: 1) ran out of memory; 2) experienced an internal error; 3) was interrupted by the user.

B.3.5 CVPACK Exit Codes

Code	Meaning
0	No error.
1	Program error. Commands or files given as input to the utility produced the error.

Appendix C

Error Messages

C.1 CodeView Error Messages

The CodeView debugger displays an error message whenever it detects a command it cannot execute. Most errors (start-up errors are the exception) terminate the CodeView command under which the error occurred but do not terminate the debugger. You may see any of the following messages.

? cannot display

The Display Expression command (**?**) has been passed a valid symbol it cannot display. A variable with enumeration type cannot be displayed.

Argument to IMAG/DIMAG must be simple type

You specified an argument to an **IMAG** or **DIMAG** function not permitted, such as an array with no subscripts.

Array must have subscript

You specified an array without any subscripts in an expression, such as `IARRAY+2`. A correct example would be `IARRAY[1]+2`.

Bad address

You specified an address in an invalid form.

For instance, you may have entered an address containing hexadecimal characters when the radix is decimal.

Bad breakpoint command

You typed an invalid breakpoint number with the Breakpoint Clear, Breakpoint Disable, or Breakpoint Enable command.

The number must be in the range 0 to 19.

Bad emulator info

The CodeView debugger cannot read data from the floating-point emulator.

Bad flag

You specified an invalid flag mnemonic with the Register dialog command (**R**).

Use one of the mnemonics displayed when you enter the command **RF**.

Bad format string

You used an invalid format specifier following an expression.

Expressions used with the Display Expression, Watch, Watchpoint, and Tracepoint commands can have CodeView format specifiers set off from the expression by a comma. The valid format specifiers are **c**, **d**, **e**, **E**, **f**, **g**, **G**, **i**, **o**, **s**, **u**, **x**, **X**. Some format specifiers can be preceded by the prefix **h** or **l**. See the Display Expression command for more information about format specifiers.

Bad integer or real constant

You specified an illegal numeric constant in an expression.

Bad intrinsic function

You specified an illegal intrinsic function name in an expression.

Bad radix (use 8, 10, or 16)

With the **N** command, you can use only octal, decimal, and hexadecimal radixes.

Bad register

You typed the Register command (**R**) with an invalid register name.

Use AX, BX, CX, DX, SP, BP, SI, DI, DS, ES, SS, CS, IP, or F.

Bad subscript

You entered an illegal subscript expression for an array, such as `IARRAY(3.3)` or `IARRAY((3,3))`. The correct expression for this example (in BASIC or FORTRAN) is `IARRAY(3,3)`.

Bad type case

The types of the operands in an expression are incompatible.

Bad type (use one of 'ABDILSTUW')

The valid dump types are ASCII (**A**), Byte (**B**), Double Word (**D**), Integer (**I**), Long Real (**L**), Short Real (**S**), 10-Byte Real (**T**), Unsigned (**U**), and Word (**W**).

Badly formed type

The type information in the symbol table of the file you are debugging is incorrect.

If the message occurs, please note the circumstances of the error and inform Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

Breakpoint # or '*' expected

You entered the Breakpoint Clear (**BC**), Breakpoint Disable (**BD**), or Breakpoint Enable (**BE**) command with no argument.

These commands require that you specify the number of the breakpoint to be acted on, or that you specify the asterisk (*) indicating that all breakpoints are to be acted on.

Cannot use struct or union as scalar

A struct or union variable cannot be used as a scalar value in a C expression.

Such variables must be followed by a file separator or preceded by the address of operator.

Cannot cast complex constant component into REAL

Both the real and imaginary components of a **COMPLEX** constant must be compatible with the type **REAL**.

Cannot cast IMAG/DIMAG argument to COMPLEX

Arguments to **IMAG** and **DIMAG** must be simple numeric types.

Cannot find *filename*

The CodeView debugger could not find the executable file you specified when you started.

You may have misspelled the file name, or the file is in a different directory.

Character constant too long

You specified a character constant too long for the FORTRAN expression evaluator (the limit is 126 bytes).

Character too big for current radix

In a constant, you specified a radix that is larger than the current CodeView radix.

Use the **N** command to change the radix.

Constant too big

The CodeView debugger cannot accept an unsigned constant number larger than 4,294,967,295 (16#FFFFFFFF).

CPU not an 80386

The 386 option cannot be selected if you are using a machine without an 80386 processor.

Divide by zero

An expression in an argument of a dialog command attempts to divide by zero.

EMM error

The debugger is failing to use EMM correctly. Please contact Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

EMM hardware error

The Expanded Memory routines report a hardware error. Your expanded memory board may need replacement.

EMM memory not found

You tried to use the /E option without having installed expanded memory. You must make this installation with software that accesses the memory according to the Lotus/Intel/Microsoft EMS specification.

EMM software error

The Expanded Memory routines report a software error. Reinstall EMM software.

Expression not a memory address

A Tracepoint command was given without a symbol that evaluates to a single memory address.

For example, the commands `TP?1` and `TP?a+b` each produce this error message. The proper way to put a tracepoint on the word at address 1 is with the command `TPW 1`.

Expression too complex

An expression given as a dialog-command argument is too complex.

Try simplifying the expression.

Extra input ignored

You specified too many arguments to a command.

The CodeView debugger evaluates the valid arguments and ignores the rest. Often in this situation the debugger does not evaluate the arguments in the way you intended.

Flip/Swap option off - application output lost

The program you are debugging is writing to the screen, but the output cannot be displayed because you have turned off the flip/swap option.

Floating point error

This message should not occur, but if it does, please note the circumstances of the error and inform Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

Floating point not loaded

This message occurs when the current thread has not initialized its own emulator. Each thread has its own floating-point emulator.

Function call before 'main'

This message occurs when you attempt to evaluate a program-defined function before you have entered the main function.

Execute at least to the beginning of the main function before attempting to evaluate program-defined functions.

Illegal instruction

This message usually indicates that a divide-by-zero machine instruction was attempted.

Index out of bound

You specified a subscript value outside the bounds declared for the array.

Insufficient EMM memory

Not enough expanded memory is available to hold the program's symbol table.

Internal debugger error

If this message occurs, please note the circumstances of the error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

Invalid argument

One of the arguments you specified is not a valid CodeView expression.

Invalid executable file format - please relink

The executable file was not linked with the version of the linker released with this version of the CodeView debugger. Relink with the more current version of the linker.

Invalid option

The option specified cannot be used with the CodeView Option command.

Missing '"'

You specified a string as an argument to a dialog command, but you did not supply a closing double quotation mark.

Missing '('

An argument to a dialog command was specified as an expression containing a right parenthesis, but no left parenthesis.

Missing ')'

An argument to a dialog command was specified as an expression containing a left parenthesis, but no right parenthesis.

Missing ']'

An argument to a dialog command was specified as an expression containing a left bracket, but no right bracket.

This error message can also occur if a regular expression is specified with a right bracket but no left bracket.

Missing '(' in complex constant

The debugger is expecting an opening parenthesis of a complex constant in an expression, but it is missing.

Missing ')' in complex constant

The debugger expects a closing parenthesis of a complex constant in an expression.

Missing ')' in substring

The debugger expects a closing parenthesis of a substring expression.

Missing '(' to intrinsic

The debugger expects an opening parenthesis for an intrinsic function.

Missing ')' to intrinsic

The debugger expects a closing parenthesis for an intrinsic function.

No closing single quote

You specified a character in an expression used as a dialog-command argument, but the closing single quotation mark is missing.

No code at this line number

You tried to set a breakpoint on a source line that does not correspond to machine code. (In other words, the source line does not contain an executable statement.)

For instance, the line may be a data declaration or a comment.

No free EMM memory handles

The debugger cannot find an available handle. EMM software allocates a fixed number of memory handles (usually 256) to be used for specific tasks.

No match of regular expression

No match was found for the regular expression you specified with the Search command or with the Find selection from the Search menu.

No previous regular expression

You selected Previous from the Search menu, but there was no previous match for the last regular expression specified.

No source lines at this address

The address you specified as an argument for the View command (**V**) does not have any source lines.

For instance, it could be an address in a library routine or an assembly-language module.

No such file/directory

A file specified in a command argument or in response to a prompt does not exist.

For instance, the message appears when you select Load from the File menu and then enter the name of a nonexistent file.

No symbolic information = CV options not used or wrong LINK version

The program file you specified is not in the CodeView format.

You cannot debug in source mode unless you recreate the file in CodeView format. However, you can debug in assembly mode.

Not a text file

You attempted to load a file by using the Load selection from the File menu or using the View command, but the file is not a text file.

The CodeView debugger determines if a file is a text file by checking the first 128 bytes for characters that are not in the ASCII ranges 9 to 13 and 20 to 126.

Not an executable file

The file you specified to be debugged when you started the CodeView debugger is not an executable file having the extension .EXE or .COM.

Not enough space

You typed the Shell Escape command (!) or selected Shell from the File menu, but there is not enough free memory to execute COMMAND.COM.

Since memory is released by code in the FORTRAN start-up routines, this error always occurs if you try to use the Shell Escape command before you have executed any code. Use any of the code-execution commands (Trace, Program Step, or Go) to execute the FORTRAN start-up code, then try the Shell Escape command again. The message also occurs with assembly-language programs that do not specifically release memory.

Object too big

You entered a Tracepoint command with a data object (such as an array) larger than 128 bytes.

Operand types incorrect for this operation

An operand in a FORTRAN expression had a type incompatible with the operation applied to it.

For example, if `P` is declared as `CHARACTER P(10)`, the `? P+5` would produce this error, since a character array cannot be an operand of an arithmetic operator.

Operator must have a struct\union type

You have used one of the C member-selection operators (–, >, or .) in an expression that does not reference an element of a structure or union.

Operator needs lvalue

You specified an expression that does not evaluate to a memory location in an operation that requires one. (An lvalue is an expression that refers to a memory location.)

For example, `buffer(count)` is correct because it represents a symbol in memory. However, `I .EQV. 10` is invalid because it evaluates to TRUE or FALSE instead of to a single memory location.

Overlay not resident

You tried to unassemble machine code from a function that is currently not in memory.

Program terminated normally (*number*)

You executed your program to the end. The number displayed in parentheses is the exit code returned to DOS by your program.

You must use the Restart command (or the Start menu selection) to start the program before executing more code.

Radix must be between 2 and 36 inclusive

You specified a radix outside the allowable range.

Register variable out of scope

You tried to specify a register variable by using the period (.) operator and a routine name.

For example, if you are in a third-level routine, you can display the value of a local variable called `local` in a second-level routine called `parent` with the following command:

```
? parent.local
```

However, this command will not work if `local` is declared as a register variable.

Regular expression too complex

The regular expression specified is too complex for the CodeView debugger to evaluate.

Regular expression too long

The regular expression specified is too long for the CodeView debugger to evaluate.

Restart program to debug

You have executed to the end of the program you are debugging.

Simple variable cannot have argument

In an expression, you specified an argument to a simple variable.

For example, given the declaration `INTEGER NUM`, the expression `NUM(I)` is not allowed.

Substring range out of bound

A character expression exceeds the length specified in the **CHARACTER** statement.

Syntax error

You specified an invalid command line for a dialog command.

Check for an invalid command letter. This message also appears if you enter an invalid assembly-language instruction using the Assembly command. The error is preceded by a caret that points to the first character the CodeView debugger could not interpret.

Too few array bounds given

The bounds that you specified in an array subscript do not match the array declaration.

For example, given the array declaration `INTEGER IARRAY(3,4)`, the expression `IARRAY(I)` would produce this error message.

Too many array bounds given

The bounds that you specified in an array subscript do not match the array declaration.

For example, given the array declaration `INTEGER IARRAY(3,4)`, the expression `IARRAY(I,3,J)` would produce this error message.

Too many breakpoints

You tried to specify a 21st breakpoint; the CodeView debugger permits only 20.

Too many open files

You do not have enough file handles for the CodeView debugger to operate correctly.

You must specify more files in your CONFIG.SYS file. See the DOS user's guide for information on using the CONFIG.SYS file.

Type clash in function argument

The type of an actual parameter does not match the corresponding formal parameter.

This message also appears when a subroutine that uses alternate returns is called and the values of the return labels in the actual parameter list are not 0.

Type conversion too complex

You tried to type cast an element of an expression in a type other than the simple types or with more than one level of indirection.

An example of a complex type would be type casting to a struct or union type. An example of two levels of indirection is `char **`.

Unable to open file

A file you specified in a command argument or in response to a prompt cannot be opened.

For instance, this message appears when you select Load from the File menu, and then enter the name of a file that is corrupted or has its file attributes set so that it cannot be opened.

Unknown symbol

You specified an identifier not in the CodeView debugger's symbol table.

Check for a misspelling. This message may also occur if you try to use a local variable in an argument when you are not in the routine where the variable is defined. The message also occurs when a subroutine that uses alternate returns is called and the values of the return labels in the actual parameter list are not 0.

Unrecognized option *option*
Valid options: /B /C*command* **/D /F /I /M /S /T /W /43 /2**

You entered an invalid option when starting the CodeView debugger.

Try retyping the command line.

Usage cv [*options*] **file** [*arguments*]

You failed to specify an executable file when you started CodeView.

Try again with the syntax shown in the message.

Video mode changed without the /S option

The program changed video modes (either to or from graphics modes) when screen swapping was not specified.

You must use the /S option to specify screen swapping when debugging graphics programs. You can continue debugging when you get this message, but the output screen of the debugged program may be damaged.

Warning: packed file

You started the CodeView debugger with a packed file as the executable file.

You can attempt to debug the program in assembly mode, but the packing routines at the start of the program may make this difficult. You cannot debug in source mode because all symbolic information is stripped from a file when it is packed with the /EXEPACK linker option.

Wrong number of function arguments

You specified an incorrect number of arguments when you tried to evaluate a function in a CodeView expression.

C.2 LINK Error Messages

This section lists and describes error messages generated by the Microsoft Segmented-Executable Linker, LINK.

Fatal errors cause the linker to stop execution. Fatal error messages have the following format:

location : error L1*xxx*: *messagetext*

Nonfatal errors indicate problems in the executable file. LINK produces the executable file. Nonfatal error messages have the following format:

location : error L2*xxx*: *messagetext*

Warnings indicate possible problems in the executable file. LINK produces the executable file. Warnings have the following format:

location : warning L4*xxx*: *messagetext*

In all three kinds of messages, location is the input file associated with the error, or LINK if there is no input file. If the input file is an .OBJ or .LIB file and has a module name, the module name is enclosed in parentheses, as shown below.

```
SLIBC.LIB(_file)
MAIN.OBJ(main.c)
TEXT.OBJ
```

The following error messages may appear when you link object files with the Microsoft Segmented-Executable Linker, LINK.

C.2.1 LINK Fatal Error Messages

Number	LINK Error Message
L1001	*option* **: option name ambiguous**

A unique option name did not appear after the option indicator (/). For example, the command

```
LINK /N main;
```

generates this error, since LINK cannot tell which of the options beginning with the letter "N" was intended.

L1002 *option* **: unrecognized option name**

An unrecognized character followed the option indicator (/), as shown below.

```
LINK /ABCDEF main;
```

Number	LINK Error Message
L1003	**/QUICKLIB, /EXEPACK incompatible**
	You cannot link with both the /QU option and the /E option.
L1004	*option* **: invalid numeric value**
	An incorrect value appeared for one of the linker options. For example, a character string was given for an option that requires a numeric value.
L1005	**/PACKCODE : packing limit exceeds 65536 bytes**
	The value supplied with the /PACKCODE option exceeds the limit of 65,536 bytes.
L1006	*option-text* **: stack size exceeds 65535 bytes**
	The value given as a parameter to the /STACKSIZE option exceeds the maximum allowed.
L1007	*option* **: interrupt number exceeds 255**
	A number greater than 255 was given as the /OVERLAYINTERRUPT option value.
L1008	*option* **: segment limit set too high**
	The /SEGMENTS option specified a limit on the number of segments allowed greater than 3,072.
L1009	*number* **: CPARMAXALLOC : illegal value**
	The number specified in the /CPARMAXALLOC option was not in the range 1– 65,535.
L1020	**no object modules specified**
	No object-file names were specified to the linker.
L1021	**cannot nest response files**
	A response file occurred within a response file.
L1022	**response line too long**
	A line in a response file was longer than 127 characters.
L1023	**terminated by user**
	You entered CTRL+C.

Number	LINK Error Message
L1024	**nested right parentheses**
	The contents of an overlay were typed incorrectly on the command line.
L1025	**nested left parentheses**
	The contents of an overlay were typed incorrectly on the command line.
L1026	**unmatched right parenthesis**
	A right parenthesis was missing from the contents specification of an overlay on the command line.
L1027	**unmatched left parenthesis**
	A left parenthesis was missing from the contents specification of an overlay on the command line.
L1030	**missing internal name**
	An **IMPORT** statement specified an ordinal in the module-definition file without including the internal name of the routine. The name must be given if the import is by ordinal.
L1031	**module description redefined**
	A **DESCRIPTION** statement in the module-definition file was specified more than once, a procedure that is not allowed.
L1032	**module name redefined**
	The module name was specified more than once (via a **NAME** or **LIBRARY** statement), a procedure that is not allowed.
L1040	**too many exported entries**
	The module-definition file exceeded the limit of 3,072 exported names.
L1041	**resident-name table overflow**
	The size of the resident-name table exceeds 65,534 bytes. (An entry in the resident-name table is made for each exported routine designated RESIDENT-NAME, and consists of the name plus three bytes of information. The first entry is the module name.)
	Reduce the number of exported routines or change some to nonresident status.

Number	LINK Error Message
L1042	**nonresident-name table overflow**

The size of the nonresident-name table exceeds 65,534 bytes. (An entry in the nonresident-name table is made for each exported routine not designated RESIDENT-NAME, and consists of the name plus three bytes of information. The first entry is the **DESCRIPTION** statement.)

Reduce the number of exported routines or change some to resident status.

L1043 **relocation table overflow**

More than 32,768 long calls, long jumps, or other long pointers appeared in the program.

Try replacing long references with short references where possible, and recreate the object module.

L1044 **imported-name table overflow**

The size of the imported-names table exceeds 65,534 bytes. (An entry in the imported-names table is made for each new name given in the IMPORTS section—including the module names—and consists of the name plus one byte.)

Reduce the number of imports.

L1045 **too many TYPDEF records**

An object module contained more than 255 TYPDEF records. These records describe communal variables. This error can appear only with programs produced by the Microsoft FORTRAN Compiler or other compilers that support communal variables. (TYPDEF is a DOS term. It is explained in the *Microsoft MS-DOS Programmer's Reference* and in other reference books on DOS.)

L1046 **too many external symbols in one module**

An object module specified more than the limit of 1,023 external symbols.

Break the module into smaller parts.

L1047 **too many group, segment, and class names in one module**

The program contained too many group, segment, and class names.

Reduce the number of groups, segments, or classes, and recreate the object file.

L1048 **too many segments in one module**

An object module had more than 255 segments.

Split the module or combine segments.

Number	LINK Error Message
L1049	**too many segments**

The program had more than the maximum number of segments. (The /SEGMENTS option specifies the maximum legal number; the default is 128.)

Relink by using the /SEGMENTS option with an appropriate number of segments.

L1050 **too many groups in one module**

LINK encountered more than 21 group definitions (GRPDEF) in a single module.

Reduce the number of group definitions or split the module. (Group definitions are explained in the *Microsoft MS-DOS Programmer's Reference* and in other reference books on DOS.)

L1051 **too many groups**

The program defined more than 20 groups, not counting DGROUP.

Reduce the number of groups.

L1052 **too many libraries**

An attempt was made to link with more than 32 libraries.

Combine libraries, or use modules that require fewer libraries.

L1053 **out of memory for symbol table**

The program had more symbolic information (such as public, external, segment, group, class, and file names) than could fit in available memory.

Try freeing memory by linking from the DOS command level instead of from a MAKE file or an editor. Otherwise, combine modules or segments and try to eliminate as many public symbols as possible.

L1054 **requested segment limit too high**

The linker did not have enough memory to allocate tables describing the number of segments requested. (The default is 128 or the value specified with the /SEGMENTS option.)

Try linking again by using the /SEGMENTS option to select a smaller number of segments (for example, use 64 if the default was used previously), or free some memory by eliminating resident programs or shells.

L1056 **too many overlays**

The program defined more than 63 overlays.

Number	LINK Error Message

L1057 **data record too large**

A LEDATA record (in an object module) contained more than 1,024 bytes of data. This is a translator error. (LEDATA is a DOS term that is explained in the *Microsoft MS-DOS Programmer's Reference* and in other DOS reference books.)

Note which translator (compiler or assembler) produced the incorrect object module and the circumstances. Please report this error to Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

L1061 **out of memory for /INCREMENTAL**

The linker ran out of memory when trying to process the additional information required for ILINK support.

Disable incremental linking.

L1062 **too many symbols for /INCREMENTAL**

The program had more symbols than can be stored in the .SYM file.

Reduce the number of symbols or disable incremental linking.

L1063 **out of memory for CodeView information**

The linker was given too many object files with debug information, and the linker ran out of space to store it.

Reduce the number of object files that have debug information.

L1064 **out of memory**

The linker was not able to allocate enough memory from the operating system to link the program. On OS/2, try increasing the swap space. Otherwise, reduce the size of the program in terms of code, data, and symbols. On OS/2, consider splitting the program into dynlink libraries.

L1070 *name* **: segment size exceeds 64K**

A single segment contained more than 64K of code or data.

Try compiling and linking using the large model.

L1071 **segment _TEXT larger than 65520 bytes**

This error is likely to occur only in small-model C programs, but it can occur when any program with a segment named _TEXT is linked using the /DOSSEG option of the LINK command. Small-model C programs must reserve code addresses 0 and 1; this range is increased to 16 for alignment purposes.

Try compiling and linking using the large model.

Number	LINK Error Message
L1072	**common area longer than 65536 bytes**

The program had more than 64K of communal variables. This error cannot appear with object files generated by the Microsoft Macro Assembler, MASM. It occurs only with programs produced by the Microsoft FORTRAN Compiler or other compilers that support communal variables.

L1073 **file segment limit exceeded**

The number of physical or file segments exceeds the limit of 254 imposed by OS/2 protected mode and by Windows for each application or dynamic-link library. (A file segment is created for each group definition, nonpacked logical segment, and set of packed segments.)

Reduce the number of segments or put more information into each segment. Make sure that the /PACKCODE and/or the /PACKDATA options are on.

L1074 *name* **: group larger than 64K bytes**

The given group exceeds the limit of 65,536 bytes.

Reduce the size of the group, or remove any unneeded segments from the group (refer to the map file for a listing of segments).

L1075 **entry table larger than 65535 bytes**

The entry table exceeds the limit of 65,535 bytes. (There is an entry in this table for each exported routine for each address that is the target of a far relocation, and for one of the following conditions when true: the target segment is designated **IOPL**; or **PROTMODE** is not enabled and the target segment is designated **MOVABLE**.)

Declare **PROTMODE** if applicable, or reduce the number of exported routines, or make some segments **FIXED** or **NOIOPL** if possible.

L1078 **file segment alignment too small**

The segment-alignment size given with the /ALIGN:*number* option was too small. Try increasing *number*.

L1080 **cannot open list file**

The disk or the root directory was full.

Delete or move files to make space.

L1081 **out of space for run file**

The disk on which the .EXE file was being written was full.

Free more space on the disk and restart the linker.

Number	LINK Error Message
L1082	*name* **: stub file not found** The linker could not open the file given in the **STUB** statement in the module-definition file.
L1083	**cannot open run file** The disk or the root directory was full. Delete or move files to make space.
L1084	**cannot create temporary file** The disk or root directory was full. Free more space in the directory and restart the linker.
L1085	**cannot open temporary file** The disk or the root directory was full. Delete or move files to make space.
L1086	**scratch file missing** An internal error has occurred. Note the circumstances of the problem and contact Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.
L1087	**unexpected end-of-file on scratch file** The disk with the temporary linker-output file was removed.
L1088	**out of space for list file** The disk (where the listing file was being written) is full. Free more space on the disk and restart the linker.
L1089	*filename* **: cannot open response file** LINK could not find the specified response file. This usually indicates a typing error.
L1090	**cannot reopen list file** The original disk was not replaced at the prompt. Restart the linker.

Number	LINK Error Message
L1091	**unexpected end-of-file on library**
	The disk containing the library was probably removed.
	Replace the disk containing the library and run the linker again.
L1092	**cannot open module-definitions file**
	The linker could not open the module-definition file specified on the command line or in the response file.
L1093	*filename* **: object not found**
	One of the object files specified in the linker input was not found.
	Restart the linker and specify the object file.
L1094	*file* **: cannot open file for writing**
	The linker was unable to open the file with write permission.
	Check file permissions.
L1095	*file* **: out of space on file**
	The linker ran out of disk space for the specified output file.
	Delete or move files to make space.
L1100	**stub .EXE file invalid**
	The file specified in the **STUB** statement is not a valid real-mode executable file.
L1101	**invalid object module**
	One of the object modules was invalid.
	If the error persists after recompiling, please contact Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.
L1102	**unexpected end-of-file**
	An invalid format for a library was encountered.

Number	LINK Error Message
L1103	**attempt to access data outside segment bounds**

A data record in an object module specified data extending beyond the end of a segment. This is a translator error.

Note which translator (compiler or assembler) produced the incorrect object module and the circumstances in which it was produced. Please report this error to Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

L1104 *filename* **: not valid library**

The specified file was not a valid library file. This error causes LINK to abort.

L1105 **invalid object due to aborted incremental compile**

Delete the object file, recompile the program, and relink.

L1113 **unresolved COMDEF; internal error**

Note the circumstances of the error and contact Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

L1114 **file not suitable for /EXEPACK; relink without**

For the linked program, the size of the packed load image plus packing overhead was larger than that of the unpacked load image.

Relink without the /EXEPACK option.

L1115 *option*: **option incompatible with overlays**

The given option is not compatible with overlays. Remove the option or else do not use overlaid modules.

L1123 *name* **: segment defined for both 16- and 32-bit.**

Define the segment as either 16-bit or 32-bit.

L1126 **conflicting iopl-parameter-words value**

An exported name was specified in the module-definition file with an **IOPL**-parameter-words value, and the same name was specified as an export by the Microsoft C **export** pragma with a different parameter-words value.

Number	LINK Error Message
L1127	**far segment references not allowed with /BINARY**

You used the /BINARY option (causing the linker to produce a .COM file) with modules that have a far segment reference. Far segment references are not compatible with the .COM file format. High-level-language modules cause this error message (unless the language supports tiny memory model). Assembly code that references a segment address also produces this error message. For example:

```
mov     ax, seg mydata
```

C.2.2 LINK Nonfatal Error Messages

Number	LINK Error Message
L2000	**imported starting address**

The program starting address as specified in the **END** statement in a MASM file is an imported routine. This is not supported by OS/2 or Windows.

L2001 **fixup(s) without data**

A FIXUPP record occurred without a data record immediately preceding it. This is probably a compiler error. (See the *Microsoft MS-DOS Programmer's Reference* for more information on FIXUPP.)

If the error persists after recompiling, please contact Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one or your manuals.

L2002 **fixup overflow at *number* in segment** *name*

This error message will be followed by one of the following:

1. "Target external '*name*.'"
2. frm seg *name1*, tgt seg *name2*, tgt offset *number*.

A fixup overflow is essentially an attempted reference to code or data that is impossible because the source location, i.e., where the reference is made "from," and the target address, i.e., where the reference is made "to," are too far apart. A close look at the source location is often all you need to correct the problem.

Revise the source file and recreate the object file. (For information about frame and target segments, see the *Microsoft MS-DOS Programmer's Reference*.)

Number	LINK Error Message
L2003	**intersegment self-relative fixup at *number* in segment name**

The program issued a near call or jump to a label in a different segment. This error most often occurs when you specifically declare an external procedure to be near and it should be declared as far.

L2004 **LOBYTE-type fixup overflow**

A LOBYTE fixup generated an address overflow. (See the *Microsoft MS-DOS Programmer's Reference* for more information.)

L2005 **fixup type unsupported**

A fixup type occurred that is not supported by the Microsoft linker. This is probably a compiler error.

Note the circumstances of the failure and contact Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

L2010 **too many fixups in LIDATA record**

The number of far relocations (pointer- or base-type) in a LIDATA record exceeds the limit imposed by the linker. This is typically produced by the **DUP** statement in an .ASM file. The limit is dynamic: a 1,024-byte buffer is shared by relocations and the contents of the LIDATA record; there are eight bytes per relocation.

Reduce the number of far relocations in the **DUP** statement.

L2011 *name* **: NEAR/HUGE conflict**

Conflicting **NEAR** and **HUGE** attributes were given for a communal variable. This error can occur only with programs produced by the Microsoft FORTRAN Compiler or other compilers that support communal variables.

L2012 *name* **: array-element size mismatch**

A far communal array was declared with two or more different array-element sizes (for instance, an array was 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 Microsoft Macro Assembler. It occurs only with the Microsoft FORTRAN Compiler and any other compiler that supports far communal arrays.

Number	LINK Error Message
L2013	**LIDATA record too large**

A LIDATA record contained more than 512 bytes. This is probably a compiler error.

Note the circumstances of the failure and contact Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

L2022 *name* **(alias** *internalname***) : export undefined**

The internal name of the given exported routine is undefined.

NumberLINK Error Message

L2023 *name* **(alias** *internalname***) : export imported**

The internal name of the given exported routine conflicts with the internal name of a previously imported routine. The set of imported and exported names cannot overlap.

L2024 *name* **: special symbol already defined**

Your program defined a symbol name already used by the linker for one of its own low-level symbols. (For example, the linker generates special symbols used in overlay support and other operations.)

Choose another name for the symbol in order to avoid conflict.

L2025 *name* **: symbol defined more than once.**

The same symbol has been found in two different object files.

L2026 **entry ordinal** *number*, **name** *name* **: multiple definitions for the same ordinal**

The given exported name with the given ordinal number conflicted with a different exported name previously assigned to the same ordinal. Only one name can be associated with a particular ordinal.

L2027 *name* **: ordinal too large for export**

The given exported name was assigned an ordinal that exceeded the limit of 3,072.

Number	LINK Error Message
L2028	**automatic data segment plus heap exceed 64K**

The total size of data declared in DGROUP, plus the value given in HEAPSIZE in the module-definition file, plus the stack size given by the /STACKSIZE option or STACKSIZE module-definition file statement, exceeds 64K. Reduce near-data allocation, HEAPSIZE, or stack.

L2029 *name* **: unresolved external**

The name that comes before `in file(s)` is the unresolved external symbol. On the next line is a list of object modules that have made references to this symbol. This message and the list are also written to the map file, if one exists.

L2030 **starting address not code (use class 'CODE')**

The program starting address, as specified in the **END** statement of an .ASM file, should be in a code segment. (Code segments are recognized if their class name ends in `'CODE'` .) This is an error in OS/2 protected mode.

The error message may be disabled by including the **REALMODE** statement in the module-definition file.

L2041 **stack plus data exceed 64K**

If the total of near data and requested stack size exceeds 64K, the program will not run correctly. The linker checks for this condition only when /DOSSEG is enabled, which is the case in the library startup module.

Reduce the stack size.

L2043 **Quick Library support module missing**

You did not link with the required QUICKLIB.OBJ module when creating a Quick library.

L2044 *name* **: symbol multiply defined, use /NOE**

The linker found what it interprets as a public-symbol redefinition, probably because you have redefined a symbol defined in a library. Relink with the /NOEXTDICTIONARY (NOE) option. If error L2025 results for the same symbol, then you have a genuine symbol-redefinition error.

Number	LINK Error Message
L2045	*segmentname* **: segment with > 1 class name not allowed with /INC**

Your program defined a segment more than once, giving the segment different class names. Different class names for the same segment are not allowed when you link with the /INCREMENTAL option. Normally, this error should never appear unless you are programming with MASM. For example, if you give the two MASM statements

```
_BSS segment 'BSS'
_BSS segment 'DATA'
```

then the statements have the effect of defining two distinct segments with the same name but different classes. This situation is incompatible with the /INCREMENTAL option.

L2047 **IOPL attribute conflict - segment:** *segname* **in group:** *grpname*

The segment *segname* is the a member of the group *grpname*, but has a different **IOPL** attribute from other segments in the group.

C.2.3 LINK Warning Messages

Number	LINK Error Message
L4000	**seg disp. included near offset in segment** *name*

This is the warning generated by the /WARNFIXUP option. See Section 13.3.31, "Issuing Fixup Warnings," for more information on that option.

L4001 **frame-relative fixup, frame ignored near** *offset* **in segment** *name*

A reference is made relative to a segment that is different from the target segment of the reference. For example, if `_foo` is defined in segment `_TEXT`, the instruction `call DGROUP:_foo` produces this warning. The frame DGROUP is ignored, so the linker treats the call as if it were `call _TEXT:_foo`.

L4002 **frame-relative absolute fixup near** *offset* **in segment** *name*

A reference is made similar to the type described in L4001, but both segments are absolute (defined with **AT**). The linker treats the executable file as if the file were to run in real mode only.

L4003 **intersegment self-relative fixup at** *offset* **in segment** *name* **pos:** *offset* **target external '***name***'**

The linker found an intersegment self-relative fixup. This error may be caused by compiling a small-model program with the /NT option.

Number	LINK Error Message
L4010	**invalid alignment specification**

The number specified in the /ALIGNMENT option must be a power of 2 in the range 2–32,768, inclusive.

L4011 **PACKCODE value exceeding 65500 unreliable**

The packing limit specified with the /PACKCODE option was between 65,500 and 65,536. Code segments with a size in this range are unreliable on some steppings of the 80286 processor.

L4012 **load-high disables EXEPACK**

The /HIGH and /EXEPACK options cannot be used at the same time.

L4013 **invalid option for new-format executable file ignored**

The use of overlays with the options /CPARMAXALLOC, /DSALLOCATE, and /NOGROUPASSOCIATION is not allowed with either OS/2 protected-mode or Windows executable files.

L4014 *option* **: invalid option for old-format executable file ignored**

The /ALIGNMENT option is invalid for real-mode executables.

L4015 **/CODEVIEW disables /DSALLOCATE**

The /CODEVIEW and /DSALLOCATE options cannot be used at the same time.

L4016 **/CODEVIEW disables /EXEPACK**

The /CODEVIEW and /EXEPACK options cannot be used at the same time.

L4020 *name* **: code-segment size exceeds 65500**

Code segments of 65,501–65,536 bytes in length may be unreliable on the Intel 80286 processor.

L4021 **no stack segment**

The program did not contain a stack segment defined with **STACK** combine type. This message should not appear for modules compiled with the Microsoft FORTRAN Compiler, but it could appear for an assembly-language module.

Normally, every program should have a stack segment with the combine type specified as **STACK**. You may 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 LINK earlier than Version 2.40 might cause this message since these linkers search libraries only once.

Number	LINK Error Message
L4022	*group1*, *group2* **: groups overlap**

The named groups overlap. Since a group is assigned to a physical segment, groups cannot overlap with either OS/2 protected-mode or Windows executable files.

Reorganize segments and group definitions so the groups do not overlap. Refer to the map file.

L4023 *name* (*internal name*) **: export internal name conflict**

The internal name of the given exported routine conflicted with the internal name of a previous import definition or export definition.

L4024 *name* **: multiple definitions for export name**

The given name was exported more than once, an action that is not allowed.

L4025 *name* **: import internal name conflict**

The internal name of the given imported routine (import is either a name or a number) conflicted with the internal name of a previous export or import.

L4026 *dynlib.import (name)* **: self-imported**

The given imported routine was imported from the module being linked. This is not supported on some systems.

L4027 *name* **: multiple definitions for import internal-name**

The given internal name was imported more than once. Previous import definitions are ignored.

L4028 *name* **: segment already defined**

The given segment was defined more than once in the **SEGMENTS** statement of the module-definition file.

L4029 *name* **: DGROUP segment converted to type data**

The given logical segment in the group DGROUP was defined as a code segment. (DGROUP cannot contain code segments because the linker always considers DGROUP to be a data segment. The name DGROUP is predefined as the automatic data segment.) The linker converts the named segment to type "data."

Number	LINK Error Message
L4030	*name* **: segment attributes changed to conform with automatic data segment** The given logical segment in the group DGROUP was given sharing attributes (**SHARED/NONSHARED**) that differed from the automatic data attributes as declared by the DATA *instance* specification (**SINGLE/MULTIPLE**). The attributes are converted to conform to those of DGROUP. Refer to Error L4029 for more information on DGROUP.
L4031	*name* **: segment declared in more than one group** A segment was declared to be a member of two different groups. Correct the source file and recreate the object files.
L4032	*name* **: code-group size exceeds 65500 bytes** The given code group has a size between 65,500 and 65,536 bytes, a size that is unreliable on some steppings of the 80286 processor.
L4034	**more than 239 overlay segments; extra put in root** Your program designated more than the limit of 239 segments to go in overlays. Starting with the 234th segment, they are assigned to the root (that is, the permanently resident portion of the program).
L4036	**no automatic data segment** The application did not define a group named DGROUP. DGROUP has special meaning to the linker, which uses it to identify the automatic or default data segment used by the operating system. Most OS/2 protected-mode and Windows applications require DGROUP. This warning will not be issued if DATA NONE is declared or if the executable file is a dynamic-link library.
L4038	**program has no starting address** Your OS/2 or Windows application had no starting address, which usually will cause the program to fail. Higher-level languages automatically specify a starting address. If you are writing an assembly-language program, specify a starting address with the **END** statement. Real-mode programs and dynamic-link libraries should never receive this message, regardless whether or not they have starting addresses.
L4042	**cannot open old version** The file specified in the **OLD** statement in the module-definition file could not be opened.

Number	LINK Error Message
L4043	**old version not segmented-executable format**

The file specified in the **OLD** statement in the module-definition file was not a valid OS/2 protected-mode or Windows executable file.

L4045 **name of output file is** *name*

The prompt for the run-file field gave an inaccurate default because /QUICKLIB option was not used early enough. The output will be a Quick library with the name given in the error message.

L4046 **module name different from output file name**

The name of the executable file as specified in the **NAME** or **LIBRARY** statement is different from the output file name. This may cause problems; consult the documentation for your operating system.

L4050 **too many public symbols for sorting**

The linker uses the stack and all available memory in the near heap to sort public symbols for the /MAP option. If the number of public symbols exceeds the space available for them, this warning is issued and the symbols are not sorted in the map file but listed in an arbitrary order.

Reduce the number of symbols.

L4051 *filename* **: cannot find library**

The linker could not find the specified file.

Enter a new file name, a new path specification, or both.

L4053 **VM.TMP : illegal file name; ignored**

VM.TMP appeared as an object-file name.

Rename the file and rerun the linker.

L4054 *filename* **: cannot find file**

The linker could not find the specified file.

Enter a new file name, a new path specification, or both.

L4067 **ignoring start address not equal to 0x100 for /TINY**

The code specified a starting address other than the assumed address of 100 hex for a .COM file created with the /TINY option. The linker is proceeding to start the .COM file at 100 hex, regardless of the specified address.

Present only in the DOS-only 3.xx linkers and the executable 5.xx linkers.

C.3 ILINK Error Messages

This section lists and describes error messages generated by the Microsoft Incremental Linker, ILINK.

Fatal errors cause the linker to end the linking session. Fatal error messages have the following format:

location : `error` `L1`*xxx* : *messagetext*

Incremental violations cause ILINK to end the linking session and carry out the command specified by the /e option. Incremental violations messages have the following format:

location : `error` `L2`*xxx* : *messagetext*

Warnings give notice of certain conditions without ending the operation of ILINK. Warnings have the following format:

location : `warning` `L4`*xxx* : *messagetext*

In all three kinds of messages, location is the input file associated with the error. If the input file is an .OBJ or .LIB file and has a module name, the module name is enclosed in parentheses, as shown in the following examples:

```
SLIBC.LIB (_file)
MAIN.OBJ (main.c)
TEXT.OBJ
```

The following error messages may appear when you link object files with the Microsoft Incremental Linker, ILINK.

C.3.1 ILINK Fatal Errors

Number	ILINK Error Message
L1105	**invalid object due to aborted incremental compile** Delete the object file, recompile the program, and relink.
L1200	**.SYM seek error** The .SYM file could not be properly read. Try redoing a full link with the /INCREMENTAL option.
L1201	**.SYM read error** The .SYM file could not be properly read. Try redoing a full link with the /INCREMENTAL option.

Number	ILINK Error Message
L1202	**.SYM write error**

The disk is full or the .SYM file already exists and has the **READONLY** attribute.

L1203 **map for segment** *name* **exceeds 64K**

The symbolic information associated with the given segment exceeds 64K bytes, an amount more than ILINK can handle.

L1204 **.ILK write error**

The disk is full or the .SYM file already exists and has the **READONLY** attribute.

L1205 **fixup overflow at** *address* **in segment** *name*

A FIXUPP object record with the given location referred to a target too far away to be correctly processed. This messages indicates an error in translation by the compiler or assembler.

L1206 **.ILK seek error**

The .ILK file is corrupted. Do a full link. If the error persists, note the circumstance of the error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

L1207 **.ILK file too large**

The .ILK file is too large for ILINK to process. Do a full link.

L1208 **invalid .SYM file**

The .SYM file is invalid; delete the file and do a full link. If the problem persists, contact Microsoft Product Support.

L1209 **.OBJ close error**

The operating system returned an error when ILINK attempted to close one of the .OBJ files.

L1210 **.OBJ read error**

The .OBJ file has an unreadable structure. Try rebuilding the .OBJ file and doing a full link. This message indicates an error in translation by the compiler or assembler.

Number	ILINK Error Message
L1211	**too many LNAMES**
	An object module has more than 255 LNAME records.
L1212	**too many SEGDEFs**
	The given object module has more than 100 SEGDEF records. A SEGDEF record defines logical segments.
L1213	**too many GRPDEFs**
	The given object module has more than 10 GRPDEF records. A GRPDEF record defines physical segments.
L1214	**too many COMDEFs**
	The total number of COMDEF and EXTDEF records exceeded the limit. The limit on the total of COMDEF records (communal data variables) and EXTDEF records (external references) is 1,023. Use fewer communal or external variables in your program.
L1215	**too many EXTDEFs**
	The total number of COMDEF and EXTDEF records exceeded the limit. The limit on the total of COMDEF records (communal data variables) and EXTDEF records (external references) is 1,023. Use fewer communal or external variables in your program.
L1216	**symbol *name* multiply defined**
	The given symbol is defined more than once.
L1217	**internal error #3**
	Note the circumstance of the error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.
L1218	**.EXE file too big, change alignment**
	The segment-sector alignment value in the .EXE file is too small to express the size of one of the segments. Do a full link and increase the alignment value with the /ALIGNMENT option to LINK.
L1219	**too many library files**
	The number of libraries exceeded ILINK's limit of 32 libraries (.LIB files). Reduce the number of libraries.

Number	ILINK Error Message
L1220	**seek error on library**
	A library (.LIB file) is corrupted. Do a full link and check your .LIB files.
L1221	**library close error**
	The operating system returned an error when ILINK attempted to close one of the libraries (.LIB files). Do a full link. If the error persists, note the circumstances of the error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.
L1222	**error closing .EXE file**
	The operating system returned an error when ILINK attempted to close the executable file. Do a full link. If the error persists, contact Microsoft Product Support.
L1223	**could not update time on** *file*
	The operating system returned an error when ILINK attempted to update the time on the given file. Possibly the file had the **READONLY** attribute set.
L1224	**invalid flag** *character*
	You used incorrect syntax on the ILINK command line.
L1225	**only one -e command allowed**
	You used incorrect syntax on the ILINK command line.
L1226	**terminated by user**
	You pressed CTRL+C or CTRL+BREAK, an action that interrupts and terminates ILINK.
L1227	**file** *name* **write protected**
	The .EXE, .ILK, or .SYM file that ILINK attempted to update has the **READONLY** attribute.
L1228	**file** *name* **missing**
	ILINK could not find one of the .OBJ files specified on the command line.
L1229	**invalid .OBJ format**
	There may be one of several problems: error in compiler translation, corrupted object file, invalid object file (possibly text file), or object file could not be read or found.

Number ILINK Error Message

L1230 **invalid *file* record: position** = *address*

The given .OBJ file has an invalid format or one unrecognized by ILINK. This message may indicate an error in translation by the compiler or assembler.

L1231 **file *name* was not full linked**

You specified an .OBJ file in the ILINK command line that was not in the list of files in the most recent full link.

L1232 **cannot run** *program*

ILINK is unable to execute a program specified for execution with the \e command-line option. Make sure the program is in the search path and is an .EXE or .COM file.

L1233 *program* **returned** *return-code*

The given program was specified with the \e option. When ILINK executed this program, it terminated with the given nonzero return code. ILINK cannot continue to the next commands, if any.

L1234 **error creating** *file*

ILINK was unable to create the batch file for executing the \e commands. Make sure the directory given in TMP or TEMP, or the current directory, exists and can be written to.

L1235 **error writing to** *file*

ILINK experienced an error while writing the batch file for executing the \e commands. Make sure the drive for TMP or TEMP or the current drive has enough free space.

L1240 **far references in STRUC fields not supported**

ILINK currently does not support **STRUC** definitions like this:

```
extrn   func:FAR
    rek     STRUC
    far_adr DD      func    ; Initialized far address
                            ;   within a STRUC
    rek     ENDS
```

To use ILINK, change your code to get rid of the far address within the **STRUC**.

L1241 **too many defined segments**

ILINK has a limit of 255 physical segments (that is, segments defined in the object module as opposed to groups or logical segments). To use ILINK, reduce the number of segments.

Number	ILINK Error Message
L1242	**too many modules**

The program exceeds ILINK's limit of 1,204 modules. Reduce the number of modules.

L1243 **cannot link 64K-length segments**

The program has a segment larger than 65,535 bytes.

L1244 **cannot link iterated segments**

ILINK cannot handle programs linked with the /EXEPACK linker option.

C.3.2 Incremental Violations

Number	ILINK Error Message
L1250	*number* **undefined symbols**

A number of symbols were referred to in fixups but never publicly defined in the program. The given number indicates how many of these undefined symbols were found.

L1251 **invalid module reference** *library*

The program makes a dynamic-link reference to a dynamic-link library that is not recognized or declared by the .EXE file.

L1252 **file** *name* **does not exist**

ILINK could not find the given file required for ILINK operation.

L1253 **symbol** *name* **deleted**

A symbol was deleted from an incrementally linked module.

L1254 **new segment definition** *name*

A segment was added to the program.

L1255 **changed segment definition** *name*

The segment contribution changed for the given module; it contributed to a segment it did not previously contribute to, or a segment contribution was removed.

L1256 **segment** *name* **grew too big**

The given segment grew beyond the padding for the given module.

Number	ILINK Error Message
L1257	**new group definition** *name* A new group was defined via the GROUP directive in assembly language or via the \ND C compiler option.
L1258	**group** *name* **changed to include** *segment* The list of segments included in the given group changed.
L1259	**symbol** *name* **changed** The given data symbol moved (is now at a new address).
L1260	**cannot add new communal data symbol** *name* A new communal data symbol was added as an uninitialized variable in C or with the COMM feature in MASM.
L1261	**communal variable** *name* **grew too big** The given communal variable changed size too much.
L1262	**invalid symbol type for** *symbol* A symbol which was previously a code symbol became a data symbol or vice versa.
L1263	**new Codeview symbolic info** A module previously compiled without \Zi was compiled with \Zi.
L1264	**new line-number info** A module previously compiled without \Zi or \Zd was compiled with \Zi or \Zd.
L1265	**new public CodeView info** New information on public symbol addresses was added.
L1266	**invalid .EXE file** The .EXE file is invalid. Make sure you are using an up-to-date linker. If the error persists, note the circumstances of the error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.
L1267	**invalid .ILK file** The .ILK file is invalid. Make sure you are using an up-to-date linker. If the error persists, notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

Number	ILINK Error Message
L1268	**.SYM/.ILK mismatch**

The .SYM and .ILK files are out of sync. Make sure you are using an up-to-date linker. If the error persists, note the circumstances of the error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

L1269 **library** *name* **has changed**

The given library has changed.

L1270 **entry table expansion not implemented**

The program call tree changed in such a way that ILINK could not process it correctly. This problem is caused by new calls to a routine from another routine that did not call it before. Do a full link.

L1271 **segment** *index* **with relocs exceeds 64K; cannot move**

The given segment, referred to by its index within the program's segment table, is too big along with its runtime relocations for ILINK to process the segment correctly.

L1272 **.ILK read error**

The .ILK file does not exist or was not in the expected format.

L1273 **out of memory**

ILINK ran out of memory for processing the input. If you are running ILINK while using the NMAKE utility, try running ILINK from the shell (that is, directly from the operating-system prompt). Otherwise, do a full link.

C.3.3 ILINK Warning Messages

Number	ILINK Error Message
L4201	**fixup frame relative to an (as yet) undefined symbol - assuming ok**

See documentation for LINK error messages L4001 and L4002.

L4202 **module contains TYPEDEFs - ignored**

The .OBJ file contains type definitions. ILINK ignores these records.

Number	ILINK Error Message
L4203	**module contains BLKDEFs - ignored**

The .OBJ file contains records no longer supported by Microsoft language compilers.

L4204 **old .EXE free information lost**

The free list in the .EXE file has been corrupted. The free list represents "holes" in the EXE file made available when segments moved to new locations.

L4205 **file *name* has no useful contribution**

The given module makes no contribution to any segment.

L4206 **main entry point moved**

The program starting address changed. You may want to consider doing a full link.

C.4 LIB Error Messages

Error messages generated by the Microsoft Library Manager, LIB, have one of the following formats:

{*filename* | `LIB`} `: fatal error U1xxx:` *messagetext*
{*filename* | `LIB`} `: nonfatal error U2xxx:` *messagetext*
{*filename* | `LIB`} `: warning U4xxx:` *messagetext*

The message begins with the input-file name (*filename*), if one exists, or with the name of the utility. If possible, LIB prints a warning and continues operation. In some cases errors are fatal, and LIB terminates processing. LIB may display the following error messages.

C.4.1 Fatal LIB Error Messages

Number	LIB Error Message
U1150	**page size too small**

The page size of an input library was too small, indicating an invalid input .LIB file.

U1151 **syntax error : illegal file specification**

A command operator such as a minus sign (–) was given without a following module name.

Number	LIB Error Message
U1152	**syntax error : option name missing**

A forward slash (/) was given without an option after it.

U1153 **syntax error : option value missing**

The /PAGESIZE option was given without a value following it.

U1154 **option unknown**

An unknown option was given. Currently, LIB recognizes the /PAGESIZE, /NOIGNORECASE, and /IGNORECASE options.

U1155 **syntax error : illegal input**

The given command did not follow correct LIB syntax as specified in Chapter 15, "Managing Libraries with LIB."

U1156 **syntax error**

The given command did not follow the correct LIB syntax as specified in Chapter 15, "Managing Libraries with LIB."

U1157 **comma or new line missing**

A comma or carriage return was expected in the command line but did not appear. This may indicate an incorrectly placed comma, as in the following line:

```
LIB math.lib,-mod1+mod2;
```

The line should have been entered as follows:

```
LIB math.lib -mod1+mod2;
```

U1158 **terminator missing**

Either the response to the `Output library` prompt or the last line of the response file used to start LIB did not end with a carriage return.

U1161 **cannot rename old library**

LIB could not rename the old library to have a .BAK extension because the .BAK version already existed with read only protection.

Change the protection on the old .BAK version.

U1162 **cannot reopen library**

The old library could not be reopened after it was renamed to have a .BAK extension.

Number	LIB Error Message
U1163	**error writing to cross-reference file**
	The disk or root directory was full.
	Delete or move files to make space.
U1170	**too many symbols**
	More than 4,609 symbols appeared in the library file.
U1171	**insufficient memory**
	LIB did not have enough memory to run.
	Remove any shells or resident programs and try again, or add more memory.
U1172	**no more virtual memory**
	Try using the /NOEXTDICTIONARY. The current library exceeds the 512K byte limit imposed by LIB option. Try using the /NOEXITDICTIONARY or reduce the number of object modules.
U1173	**internal failure**
	Note the circumstances of the failure and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.
U1174	**mark: not allocated**
	Note the circumstances of the failure and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.
U1175	**free: not allocated**
	Note the circumstances of the error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.
U1180	**write to extract file failed**
	The disk or root directory was full.
	Delete or move files to make space.
U1181	**write to library file failed**
	The disk or root directory was full.
	Delete or move files to make space.

Number	LIB Error Message
U1182	*filename* **: cannot create extract file**

The disk or root directory was full, or the specified extract file already existed with read-only protection.

Make space on the disk or change the protection of the extract file.

U1183 **cannot open response file**

The response file was not found.

U1184 **unexpected end-of-file on command input**

An end-of-file character was received prematurely in response to a prompt.

U1185 **cannot create new library**

The disk or root directory was full, or the library file already existed with read-only protection.

Make space on the disk or change the protection of the library file.

U1186 **error writing to new library**

The disk or root directory was full.

Delete or move files to make space.

U1187 **cannot open VM.TMP**

The disk or root directory was full.

Delete or move files to make space.

U1188 **cannot write to VM**

The library manager cannot write to the virtual memory. Note the circumstances of the failure and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

U1189 **cannot read from VM**

The library manager cannot read the virtual memory. Note the circumstances of the error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

Number	LIB Error Message
U1190	**interrupted by user**

You interrupted LIB during its operation, with CTRL+C or CTRL+BREAK.

Number	LIB Error Message
U1200	*name* **: invalid library header**

The input library file had an invalid format. It was either not a library file or it had been corrupted.

Number	LIB Error Message
U1203	*name* **: invalid object module near** *location*

The module specified by *name* was not a valid object module.

C.4.2 Nonfatal LIB Error Messages

Number	LIB Error Message
U2152	*filename* **: cannot create listing**

The directory or disk was full, or the cross-reference-listing file already existed with read-only protection.

Make space on the disk or change the protection of the cross-reference-listing file.

Number	LIB Error Message
U2155	*modulename* **: module not in library; ignored**

The specified module was not found in the input library.

Number	LIB Error Message
U2157	*filename* **: cannot access file**

LIB was unable to open the specified file.

Number	LIB Error Message
U2158	*libraryname* **: invalid library header; file ignored**

The input library had an incorrect format.

Number	LIB Error Message
U2159	*filename* **: invalid format** *hexnumber*; **file ignored**

The signature byte or word hexnumber of the given file was not one of the following recognized types: Microsoft library, Intel library, Microsoft object, or Xenix archive.

C.4.3 Warning LIB Error Messages

Number	LIB Error Message
U4150	*modulename* **: module redefinition ignored**

A module was specified to be added to a library but a module with the same name was already in the library, or a module with the same name was found more than once in the library.

U4151 *name* **: symbol defined in** *modulename*, **redefinition ignored**

The specified symbol was defined in more than one module.

U4153 *number* **: page size too small; ignored**

The value specified in the /PAGESIZE option was less than 16.

U4155 *modulename* **: module not in library**

A module specified to be replaced does not already exist in the library. LIB adds the module anyway.

U4156 *libraryname* **: output-library specification ignored**

An output library was specified in addition to a new library name. For example, specifying

```
LIB new.lib+one.obj,new.lst,new.lib
```

where `new.lib` does not already exist, causes this error.

Number	LIB Error Message
U4157	**insufficient memory, extended dictionary not created**

Insufficient memory prevented LIB from creating an extended dictionary. The library is still valid, but the linker will not be able to take advantage of the extended dictionary to speed linking.

U4158 **internal error, extended dictionary not created**

An internal error prevented LIB from creating an extended dictionary. The library is still valid, but the linker will not be able to take advantage of the extended dictionary to speed linking.

C.5 NMAKE Error Messages

Error messages from the NMAKE utility have one of the following formats:

{*filename* | NMAKE} : fatal error U1*xxx*: *messagetext*
{*filename* | NMAKE} : warning U4*xxx*: *messagetext*

The message begins with the input-file name (*filename*) and line number, if one exists, or with the name of the utility.

NMAKE generates the following error messages.

C.5.1 Fatal NMAKE Error Messages

Number	NMAKE Error Message

U1000 **syntax error: ')' missing in macro invocation**

A left parenthesis (( ) appeared without a matching right parenthesis ( )) in a macro invocation. The correct form is $(*name*), or $*n* for one-character names.

U1001 **syntax error : illegal character '*character*' in macro**

A nonalphanumeric character other than an underscore (_) appeared in a macro.

U1002 **syntax error : bad macro invocation '$'**

A single dollar sign ($) appeared without a macro name associated with it. The correct form is $(*name*). To use a dollar sign in the file, type it twice or precede it with a caret (^).

U1003 **syntax error : '=' missing in macro**

The equal sign (=) was missing in a macro definition. The correct form is '*name* = *value*'.

U1004 **syntax error : macro name missing**

A macro invocation appeared without a name. The correct form is $(*name*).

U1005 **syntax error : text must follow ':' in macro**

A string substitution was specified for a macro, but the string to be changed in the macro was not specified.

U1016 **syntax error : closing '"' missing**

An opening double quotation mark (") appeared without a closing double quotation mark.

Number	NMAKE Error Message
U1017	**unknown directive** '*directive*'
	The directive specified is not one of the recognized directives.
U1018	**directive and/or expression part missing**
	The directive is incompletely specified. The expression part is required.
U1019	**too many nested if blocks**
	Note the circumstances of the failure and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.
U1020	**EOF found before next directive**
	A directive, such as **!ENDIF**, was missing.
U1021	**syntax error : else unexpected**
	An **!ELSE** directive was found that was not preceded by **!IF**, **!IFDEF**, or **!IFNDEF**, or was placed in a syntactically incorrect place.
U1022	**Missing terminating char for string/program invocation :** '*character*'
	The closing double quotation mark (") in a string comparison in a directive was missing, or the closing bracket (]) in a program invocation in a directive was missing.
U1023	**syntax error present in expression**
	An expression is invalid. Check the allowed operators and operator precedence.
U1024	**illegal argument to !CMDSWITCHES**
	An unrecognized command switch was specified.
U1031	**file name missing**
	An include directive was found, but the name of the file to include was missing.
U1033	**syntax error :** '*string*' **unexpected**
	The specified string is not part of the valid syntax for a makefile.
U1034	**syntax error : separator missing**
	The colon (:) that separates target(s) and dependent(s) is missing.

Number	NMAKE Error Message

U1035 **syntax error : expected separator or '='**

Either a colon (:), implying a dependency line, or an equal sign (=), implying a macro definition, was expected.

U1036 **syntax error : too many names to left of '='**

Only one string is allowed to the left of a macro definition.

U1037 **syntax error : target name missing**

A colon (:) was found before a target name was found. At least one target is required.

U1038 **internal error : lexer**

Note the circumstances of the failure and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

U1039 **internal error : parser**

Note the circumstances of the failure and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

U1040 **internal error : macro-expansion**

Note the circumstances of the failure and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

U1041 **internal error : target building**

Note the circumstances of the failure and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

U1042 **internal error : expression stack overflow**

Note the circumstances of the failure and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

U1043 **internal error : temp file limit exceeded**

Note the circumstances of the failure and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

Number	NMAKE Error Message
U1044	**internal error : too many levels of recursion building a target** Note the circumstances of the failure and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.
U1050	*user-specified text* The message specified with the **!ERROR** directive is displayed.
U1051	**'***progname***' usage : [-acdeinpqrst -f makefile -x stderrfile] [macrodefs] [targets]** An error was made trying to invoke NMAKE. Use the specified form.
U1052	**out of memory** The program ran out of space in the far heap. Split the description into smaller and simpler pieces.
U1053	**file '***filename***' not found** The file was not found. The file name might not be properly specified in the makefile.
U1054	**file '***filename***' unreadable** The file cannot be read. The file might not have the appropriate attributes for reading.
U1055	**can't create response file '***filename***'** The response file cannot be created.
U1056	**out of environment space** The environment space limit was reached. Restart the program with a larger environment space.
U1057	**can't find command.com** The COMMAND.COM file could not be found.
U1058	**unlink of file '***filename***' failed** Unlink of the temporary response file failed.

Number	NMAKE Error Message
U1059	**terminated by user**
	Execution of NMAKE aborted because you typed CTRL+C or CTRL+BREAK.
U1070	**cycle in macro definition** '*macroname*'
	A circular definition was detected in the macro definition specified. This is an invalid definition.
U1071	**cycle in dependency tree for target** '*targetname*'
	A circular dependency was detected in the dependency tree for the specified target. This is invalid.
U1072	**cycle in include files** *filenames*
	A circular inclusion was detected in the include files specified. That is, each file includes the other.
U1073	**don't know how to make** '*targetname*'
	The specified target does not exist and there are no commands to execute or inference rules given for it. Hence it cannot be built.
U1074	**macro definition too long**
	The macro definition is too long.
U1075	**string too long**
	The text string would overflow an internal buffer.
U1076	**name too long**
	The macro name, target name, or build-command name would overflow an internal buffer. Macro names may be at most 128 characters.
U1077	'*program*' **: return code** *value*
	The given program invoked from NMAKE failed, returning the error code *value*.
U1078	**constant overflow at** '*directive*'
	A constant in *directive*'s expression was too big.
U1079	**illegal expression: divide by zero present**
	An expression tries to divide by zero.

Number	NMAKE Error Message
U1080	**operator and/or operand out of place: usage illegal** The expression incorrectly uses an operator or operand. Check the allowed set of operators and their precedence.
U1081	*'program'* **: program not found** NMAKE could not find the given program in order to run it. Make sure that the program is in the current path and has the correct extension.
U1082	*command* **cannot execute command: out of memory** NMAKE cannot execute the given command because there is not enough memory. Free memory and run NMAKE again.
U1085	**can't mix implicit and explicit rules** A regular target was specified along with the target for a rule (which has the form *.fromext.toext*). This is invalid.
U1086	**inference rule can't have dependents** Dependents are not allowed when an inference rule is being defined.
U1087	**can't have : and :: dependents for same target** A target cannot have both a single-colon (:) and a double-colon (::) dependency.
U1088	**invalid separator on inference rules: '::'** Inference rules can use only a single-colon (:) separator.
U1089	**can't have build commands for pseudotarget** *'targetname'* Pseudotargets (for example, .PRECIOUS, .SUFFIXES) cannot have build commands specified.
U1090	**can't have dependents for pseudotarget** *'targetname'* The specified pseudotarget (for example, .SILENT, .IGNORE) cannot have a dependent.
U1091	**invalid suffixes in inference rule** The suffixes being used in the inference rule are invalid.

Number	NMAKE Error Message
U1092	**too many names in rule**

An inference rule cannot have more than one pair of extensions (*.fromext.toext*) as a target.

Number	NMAKE Error Message
U1093	**can't mix special pseudotargets**

It is illegal to list two or more pseudotargets together.

C.5.2 Warning NMAKE Error Messages

Number	NMAKE Error Message
U4011	**command file can only be invoked from command line**

A command file cannot be invoked from within another command file. Such an invocation is ignored.

Number	NMAKE Error Message
U4012	**resetting value of special macro '***macroname***'**

The value of a macro such as S(MAKE) was changed within a description file.

The name by which this program was invoked is not a tagged section in the TOOLS.INI file.

Number	NMAKE Error Message
U4015	**no match found for wildcard '***filename***'**

There are no file names that match the specified target or dependent file with the wild-card characters asterisk (*) and question mark (?).

Number	NMAKE Error Message
U4016	**too many rules for target '***targetname***'**

Multiple blocks of build commands are specified for a target using single colons (:) as separators.

Number	NMAKE Error Message
U4017	**ignoring rule** *rule* **(extension not in .SUFFIXES)**

The rule was ignored because the suffix(es) in the rule are not listed in the .SUFFIXES list.

Number	NMAKE Error Message
U4018	**special macro undefined '***macroname***'**

The special macro *macroname* is undefined.

Number	NMAKE Error Message
U4019	**Filename '***filename***' too long; truncating to 8.3**

The base name of the file has more than eight characters or the extension has more than three characters. NMAKE truncates the name to an eight-character base and a three-character extension.

Number	NMAKE Error Message
U4020	**removed target '***target***'**

Execution of NMAKE was interrupted while it was trying to build the given target, and therefore the target was incomplete. Because the target was not specified in the .PRECIOUS list, NMAKE has deleted it.

C.6 EXEMOD Error Messages

Error messages from the Microsoft EXE File Header Utility, EXEMOD, have one of the following formats:

{*filename* | EXEMOD} : fatal error U1xxx: *messagetext*
{*filename* | EXEMOD} : warning U4xxx: *messagetext*

The message begins with the input-file name (*filename*), if one exists, or with the name of the utility. If possible, EXEMOD prints a warning and continues operation. In some cases errors are fatal and EXEMOD terminates processing.

EXEMOD generates the following error messages:

C.6.1 Fatal EXEMOD Error Messages

Number	EXEMOD Error Message
U1050	**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 hyphen (-). The single brackets ([ ]) in the error message indicate that your choice of the item within them is optional.

U1051 **invalid .EXE file : bad header**

The specified input file is not an executable file or it has an invalid file header.

U1052 **invalid .EXE file : actual length less than reported**

The second and third fields in the input-file header indicate a file size greater than the actual size.

U1053 **cannot change load-high program**

When the minimum allocation value and the maximum allocation value are both 0, the file cannot be modified.

Number	EXEMOD Error Message
U1054	**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.

U1055 *filename* **: cannot find file**

The file specified by *filename* could not be found.

U1056 *filename* **: permission denied**

The file specified by *filename* was a read only file.

C.6.2 Warning EXEMOD Error Messages

Number	EXEMOD Error Message
U4050	**packed file**

The given file was a packed file. This is a warning only.

U4051 **minimum allocation less than stack; correcting minimum**

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.

U4052 **minimum allocation greater than maximum; correcting maximum**

If the minimum allocation value is greater than the maximum allocation value, the maximum allocation value is adjusted. This is a warning message only; EXEMOD will still modify the file. The values shown if you ask for a display of DOS header values will be the values after the packed file is expanded.

C.7 SETENV Error Messages

Messages generated by the Microsoft Environment Expansion Utility, SETENV, have the following format:

{*filename* | `SETENV` } : `fatal error U1` *xxx*: *messagetext*

The message begins with the input-file name (*filename*), if one exists, or with the name of the utility.

SETENV generates the following fatal error messages:

Number	SETENV Error Message
U1080	**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.

U1081 **unrecognizable COMMAND.COM**

The COMMAND.COM file was not one of the accepted versions (DOS Versions 2.0, 2.1, 2.11, 3.0, and 3.1).

U1082 **maximum for Version 3.1 : 992**

You specified a file recognized as COMMAND.COM for IBM PC-DOS, Version 3.1, and gave an environment size greater than 992 bytes, the maximum allowed for that version.

U1083 **maximum environment size : 65520**

The environment size specified was greater than 65,520 bytes, the maximum size allowed.

U1084 **minimum environment size : 160**

The environment size specified was less than 160 bytes, the minimum size allowed.

U1085 *filename* **: cannot find file**

The specified file was not found, perhaps because it was a directory or some other special file.

U1086 *filename* **: permission denied**

The specified file was a read only file.

U1087 *filename* **: unknown error**

An unknown system error occurred while the specified file was being read or written.

Try running SETENV again.

8087 or 80287 coprocessor Intel hardware products that perform much faster math calculations than the main processor.

adapter A term sometimes used to refer to printed-circuit cards that plug into a computer and control a device, such as a video display or a printer.

address An expression that evaluates to a location in memory. Addresses can be given in the segment offset format. If the segment is not given, the default segment is assumed. The default segment is CS for commands related to code and DS for commands related to data.

address range A range of memory bounded by two addresses. The range can be specified in the normal format by giving the starting and ending addresses (inclusive), or it can be specified in the object-range format by specifying the starting address followed first by the letter (uppercase or lowercase) and then by the number of objects in the range (0x100 L 10, for example, specifies the range from 0x100 to 0x109, inclusive).

Applications Program Interface (API) The set of calls a program uses to obtain services from the operating system. The term API denotes a service interface, whatever its form. Generally used to refer to OS/2 system calls.

argc The conventional name for the first argument to the main function in a C source program (an integer specifying how many arguments are passed to the program from the command line).

argument A value passed to a function.

argv The conventional name for the second argument to the main function in a C source program (a pointer to an array of strings). The first string is the program name and each following string is an argument passed to the program from the command line.

array A set of elements with the same type.

ASCII (American Standard Code for Information Interchange) A set of 256 codes that many computers use to represent letters, digits, special characters, and other symbols. Only the first 128 of these codes are standardized; the remaining 128 are special characters defined by the computer manufacturer.

assembly mode The mode in which the CodeView debugger displays assembly-language-instruction mnemonics to represent the code being executed.

base name The part of a file name before the extension. For example, SAMPLE is the base name of the file SAMPLE.BAS.

BASIC A programming language included with versions of DOS. BASIC is an acronym for Beginner's All-purpose Symbolic Instruction Code.

Basic Input/Output System (BIOS) The code built into system memory that provides hardware interface routines for programs. You can trace into the BIOS with the CodeView debugger, using assembly mode.

batch file A text file containing MS-DOS commands that can be invoked from the MS-DOS command line.

breakpoint A specified address where program execution will be halted. The CodeView debugger interrupts execution whenever the program reaches an address where a breakpoint has been set. See also "watchpoint" and "tracepoint" for a description of conditional breakpoints.

buffer An area in memory in which a copy of the file is kept and changed as you edit. This buffer is copied to disk when you do a save operation.

call gate A special LDT or GDT entry that describes a subroutine entry point rather than a memory segment. A far call to a call gate selector will cause a transfer to the entry point specified in the call gate. This is a feature of the 80286/80386 hardware and is normally used to provide a transition from a lower privilege state to a higher one.

CGA IBM's Color Graphics Adapter.

character string A sequence of bytes treated as a set of ASCII letters or numbers.

Child process A process created by another process (its parent process).

click To press and release one of the mouse buttons while pointing the mouse at an object on the screen.

Color Graphics Adapter (CGA) A video adapter capable of displaying text characters or graphics pixels. Color can also be displayed with the appropriate display monitor.

command An instruction you use to control a computer program, such as DOS or an application program.

command file A file that contains the program or instructions required to carry out a command. If the file's extension is COM or EXE, the command file contains machine instructions; if its extension is BAT, the command file is a batch file and contains DOS commands; if its extension is CMD, the command file contains OS/2 commands.

compile The action performed to translate programming language statements to a form that can be executed by the computer.

constant A value that does not change during program execution. A variable, on the other hand, is a value that can—and usually does—change during program execution.

constant expression Any expression that evaluates to a constant and may involve integer constants, character constants, floating-point constants, enumeration constants, type casts to integral and floating-point types, and other constant expressions.

CPU Central Processing Unit, or the main processor in a computer. For example, the CPU is an Intel 8088 in PCs and an 80286 in PC/ATs.

Ctrl-C Same as CTRL-BREAK.

Ctrl-S Same as CTRL-NUM LOCK.

cursor The thin blinking line that represents the current location where the next character you type appears. The cursor automatically moves to the dialog window when you start entering a command.

debugger A program that helps the programmer locate the source of problems found during run-time testing of a program.

dialog box A box that appears when you choose a menu command that requires additional information.

dialog commands Commands entered in the dialog window in window mode, or any command in sequential mode. Dialog commands consist of one- or two-character commands that can usually be followed by arguments.

dialog window The window at the bottom of the CodeView screen where dialog commands can be entered and previously entered dialog commands can be reviewed.

double precision A real (floating-point) value that occupies eight bytes of memory. Double-precision values are accurate to 15 or 16 digits.

drag To point the mouse at an object on the screen, press a mouse button, and then move the mouse while holding the button down.

dump Contents of memory displayed at a specified memory location. In the CodeView debugger, the size of the object to be displayed is specified with a type character from the following list: **A** (ASCII), **B** (Byte), **I** (Integer), **U** (Unsigned Integer), **W** (Word), **D** (Double Word), **SP** (Short Real), **L** (Long Real), or **T** (10-Byte Real).

dynamic link A method of postponing the resolution of external references until loadtime or run-time. A dynamic link allows the called subroutines to be packaged, distributed, and maintained independently of their callers. OS/2 extends the dynamic link mechanism to serve as the primary method by which all system and nonsystem services are obtained.

dynamic-link library A file, in a special format, that contains the binary code for a group of dynamically linked subroutines.

dynamic-link routine See "dynamic link."

Enhanced graphics adapter (EGA) A video adapter capable of displaying in all the modes of the color graphics adapter (CGA) plus additional modes. The CodeView 43 option displays in the EGA's 43-line text mode.

environment strings A series of user-definable and program-definable strings associated with each process. The initial values of environment strings are established by a process's parent.

environment table The part of MS-DOS that stores environment variables and their values.

environment variable A variable stored in the environment table that provides MS-DOS with information (where to find executable files and library files, where to create temporary files, etc.).

Esc Escape key.

escape sequence A specific combination of a backslash (\) followed by a letter or combination of digits. The combination represents white-space and nongraphic characters within strings and character constants.

executable file A file with an extension of .EXE, .COM, .BAT, or .CMD. Executable files can be run by typing the file name at the system prompt.

executable program A file that contains executable program code. When the name of the file is typed at the system prompt, the statements in the file are executed.

exit code A code returned by a program to MS-DOS indicating whether the program ran successfully.

expression A combination of operands and operators that yields a single value.

Family Applications Program Interface (Family API) A standard execution environment under MS-DOS versions 2.x and 3.x and OS/2. The programmer can use the Family API to create an application that uses a subset of OS/2 functions (but a superset of MS-DOS 3.x functions) and that runs in a binary-compatible fashion under MS-DOS versions 2.x and 3.x and OS/2.

far address A memory location specified by using a segment (location of a 64K block) and an offset from the beginning of the segment. Far addresses require four bytes—two for the segment and two for the offset. A far address is also known as a segmented address. See "near address."

file A named collection of information stored on a disk. The file usually contains either data or a program.

flags A register that contains individual bits, each of which signals a condition that can be tested by a machine-level instruction. In other registers, the contents of the register are considered as a whole, while in the flags register only the individual bits have meaning. In the CodeView debugger, the current values of the most commonly used bits of the flags register are shown at the bottom of the register window. See "registers."

flipping A screen-exchange method that uses the video pages of the CGA or EGA to store both the debugging and output screens. Video pages are areas of memory reserved for screen storage. When you request the other screen, the two video pages are exchanged. This method is faster than swapping—the other screen-exchange method—but it does not work with the MA or with programs that do graphics or use the video pages. See also "screen exchange" and "swapping."

function A subroutine or procedure that returns a value.

function call A call to a a subroutine that performs a specific action. In C (source mode), subroutines are called functions. In assembly language (assembly mode), subroutines are called procedures.

global symbol A symbol that is available throughout the entire program. In the CodeView debugger, function names are always global symbols. See also "local symbol."

global variable A variable that is available throughout a module.

grandparent process The parent process of a process that created a process.

hexadecimal The base-16 numbering system whose digits are 0 through F (the letters A through F represent the decimal numbers 10 through 15). Hexadecimal is often used in computer programming because it is easily converted to and from binary, the base-2 numbering system the computer itself uses.

highlight A reverse-video area in a text box, window, or menu marking the current command chosen or text that has been selected for copying or deleting.

I/O privilege mechanism A facility that allows a process to ask a device driver for direct access to the device's I/O ports and any dedicated or mapped memory locations it has. The I/O privilege mechanism can be used directly by an application or indirectly by a dynamic-link package.

identifier A name that identifies a register or a location in memory. The terms identifier and symbol are used synonymously in CodeView documentation.

IEEE format (Institute for Electrical and Electronic Engineers, Inc.) A method of representing floating-point numbers internally.

include file A source file that is merged into a program with the **$INCLUDE** metacommand or the C **#include** directive.

integer A whole number represented inside the machine as a 16-bit two's complement binary number. An integer has a range of -32,768 to +32,767. See "long integer."

interrupt call A machine-level procedure that can be called to execute a BIOS, MS-DOS, or other function. You can trace into BIOS interrupts with the CodeView debugger, but not into the MS-DOS interrupt (0x21).

label A symbol (identifier) representing an address in the code segment (CS) register. Labels in C programs can be either function names or labels for goto statements.

link The step that the linker performs to produce an executable file. The link step resolves references to procedures or variables in other modules and creates a complete program ready for execution.

linking The process in which the linker loads modules into memory, computes absolute offset addresses for routines and variables in relocatable modules, and resolves all external references by searching the run-time library. After loading and linking, the linker saves the modules it has loaded into memory as a single executable file.

local symbol A symbol that only has value within a particular function. A function argument or a variable declared as auto or static within a function can be a local symbol. See "global symbol."

local variable A variable whose scope is confined to a particular unit of code, such as the module-level code, or a procedure within a module.

long integer A whole number represented inside the machine by a 32-bit two's complement value. Long integers have a range of -2,147,483,648 to +2,147,483,647. See "integer."

lvalue An expression (such as a variable name) that refers to a single memory location and is required as the left-hand operand of an assignment operation or the single operand of a unary operator. For example, `X1` is an lvalue, but `X1+X2` is not.

machine code A series of binary numbers that a microprocessor executes as program instructions.

macro A method for representing a long series of characters or statements with a symbol. The macro is expanded by the C or MASM preprocessor. C and MASM each have their own conventions for defining macros.

math coprocessor An optional hardware component, such as an 8087 or 80287 chip, that improves the speed of arithmetic involving floating-point numbers.

menu bar The bar at the top of the CodeView display containing menu titles and the titles Trace and Go.

Microsoft binary format A method of representing floating-point numbers internally.

module A discrete group of statements. Every program has at least one module (the main module). In most cases, each module corresponds to one source file. When you save a program containing multiple modules, each module is saved in a separate disk file.

Monochrome Adapter (MA) A video adapter capable of displaying only in black and white. Most monochrome adapters display text only; individual graphics pixels cannot be displayed. The CodeView debugger recognizes monochrome adapters and automatically selects swapping as the screen-exchange mode.

mouse A pointing device that fits under your hand and rolls in any direction on a flat surface. By moving the mouse, you can move the mouse pointer in a corresponding direction on the screen. See "pointer."

mouse pointer The reverse-video square that moves to indicate the current position of the mouse. The mouse pointer only appears if a mouse is installed. To select an item with the mouse, move the mouse until the pointer rests on the item.

multitasking operating system An operating system in which two or more programs/threads can execute simultaneously.

NAN An acronym that stands for "not a number." The 8087 or 80287 coprocessor generates NANs when the result of an operation cannot be represented in the IEEE format. For example, if you try to add two positive numbers whose sum is larger than the maximum value supported by the data type, the coprocessor returns a NAN instead of the sum.

near address A memory location specified by using only the offset from the start of the segment. A near address requires only two bytes. See "far address."

null pointer A pointer to nothing, expressed as the integer value 0.

object file A file (with the extension .OBJ) containing relocatable machine code produced by compiling a program and used by the linker to form an executable file.

object module The contents of an object file after the file has been made part of a stand-alone library.

object range See "address range."

offset The number of bytes from the beginning of a segment in memory to a particular byte in that segment.

output screen The screen where program output is shown. The Screen Exchange command (\), Output from the View menu, and the F4 key can be used to switch to this screen. The output screen is the same as it would be if you ran the debugged program outside of the CodeView debugger.

parent process A process that creates another process, called the child process.

PID (Process Identification Number) A unique code that OS/2 assigns to a process when the process is created. The PID may be any value except 0.

pointer A variable containing the address of another variable. See "mouse pointer."

popup menu A menu that pops up when you point the mouse cursor to the menu title and press a mouse button. In the CodeView debugger, popup menus also pop up when you press the ALT key and the first letter of the menu title at the same time. You can make a selection from the menu by dragging the highlight up or down with the mouse, by pressing the UP or DOWN direction key to move the highlight, or by pressing the ALT key and the first letter of the selection title at the same time.

port The electrical connection through which the computer sends and receives data to and from devices or other computers.

precedence The relative position of an operator in the hierarchy that determines the order in which expressions are evaluated.

printf A function in the C standard library that prints formatted output according to instructions supplied with a type-specifier argument. The CodeView debugger uses a subset of the **printf** type specifiers to format expression values.

privilege mode A special execution mode (also known as ring 0) supported by the 80286/80386 hardware. Code executing in this mode can execute restricted instructions that are used to manipulate key system structures and tables. Only the OS/2 kernel and device drivers run in this mode.

procedure A general term for a SUB or FUNCTION.

procedure call A call to a subroutine that performs a specific action. In assembly language (assembly mode), subroutines are called procedures. In C (source mode), subroutines are called functions.

process The executing instance of a binary file. In OS/2, the terms task and process are used interchangeably. A process is the unit of ownership, and processes own resources such as memory, open files, dynlink libraries, and semaphores.

processor See "CPU."

program step To trace the next source line in source mode or the next instruction in assembly mode. If the source line or instruction contains a function, procedure, or interrupt call, the call is executed to the end and the CodeView debugger is ready to execute the instruction after the call. See "trace."

protected mode The operating mode of the 80286 or 80386 microprocessor that allows the operating system to use features that protect one application from another (also called protect mode). Protected mode in OS/2 supports multitasking and a whole range of special services not supported in DOS.

radix The number system in which numbers are specified. In the CodeView debugger, numbers can be entered in three radixes: 8 (octal), 10 (decimal), or 16 (hexadecimal). The default radix is 10.

real mode The operating mode of the 80286 or 80386 microprocessor that runs programs designed for the 8086/8088 microprocessor. All programs for the DOS environment run in real mode.

redirection To specify the device from which a program will receive input or to which it will send output. Normally program input comes from the keyboard, and program output goes to the screen. Redirection involves specifying a device (or file) other than the default device. In the MS-DOS operating system, input is redirected with the less-than symbol (<) and output is redirected with the greater-than symbol (>). The same symbols are used in the CodeView debugger to redirect input or output of the debugging session. In addition, the equal sign (=) can be used to redirect both input and output.

register window The optional window in which the central processing unit (CPU) registers and the bits of the flag register are displayed.

registers Special memory within the processor, where byte- or word-sized data can be stored during machine-level processing. The registers used with the 8086 family of processors are: AX, BX, CX, DX, SP, BP, SI, DI, DS, ES, SS, CS, IP, and the flags register. See "flags."

regular expressions A system of specifying text patterns that match variable text strings. The CodeView debugger supports a subset of the regular-expression characters used in the XENIX and UNIX operating systems. Regular expressions can be used to find strings in source files.

routine A procedure, function, or subroutine residing in a module, and usually carrying out a specific task.

run-time library A file containing standard functions for programs written in the Microsoft C language.

scope The parts of a program in which a given symbol has meaning. The scope of an item may be limited to the file, function, block, or function prototype in which it appears.

screen exchange The method by which both the output screen and the debugging screen are kept in memory so that both can be updated simultaneously and either viewed at the user's convenience. The two screen-exchange modes are flipping and swapping. See "flipping" and "swapping."

scroll To move text up and down, or left and right, in order to see parts that cannot fit on the screen.

segment An area of memory, less than or equal to 64K, containing code or data.

semaphore A software flag or signal used to coordinate the activities of two or more threads. A semaphore is commonly used to protect a critical section.

sequential mode The mode in which all CodeView output is sequential and no windows are available. Input and output scroll down the screen and the old output scrolls off the top of the screen when the screen is full. You cannot examine previous commands after they scroll off the top. This mode is required with computers that are not IBM compatible. The mouse and most window commands are not supported in sequential mode. Any debugging operation that can be done in window mode can also be done in sequential mode.

shell escape A method of leaving the CodeView debugger without losing the current debugging context. You can "escape to a shell," do various MS-DOS tasks, and then return to the debugger. The debugging screen will be the same as when you left it. The CodeView debugger creates the shell by saving all current operations to memory and invoking a second copy of COMMAND.COM.

single precision A real (floating-point) value that occupies four bytes of memory. Single-precision values are accurate to seven decimal places.

source file A text file containing BASIC or C-language code.

source mode The mode in which the CodeView debugger displays C source code to represent the code being executed.

stack A dynamically shrinking and expanding area of memory in which data items are stored in consecutive order and removed on a last-in, first-out basis. The stack is most commonly used to store information for function and procedure calls and for local variables.

stack frame A portion of a program's stack that contains a procedure's local variables and parameters. (In protected mode, each thread has its own stack.)

stack trace A symbolic representation of the functions that have been executed to reach the current instruction address. As a function is executed, the function address and any function arguments are pushed on the stack (the area of memory starting at the address of the SS register). Therefore, a trace of the stack always shows the currently active functions and the values of their arguments.

standard output The device to which a program sends its output unless the output is redirected. In normal DOS operation, standard output is the video display.

start-up code The code that the C compiler places at the beginning of every program to control execution of the program code. When the CodeView debugger is loaded, the first source line executed runs the entire start-up code. If you switch to assembly mode before executing any code, you can trace through the start-up code.

static linking The combining of multiple compilands into a single executable file, thereby resolving undefined external references.

string A contiguous sequence of characters, often identified by a symbolic name. In this example, Hello is a string: `PRINT "Hello"`. A string may be a constant or a variable.

structure A set of elements, which may be of different types, grouped under a single name.

structure member One of the elements of a structure.

subroutine A unit of BASIC code terminated by the **RETURN** statement. Program control is transferred to a subroutine with a **GOSUB** statement.

swapping A screen-exchange method that uses buffers to store the debugging and output screens. When you request the other screen, the two buffers are exchanged. This method is slower than flipping, the other screen-exchange method, but it works with any adapter and any type of program. See "flipping" and "screen exchange."

symbol A name that identifies a location in memory. The terms "symbol" and "identifier" are used synonymously in CodeView documentation.

temporary file A file that DOS may create when told to redirect command input or output. The file is deleted by DOS when the command is completed.

thread The OS/2 mechanism that allows more than one path of execution through the same instance of an application program.

thread ID The handle of a particular thread within a process.

thread of execution The passage of the CPU through the instruction sequence. In DOS, each program has only one thread of execution.

toggle A function key or menu selection that switches between two (and in some cases three) states. When used as a verb, toggle means to reverse the status of a feature. For example, the F3 key is a toggle that switches between source, mixed, and assembly modes. You can press the F3 key to toggle between the three modes.

trace To trace the next source line in source mode, or the next instruction in assembly mode. If the source line or instruction contains a function, procedure, or interrupt call, the first source line or instruction of the call is executed. The CodeView debugger is ready to execute the next instruction inside the call. See "program step."

tracepoint A variable breakpoint that is taken when a specified value changes. The value to be tested can be either the value of a CodeView expression, or any of the bytes in a range of memory. Tracepoints can slow program execution significantly, since the CodeView debugger has to check after executing each source line in source mode or after each instruction in assembly mode to see if the value has changed. See "breakpoint."

type cast An operation in which an operand of one type is converted to an operand of a different type.

type casting To specify a type specifier in parentheses preceding an expression to indicate the type of the expression's value. For example, if x and y are integer values with the values 5 and 2 respectively, the expression x/y indicates integer division and the expression has the value 2. The expression (float)x/y indicates real-number division and has the value 2.5.

unary operator An operator that takes a single operand. Unary operators in the C language are the complement operators (- ~ !), indirection operator (*), increment (++) and decrement (--) operators, address-of operator (&), and sizeof operator. The unary plus operator (+) is also implemented syntactically, but has no semantics associated with it.

unresolved external A symbol referenced in one assembly-language module, but not made PUBLIC in another module that is linked with it. Unresolved external references are usually caused by misspellings or by omitting the name of the module containing the desired symbol from the link command line.

user-defined type A composite data structure in BASIC that can contain both string and numeric variables, similar to a C-language structure or Pascal record. User-defined types are defined with **TYPE** statements.The data structure is defined by a **TYPE...END TYPE** statement.

watch window The window where watch statements and their values are displayed. The three kinds of watch statements are watch expressions, watchpoints, and tracepoints.

watchpoint A variable breakpoint that is taken when a specified expression becomes nonzero (true). Watchpoints can slow program execution significantly, since the CodeView debugger has to check after executing each source line in source mode or after each instruction in assembly mode to see if the value is true. See "breakpoint."

wildcard character A special character that, like the wild card in a poker game, can be used to represent any other character. DOS recognizes two wildcard characters: the question mark (?), which can represent any single character, and the asterisk (*), which can represent more than one character.

window A term that refers to an area on the screen used either to display part of a file or to enter statements.

window commands Commands that work only in the CodeView debugger's window mode. Window commands consist of function keys, mouse selection, CTRL and ALT key combinations, and selections from popup menus.

window mode The mode in which the CodeView debugger displays separate windows, which can change independently. The debugger has mouse support and a wide variety of window commands in window mode.

& (ampersand), LIB command symbol, 276
* (asterisk)
  Comment command, 206
  FORTRAN multiplication operator, 69
  LIB command symbol, 274, 281
  regular expressions, used in, 356
** (asterisks), FORTRAN exponentiation operator, 69
@ (at sign)
  NMAKE special character, 291
  Redraw command, 195
  register prefix, 80
\ (backslash)
  NMAKE continuation character, 287
  Screen Exchange command, 195
{ } (braces), NMAKE character, 289
[ ] (brackets), regular expressions, used in, 354
^ (caret)
  exponentiation operator, BASIC, 74
  NMAKE escape character, 355–356
: (colon)
  Delay command, 207
  LINK command, 229
  NMAKE separator, 292
  operator, 69, 74, 81
, (comma)
  LIB command symbol, 271
  LINK command symbol, 229
- (dash)
  NMAKE special character, 291
  regular expressions, used in, 355
$ (dollar sign)
  NMAKE macros, used in, 294
  regular expressions, used in, 356
:: (double colon), NMAKE separator, 292
= (equal sign)
  assignment operator, FORTRAN, 69
  Redirected Input and Output command, 205
! (exclamation point)
  NMAKE directives, used in, 300
  NMAKE special character, 292
  Shell Escape command, 199
/ (forward slash)
  CodeView option designator, 20
  FORTRAN division operator, 69
  LINK option character, 236
  Search command, 197
> (greater-than sign)
  CodeView prompt, 34–35
  Redirected Output command, 204
< (less-than sign), Redirected Input command, 203
– (minus sign), 273, 280–281
–* (minus sign-asterisk), LIB command symbol, 274, 281
– + (minus sign-plus sign), LIB command symbol, 273, 275, 281
(number sign)
  NAN (not a number), 118
  NMAKE comment character, 290
  Tab Set command, 200
( ) (parentheses), FORTRAN operator, 69–70
. (period)
  Current Location command, 166
  operator
    C, 66
    FORTRAN, 69–70
  regular expressions, used in, 354
+ (plus sign)
  FORTRAN operator, 69
  LIB command symbol
    Intel, XENIX files, used with, 269
    libraries, combining and specifying, 273, 275, 281
    object files, appending, 279–281
    using, 273
  LINK command symbol, 231–232
" " (quotation marks), Pause command, 207
; (semicolon)
  LIB command symbol, 270–271, 277, 282
  LINK command symbol, 229, 231–232
  NMAKE command separator, 290
_ (underscore), symbol names, used in, 67, 70
/2 option, CodeView, 22
7 (8087 command), 130
10-byte reals, dumping, 124
/43 option, CodeView, 23
/50 option, CodeView, 23
386 option, 56
8087
  command, 129
  coprocessor, 129, 174
  stack, 131
8259 trapping, 25–26

A

A (Assemble command), 172
/A option
  LINK, 237
  NMAKE, 287
/a option, ILINK, 265
Absolute addresses, 81

Accessing bytes, 83
Adapters, using two, 22
Addresses
  absolute, 143
  full, 81, 143
  segment start, 255
/ALIGNMENT option, LINK, 237
Alignment types, 255–256
Ampersand (&), LIB command symbol, 276
.AND. operator, 69
API.LIB, 341, 343
APILMR.OBJ, 343
Application import libraries, 320
Archives, XENIX, 269, 281
Arguments
  CodeView
    dialog commands, 61, 63
    program, 97
  LINK options, 237
  program, 18
  routine, 57, 166
Arithmetic operators, FORTRAN, 69
Arrays
  copying, 185
  multidimensional, and BASIC, 74
AS, NMAKE macro, 295
ASCII characters, displayed by CodeView, 119
Assemble command, 171
Assembly
  address, 172
  mode
    example, 162
    setting, 159
    using, 32
  rules, 173
Assembly, programs. *See* Macro Assembler
Assignment operator
  BASIC, 74, 97
  FORTRAN, 69
Asterisk (*), comment command symbol, 274, 281
At sign (@)
  NMAKE special character, 291
  Redraw command, 195
  register prefix, 80
Attributes. *See* Segment attributes

B

/B option
  CodeView, 24
  LINK, 237
  NMAKE, 287
/BA option, LINK, 238
Backslash (\), Screen Exchange command, 195
BASIC
  colon (:) operator, 74
  constants, 75
  expression evaluator, 65
  expressions, 73
  intrinsic functions, 76
  programs
    CodeView, preparing for, 12
    compiling and linking, 12
    writing source code, 12
  strings, 76
  symbols, 74
Batch files, exit codes, 357
/BATCH option (/BA), LINK, 237
BC (Breakpoint Clear Command), 137
BD (Breakpoint Disable command), 137
BE (Breakpoint Enable command), 138
BEGDATA class name, 240
/BINARY option (/BI), LINK, 238
BIND
  command line, 343
  described, 341
  options, 343
BL (Breakpoint List command), 139
Black-and-white display, CodeView, 24
Blocks of memory
  copying, 185
  filling, 184
  moving, 185
BP. *See* Breakpoint Set command
Braces ({ }), NMAKE character, 289
Brackets ([ ]), regular expressions, used in, 354
Breakpoint Clear command
  Run menu selection, 52
  using, 136
Breakpoint Disable command, 137
Breakpoint Enable command, 138
Breakpoint List command, 139
Breakpoint Set command
  F9 function key, 39, 59–60
  mouse, executing with, 43
  using, 133
Breakpoints
  address, 94
  conditional, 53, 133
  defined, 133
  deleting, 136
  displaying, 34, 134
  Go command, used with, 92
  listing, 139

BSS class name, 240
Buffer, CodeView command, 36, 62
BY operator, 83

C

C language
  CodeView, case sensitivity, 67
  constants, 67
  expressions, 66
  operators, 66
    CodeView, preparing for, 9
    compiling and linking, 10
    macros, 9
    writing source, 9
  strings, 68
  symbols, 67
/C option, CodeView, 24, 287
/c option, ILINK, 265
CGA (Color Graphics Adapter), 22–24
Calling conventions, 167
Calls
  menu, 57, 167
  stepping over, 91
  tracing into, 89
Canonical frame number. *See* Frame number
Caret (^)
  exponentiation operator, BASIC, 74
  NMAKE escape character, 290, 294
  regular expressions, used in, 355–356
Case sensitivity
  BASIC-expression evaluator, 75
  C symbols, 67
  CodeView, 8, 55, 63
  FORTRAN symbols, 70
  LINK, 227, 246
  Macro Assembler options, 16
CL driver, 10
Class names
  BEGDATA, 240
  BSS, 240
  CODE, 240
  linking procedure, used in, 256
  STACK, 240
Class types, 256
Click, defined, 41
!CMDSWITCHES directive, NMAKE, 301
/CO linker option, 8–9
/CO option, LINK, 238
CODE class name, 240
CODE statement, 325
Code symbol, defined, 262
CodeView
  case sensitivity, 8, 63
  colon (:) operator, 69, 74, 81
  command line, 18
  compatibility, 28–30
  compiler options
    /Od, 8
    /Zd, 8
    /Zi, 8
  defaults, 118
  EGA compatibility, 29
  executable files, 9, 16–17
  exit codes, 93–94
  interrupt program execution, 88
  language support
    BASIC, 12
    C, 9
    FORTRAN, 11
    Macro Assembler, 14
    Pascal, 13
  linker option (/CO), 8–9, 32
  mixed-language support, 16
  operators
    BY, 83
    DW, 85
    memory, 83
    WO, 84
  optimization, effect of, 8
  options
    /C, 24, 287
    command line, used in, 18
    described, 23–31
    /L, 28, 210
    /O, 29, 211
    summary, 20–22
  parameters, program, 18
  period operator (.), 66, 69–70
  restrictions, 5
  source-module files, location of, 18, 47
  start-up
    command line, 18
    commands, 24
    file configuration, 17
  symbolic information, 9
  symbols, 67, 70, 74
  syntax, summary, 191
CodeView, error messages, 361
/CODEVIEW option, LINK, 238

Colon (:)
Delay command, 207
NMAKE separator, 292
operator, 69, 74, 81
Color graphics adapter (CGA), 22–24, 27
.COM extension, debugged files, used for, 18, 32
Combine types
COMMON, 256
PRIVATE, 256
PUBLIC, 256
STACK, 256
Command buffer, 36, 62
Command line
BIND, 343
CodeView, 18
ILINK, 265
IMPLIB, 320
LIB, 270
LINK, 226
NMAKE, 286, 293
COMMAND.COM, Shell command, used with, 47, 198
Commands, CodeView
8087 command, 129
Assemble, 171
Breakpoint Clear
Run menu selection, 51–52
using, 136
Breakpoint Disable, 137
Breakpoint Enable, 138
Breakpoint List, 139
Breakpoint Set
F9 function key, 39, 59–60
mouse, executing with, 43
using, 133
calls
stepping over, 91
tracing through, 89
command buffer, 62
Comment, 206
Current Location, 165
cursor movement, 36
Delay, 207
dialog commands, 35, 61, 148
Display Expression, 100
Dump
10-Byte Reals, 124
ASCII, 119
Bytes, 119
default size, 117–118
Double Words, 122
Integers, 120
Long Reals, 123
Commands, Codeview (*continued*)
Dump (*continued*)
Short Reals, 122–123
Unsigned Integers, 121
Words, 121
Enter
ASCII, 179
Bytes, 178
default size, 178
Double Words, 181
Enter Integers, 179
Long Reals, 183
Short Reals, 182
Unsigned Integers, 180
using, 175
Words, 181
ESCAPE key, 41
Examine Symbols, 111
Execute, 51, 95
Exit, 47
Expression, 100
Fill Memory, 184
Go
destination address, 93
F5 function key, 38, 59
mouse, executing with, 44
using, 92
Goto
comment line, 93
F7 function key, 39
mouse, executing with, 43
using, 92
Graphic Display, 107
grow (increase) window size, 36
Help
F1 function key, 38, 59
menu, 58
using, 191
window mode, 58
input, redirecting, 203
mnemonic keys, 40
Move Memory, 185
Option, 201
Output, 49
output, redirecting, 204
Pause, 207
Port Input, 127
Port Output, 186
Program Step
F10 function key, 39, 60
using, 90
Quit, 192

Commands, Codeview (*continued*)
Radix setting, 193
Redirected Input and Output, 25, 203, 205
Redraw, 195
Registers
displaying, 128
F2 function key, 38, 59
mouse, executing with, 44
values, changing, 187
View menu selection, 49
Restart
Run menu selection, 51
using, 96
Screen Exchange
F4 function key, 38, 59
using, 195
scroll
line down, 42
line up, 42
page down, 37, 42
page up, 36, 42
to bottom, 37, 42
to top, 37, 42
to top of page, 37
Search
menu selections, 49
regular expressions, used with, 353
using, 196
separator line movement, 42
Set Mode
dialog command, 159
F3 function key, 38, 59
View menu selection, 48
Shell Escape
File menu selection, 47
using, 198
Stack Trace
display contents, 57
using, 166
T (Trace command), 89
Tab Set, 200
tiny (reduce) window size, 36
Trace
F8 function key, 39, 59
using, 88
Tracepoint, 60
tracing into calls, 89
Unassemble, 161–162
View, 163
Watch
menu selections, 52
sequential mode, 60
Commands, Codeview (*continued*)
Watch Delete, 53–54, 152
Watch Delete All, 54
Watch expression, 142
Watch List, 60, 154
Watchpoint
sequential mode, 60
setting, 146
Watch menu selection, 53
window, 61
Commands, NMAKE description file, 289–290
Comment command, 206
Comment lines, source code, 93–94, 134
Comments, NMAKE description file, 289–290
COMMON combine type, 256
Compare Memory command, 125
Compiler errors and CodeView, 8
Compiler options
/Od, 8
/Zd, 8
/Zi, 8, 14
COMSPEC environment variable, 198
Concatenation, string, BASIC, 74
Conditional breakpoints, 53, 133, 141
Conjunction operator, FORTRAN, 69
Consistency checking, LIB, 271, 282
Constant expressions, 301
Constant numbers
BASIC, 75
C, 67
FORTRAN, 70
CONTROL+BREAK, 39, 88
CONTROL+C, 39, 88, 148
CONTROL+F (Find command), 49
CONTROL+G (grow window size), 36
CONTROL+T (tiny window size), 36
CONTROL+U (Delete Watch command), 53
CONTROL+W (Add Watch command), 52
Controlling
data loading, 240
executable-file loading, 243
LINK, 236
segments, number of, 252
stack size, 253
Copying arrays, 185
/CP option, LINK, 198, 239
/CPARMAXALLOC option, LINK, 198, 239
Cross-reference listing, LIB, 282
CTRLEND key (scroll to bottom), 37
CTRLHOME key (scroll to top), 37
Current Location command, 165
Current location line, 34

Cursor, CodeView, 34, 61
CV.EXE, location of, 17
CV.HLP, location of, 17, 58

D

D (Dump command), 118
/D option
  Compiler, 25
  NMAKE, 287
DA (Dump ASCII command), 119
Dash (-)
  command-line options, 8, 20
  NMAKE special character, 291
  regular expressions, used in, 355
Data segments, loading, 240
DATA statement, 327
Data symbol, defined, 262
DB (Dump Bytes command), 119
DD (Dump Double Words command), 122
DEBUG, 33
Debugging, preparing for, (/CODEVIEW option), 238
Decimal notation
  BASIC, 75
  C, 67
  FORTRAN, 70
Default data segment, 329, 343
Defaults, CodeView
  address-range size, 118
  expression format, 144
  IBM Personal Computer, used with, 20
  radix, 166, 193
  segment, 81
  start-up behavior, 19
  type
    Dump command, 118
    Enter command, 178
    Watch command, 144, 151
Defaults, utilities
  libraries, ignoring, 234, 245
  NMAKE, MAKEFILE, 286–287
  responses
    LIB, 277
    LINK, 231
Definitions
  code symbol, 262
  data symbol, 262
  logical segment, 262
  memory model, 262
  module, 262
  physical segment, 262
  segment, 261
Delay command, 207
Dependency lines, 289
Dependents
  directory searches, 289
  macros for, 295
  specifying, 289
Description blocks
  described, 289
  inference rules used with, 297
  multiple for one target, 292
Description files, NMAKE
  commands, 289
  comments, 289–290
  described, 288
  macro definitions in, 293
  MAKEFILE, 287
  specifying, 286
DESCRIPTION statement, 325
Destination address, Go command, used with, 93
DGROUP, 329
  memory, allocating below, 240
  segment order, 240
DI (Dump Integers command), 120
Dialog
  box, 35, 41, 45
  commands, 34, 61, 148
  window, 34
Directives, NMAKE
  !CMDSWITCHES, 301
  !ERROR, 300
  !INCLUDE, 301
  described, 300
  !ELSE, 300
  !ENDIF, 300
  !IF, 300
  !IFDEF, 300
  !IFNDEF, 300
  !UNDEF, 294, 300
Disjunction, inclusive, 69
Display, CodeView
  assembly mode, 159, 162
  CONTROL+G (grow window size), 36
  CONTROL+T (tiny window size), 36
  CTRLEND key (scroll to bottom), 37
  CTRLHOME key (scroll to top), 37
  cursor, 34, 61
  dialog box, 35, 41, 45
  display mode, 88, 164
  DOWN ARROW key (cursor down), 36
  END key (scroll to bottom of a page), 37
  highlight, 35
  HOME key (scroll to top of page), 37

Display, CodeView (*continued*)
menu bar, 35
message box, 35, 41, 45
mouse pointer, 35
output screen, 195
PGDN key (scroll page down), 37
PGUP key (scroll page up), 36
register window, 34, 38
scroll bar, 35
separator line, 34
set mode command, 38
UP ARROW key (cursor up), 36
window, 33, 35
Display Expression command, 100
Display mode, 88, 162, 164
DL (Dump Long Reals command), 123
/DO option, LINK, 239
Dollar sign ($)
NMAKE macros, used in, 294
regular expressions, used in, 356
DOS, program header, 308
DOSCALLS.LIB, 343
/DOSSEG option, LINK, 239
Double Words (units of memory), 85
DOWN ARROW key (cursor down), 36
Drag, defined, 41
Drivers
CL, 10
FL, 11
DS (Dump Short Reals command), 122–123
/DS option, LINK, 240
DS register, described, 240
/DSALLOCATE option, LINK, 240
DT (Dump 10-Byte Reals command), 124
DU (Dump Unsigned Integers command), 121
Dump address, 118
Dump commands
10-Byte Reals, 124
ASCII, 119
Bytes, 119
default size, 118
Double Words, 122
Integers, 120
Long Reals, 123
Short Reals, 122–123
Unsigned Integers, 121
using, 117
Words, 121
DW (Dump Words command), 121
DW operator, 85
Dynamic-link function calls, 315
Dynamic-link libraries
debugging, 210-211
described, 315, 319

E

E commands
Enter, 178
Execute, 96
/E option
CodeView, 26
LINK, 241
NMAKE, 287, 297
/e option, ILINK, 265
EA (Enter ASCII command), 179
EB (Enter Bytes command), 178
Echo, redirection, used with, 204
ED (Enter Double Words command), 181–182
_edata, 240
EGA (Enhanced Graphics Adapter), 23–24, 27, 29
EI (Enter Integers command), 179
EL (Enter Long Reals command), 183
!ELSE directive, NMAKE, 300
_end, 240
End (special variable), 240
END key (scroll to bottom), 37
!ENDIF directive, NMAKE, 300
Enhanced graphics adapter (EGA), 23–24, 27, 29
Enter commands
ASCII, 179
Bytes, 178
default size, 178
Double Words, 181
Integers, 179
Long Reals, 183
Short Reals, 182
Unsigned Integers, 180
using, 175
Words, 181
Environment
enlarging, 311
variables
LIB, 233
LINK, 254
TMP, used by LINK, 235
.EQ. operator, 69
Equal sign (=)
assignment operator, FORTRAN, 69
Redirected Input and Output command, 205
.EQV. operator, 69
!ERROR directive, NMAKE, 300

Error handling, NMAKE, 291
Error messages
  CodeView, 361
  EXEMOD, 412
  ILINK, 391
  LIB, 399
  LINK, 372
  NMAKE, 405
  SETENV, 413
Errorlevel codes. *See* Exit codes
Errors, logic and syntax, 8
ES (Enter Short Reals command), 182
Escape character, NMAKE, 290, 294
ESCAPE key, 41
EU (Enter Unsigned Integers command), 180
EW (Enter Words command), 181
Examine Symbols command, 111
Exclamation point (!)
  NMAKE directives, used in, 300
  NMAKE special character, 292
  Shell Escape command, 199
.EXE extension, 18, 32
EXE header information, 309
Executable files
  CodeView
    format, 7, 9
    start-up, required for, 18
  command line, used in, 18
  compressing, 307
  extensions, 228
  headers
    changing, 307
    information, 309, 347
    size, 310
  initial register values, 310
  LINK
    naming with, 228
    specifying with prompts, 230
    specifying with response file, 232
  load size, 309
  loading, 243
  location of, 17
  maximum allocation, 310
  minimum allocation, 310
  naming, default, 228
  overlay number, 310
  packing, 241
  protected-mode format, 344
  segmented-executable format, 344
  size, 309
Executable image, 255
Execute command, 51, 95
EXEHDR, 347–348
EXEMOD
  described, 307
  display, 310
  error messages, 412
  exit codes, 359
  /H option, 308
  header information, 309
  /MAX option, 308
  maximum allocation, changing, 239
  /MIN option, 308
  /STACK option, 308
/EXEPACK option, LINK, 241, 342
EXETYPE statement, 339
Exit codes
  CodeView, 93–94
  DOS batch files, with, 357
  error level, 357
  EXEMOD, 359
  LINK, 358
  NMAKE, 357, 359
  programs for, 358
  SETENV, 359
  using, 357
Exit, DOS command, 47, 198
Exiting from LINK, 226
Expanded memory, 26
Exponentiation operator
  BASIC, 74
  FORTRAN, 69
Export definitions, 315
EXPORTS statement, 334
Expression evaluation
  CodeView requirement, 65
  Display Expression command, 100
Expressions
  BASIC, 73
  C, 66
  FORTRAN, 69
  regular
    searches, used in, 50, 196
    specifying, 353
Extensions
  auto option, 65
  default, LINK, 227
  executable files, 228
  libraries
    LIB, used with, 269–270, 279
    LINK, used with, 227
  map files, 227, 229, 244
  object files, 227–228
  .SUFFIXES, listing with, 303

F

F (Fill Memory command), 184
/F option
  CodeView, 26
  LINK, 241
  NMAKE, 286–287
F1 key (Help), 38, 59, 192
F2 key (Register), 38, 59, 128
F3 key
  (Set source/assembly), 59, 160
  (Set source/mixed/assembly), 38
F4 key (Screen Exchange), 38, 59
F5 key (Go), 38, 59, 93
F6 key (switch cursor), 36
F7 key (Goto), 39
F8 key (Trace), 39, 59, 89
F9 key
  (Breakpoint Clear), 136
  (Breakpoint Enable), 138
  (Breakpoint Set), 39, 59–60
F10 key (Program Step), 39, 60, 91
Family API, 341–342
Far-return mnemonic (RETF), 173
/FARCALLTRANSLATION option, LINK, 241
Fatal Error messages, LINK, 372
Files, menu
  DOS Shell, 47
  Exit, 47
  Load, 163
  Quit, 192
  Shell, 198
Fill Memory command, 184
Fixups, 257
FL driver, 11
Flag bits
  mouse, changing with, 44
  values
    changing, 187
    displaying, 128
Flag mnemonics, 188
Flipping, CodeView, 26–27
Format specifiers, 100–101
FORTRAN
  CodeView
    case sensitivity, 70
    support, 11
  colon (:) operator, 69
  compiler, 11
  constants, 70
  expression evaluator, 65
FORTRAN (*continued*)
  expressions, 69
  identifiers, 70
  include files, 11
  intrinsic functions, 72–73
  operators, 69
  programs
    CodeView, preparing for, 11
    writing source code, 11
  strings, 71
  symbols, 70
Forward slash (/)
  division operator, FORTRAN, 69
  option character, LINK, 236
  option designator
    CodeView, 20
    compilers, 8
  Search command, 197
Frame number, 255
Function calls
  stepping over, 91
  tracing into, 89
Function keys
  F1 (Help), 38, 58, 192
  F2 (Register), 38, 59, 128
  F3
    (Set source/assembly), 59, 160
    (Set source/mixed/assembly), 38
  F4 (Screen Exchange), 38, 59
  F5 (Go), 38, 59, 93
  F6 (Switch Cursor), 36
  F7 (Goto), 39
  F8 (Trace), 39, 59, 89
  F9
    (Breakpoint Clear), 136
    (Breakpoint Enable), 138
    (Breakpoint Set), 39, 59
  F10 (Program Step), 39, 60, 91
Functions
  binding, 342
  calls to, 167
  examining, 111
  intrinsic
    BASIC, 76
    FORTRAN, 72–73
  viewing, 57

G

G (Go command), 94
.GE. operator, 69
Global symbols. *See* Public symbols

Go command
F5 function key, 38, 59
mouse, executing with, 44
using, 92
Goto command
comment line, 93
F7 function key, 39
mouse, executing with, 43
using, 92
Graphic Display command, 107
Graphics adapters
43-line mode, 23
EGA, compatibility, 29
screen-exchange mode, 26–27
using two, 22
Graphics programs, debugging, 22, 204
Greater-than operator, FORTRAN, 69
Greater-than sign (>)
CodeView prompt, 34
Redirected Output command, 204
Greater-than-or-equal-to operator, FORTRAN, 69
Groups
DGROUP, 240
linking procedures, used in, 257
.GT. operator, 69

H

H (Help command), 192
/H option, EXEMOD, 308
Hardware ports, output to, 186
/HE option, LINK, 242
Header information, EXE file, 309
HEAPSIZE statement, 337
Help command
F1 function key, 38, 59
help file, 58
shell command, used with, 191
using, 191
view menu selection, 48
window mode, 58
Help menu, 58
/HELP option, LINK, 242
Hexadecimal notation
BASIC, 75
C, 67
FORTRAN, 70
/HI option, LINK, 240, 243
Highlight, 35
HOME key (scroll to top), 37

I

/I option
CodeView, 26
NMAKE, 288
/i option, ILINK, 265
IBM PC
compatibility with CodeView, 28, 30
recognizing CodeView, 20
Identifiers
BASIC, 74
C, 67
FORTRAN, 70
!IF directive, NMAKE, 300
!IFDEF directive, NMAKE, 300
!IFNDEF directive, NMAKE, 300
.IGNORE pseudotarget, 303
ILINK
command line, 265
described, 261
error messages, 391
guidelines, 262–263
incremental violations, 266
ILINK options
/a, 265
/c, 265
/e, 265
/i, 265
/v, 265
ILINKSTB.OVL, 264
IMPLIB
command line, 320
described, 320
Import
definitions, 315
libraries, 317–318
IMPORTS statement, 335
/INC option, LINK, 243, 264
!INCLUDE directive, NMAKE, 301
INCLUDE environment variable, NMAKE, 301
Include files
assembly programs, 15
BASIC programs, 12
C programs, 9
CodeView, 5
FORTRAN programs, 11
Pascal programs, 13
Inclusive disjunction operator, FORTRAN, 69
Incremental linking. *See* ILINK
/INCREMENTAL option, LINK, 243, 264

Incremental violations, ILINK, 266
IND (indefinite), 118
Indentation, 200
Indirect register instructions, 174
Indirection levels, CodeView, 67
INF (infinity), 118
/INF option, LINK, 243
Inference rules, 297, 299
Infinity, 118
/INFORMATION option, LINK, 243
Initializing data, 184
Instruction, current, 88, 90
Instruction-name synonyms, 174
Integers, dumping, 120
Interrupt, DOS functions, 89
Intrinsic functions
  BASIC, 76
  FORTRAN, 72–73
Invoking, NMAKE, 286

K

K (Stack Trace command), 168

L

/L option
  Codeview, 28
  using, 210
L (Restart command), 97
Labels, finding, 51, 197
.LE. operator, 69
Less-than operator, FORTRAN, 69
Less-than-or-equal-to operator, FORTRAN, 69
Less-than sign (<), Redirected Input command, 203
LET, 74
Levels of indirection, CodeView, 67
/LI option, LINK, 244
LIB
  addition commands, 278
  backup library file, 279
  changing with, 269, 279–280
  commands, specifying, 271–272
  consistency checking, 271, 282
  creating, 269, 279
  default responses, 277
  error messages, 399
  extending lines, 276
  files, listing, 282
  input, 270
  Intel, 269, 281
LIB (*continued*)
  libraries
    combining, 273, 279, 281
    index, 279
    modules, adding and deleting, 273, 279–280
  listing files, 279
  modules, extracting and deleting, 274, 278, 281
  object modules, deleting, 273, 278, 280
  operations, order of, 278
  options, /PAGESIZE, 283
  output, 274–275
  running
    command line, 270
    prompts, 276
    response file, 277
  terminating, 279
  variable, 233
LIB command symbols
  asterisk (*), 274, 281
  minus sign (-), 273, 275, 280
  minus sign-asterisk (-*), 274, 281
  minus sign-plus sign (-+), 273, 275, 281
  plus sign (+)
    libraries, combining and specifying, 275, 281
    object files, appending, 279–281
LIB options
  /NOEXTDICTIONARY (/NOE), 272
  /NOIGNORECASE (/NOI), 272
  /PAGESIZE, 271
Libraries
  See also LIB
  application import, 320
  automatic object-file processing, 228
  development, used in, 228
  dynamic link
    debugging, 210–211
    described, 315, 319
  extensions, 227
  import, 317–318
  load, 228
  regular, 229
  routines, binding, 342
  search paths, 233
  specifying
    LINK command line, 229
    LINK prompts, 230
    LINK response file, 232
  standard places, 233
Library manager. *See* LIB
LIBRARY statement, 324
Line numbers, in source-level debugging, 79

Line-number option, LINK, 244
/LINENUMBERS option, LINK, 244
LINK
  alignment types, 255
  CodeView, used with
    C example, 10
    FORTRAN example, 11
    Macro Assembler example, 16
  combine types, 256
  defaults
    command line, 229
    responses, 231
  environment variable, 254
  exit codes, 358
  exiting from, 226
  fatal error messages, 372
  file-name conventions, 227
  groups, 257
  nonfatal error messages, 382
  operation, 254
  running
    LINK command line, 226
    prompts, 230
    response file, 232
  temporary output file, 235, 251
  terminating, 226
LINK options
  abbreviations, 236
  /ALIGNMENT (/A), 237
  /BATCH (/B), 237
  batch-file mode, running in, 237
  case sensitivity, 246
  /CODEVIEW (/CO), 238
  compatibility, preserving, 246
  /CPARMAXALLOC (/CP), 239
  data loading, 240
  debugging, 238
  default libraries, ignoring, 234
  /DOSSEG (/DO), 239
  /DSALLOCATE (/DS), 240
  environment variable, using, 254
  executable files
    loading, 243
    packing, 241
  /EXEPACK (/E), 241
  /FARCALLTRANSLATION (/F), 241
  /HELP (/HE), 242
  /HIGH (/HI), 240, 243
  /INCREMENTAL (/INC), 243, 264
  /INFORMATION (/INF), 243
LINK options (*continued*)
  line numbers, displaying, 244
  /LINENUMBERS (/LI), 244
  LINK command line, specifying on, 235
  LINK prompts, responding to, 236
  linker prompting, preventing, 237
  Listing, 242
  /MAP (/M), 229, 245
  map file, 229, 245
  /NOD, object files, used with, 234, 245
  /NOEXTDICTIONARY (/NOE), 245
  /NOFARCALLTRANSLATION (/NOF), 246
  /NOIGNORECASE (/NOI), 246
  /NONULLDOSSEG (/NON), 247
  /NOPACKCODE (/NOP), 247
  numerical arguments, 237
  ordering segments, 239
  overlay interrupt, setting, 247, 259
  /OVERLAYINTERRUPT (/O), 247, 259
  /PACKCODE (/PACKC), 248–249
  /PADCODE (/PADC), 249, 263
  /PADDATA (/PADD), 250, 263
  paragraph space, allocating, 239
  /PAUSE (/PAU), 251
  pausing, 251
  process information, displaying, 243
  producing a .COM file, 238
  /QUICKLIB (/Q), 251
  /SEGMENTS (/SE), 252
  segments, 252
  /STACK (/ST), 253
  stack size, setting, 253
  /WARNFIXUP (/W), 253
Linker utility. *See* LINK
Listing files, LIB, 279, 282
Load
  libraries, LINK command line, 228
  menu selection, 97
  size, 309
Local variables, 8, 142
Logical error, 8
Logical operator, FORTRAN, 69
Logical segment, defined, 262
Long reals
  dumping, 123
  entering with CodeView, 183
Loops
  tracepoints, used with, 152
  watchpoints, used with, 148
.LT. operator, 69
Lvalue, 149

M

M (Move Memory command), 185
/M option
  Codeview, 29
  LINK, 245
/m option, BIND, 344
Macro Assembler
  assembling and linking, 16
  older versions, with CodeView, 30–32
Macros
  $$@ macro, 295, 297
  $* macro, 295
  $** macro, 295
  $? macro, 295, 297
  $@ macro, 295, 297
  $< macro, 295, 299
  in C programs, 9
  NMAKE
    $**, 295
    $?, 297
    $@, 297
    $<, 299
    AS, 295
    CC, 295
    defining, 286, 293
    MAKE, 296
    MAKEFLAGS, 296
    precedence of definitions, 297
    predefined, 295
    special characters in, 296
    substitution, 294
    undefined, 300
    using, 293
MAKEFILE, 286–287
MAKEFLAGS macro, 296, 301
Map files
  creating, 244–245
  extensions, 227, 229, 244
  frame numbers, obtaining, 255
  /MAP (/M) option, LINK, 229, 245
  naming with LINK, 229
/MAP option, LINK, 229, 245
/MAX option, EXEMOD, 308
Memory
  allocation, and EXEMOD, 310
  blocks
    copying, 185
    filling, 184
    moving, 185
Memory (*continued*)
  operators, 83
  release, 198
Memory model, defined, 262
Menu bar, 35
Menus, CodeView
  Calls
    Stack Trace command, 167
    using, 57–58
  defined, 35
  File
    DOS Shell, 47, 198
    Exit, 47
    Load, 163
    Quit, 192
  Help, 58
  keyboard, selection from, 40
  mouse, selection from, 45
  Options
    386 option, 56
    Bytes Coded, 55, 160
    Case Sense, 55
    Flip/Swap, 54–55
  Run
    Clear Breakpoints, 52, 136
    Execute, 51, 96
    Restart, 51, 97
    Start, 51, 97
  Search
    Find, 49, 196
    Label, 51, 197
    Next, 50, 197
    Previous, 50, 197
  View
    Assembly, 48, 160
    Mixed, 48
    Output, 49
    Registers, 49, 128, 187
    Source, 48, 160
  Watch
    Add Watch, 52, 143
    Delete All, 54
    Delete Watch, 53–54, 153
    Tracepoint, 53, 150
    Watchpoint, 53, 146
Message box, 35, 41, 45
/MIN option, EXEMOD, 308
Minimum allocation value, controlling, 308
Minus sign (–)
  FORTRAN, 69
  LIB command symbol, 273, 275, 280

Minus sign-asterisk (–*), LIB command symbol, 274, 281
Minus sign-plus sign (– +), LIB command symbol, 273, 275, 281
Mixed mode, 159–160
Mixed-language programming, CodeView, 16
Mnemonic keys, CodeView, 40
Module statements
  defined, 321
  listed, 321
  rules for, 322
Module-definition files
  described, 321
  import libraries and, 316–317
  OS/2 linker and, 225
  rules for, 322
  statements
    CODE, 325
    DATA, 327
    DESCRIPTION, 325
    EXETYPE, 339
    EXPORTS, 334
    HEAPSIZE, 337
    IMPORTS, 335
    LIBRARY, 324
    NAME, 323
    OLD, 338
    PROTMODE, 338
    REALMODE, 339
    SEGMENTS, 331
    STACKSIZE, 334
    STUB, 337
Modules
  defined, 262
  examination, 111
Monochrome adapter (MA), 22–24, 27
Mouse
  driver, 29
  ignore option, 29
  pointer, 35, 41
  selecting with, 41
Move Memory command, 185
MSC, 10

N

N (Radix command), 193
/n option, BIND, 343
/N option, NMAKE, 288
NAME statement, 323
Naming files, 228
NAN (not a number), 118
.NE. operator, 69
Negation operator, FORTRAN, 69
.NEQV. operator, 69
NMAKE
  command line, 286
  commands, specifying, 290
  dependency lines, 289
  dependents, specifying, 289
  description blocks, 289
  description file, described, 288
  double-colon (::) separator, 292
  error handling, 288, 291
  error messages, 405
  escape character, 290
  exit codes, 357, 359
  inference rules
    described, 297
    predefined, 299
    using, 297
  invoking, 285–286
  macro substitution, 299
  macros, listed, 296
  pseudotargets, 302, 305
  response-file generation, 304
  targets, 286, 289
  vs. previous versions, 305
NMAKE directives
  !CMDSWITCHES, 301
  described, 300
  !ELSE, 300
  !ENDIF, 300
  !ERROR, 300
  !IF, 300
  !IFDEF, 300
  !IFNDEF, 300
  !INCLUDE, 301
  listed, 300
  !UNDEF, 294, 300
NMAKE options, 287–288
NMI trapping, 25–26
/NOD option, LINK, 234, 245
/NODEFAULTLIBRARYSEARCH option, LINK, 234, 245
/NOE option, LINK, 245
/NOEXTDICTIONARY, LINK, 245
/NOF option, LINK, 246
/NOFARCALLTRANSLATION option, LINK, 246
/NOG option
  LIB, 272
  LINK, 246

/NOGROUPASSOCIATION option
LIB, 272
LINK, 246
/NOI option
LIB, 272
LINK, 246
/NOIGNORECASE option
LIB, 272
LINK, 246
/NON option, LINK, 247
Nonequivalence operator, FORTRAN, 69
Nonfatal error messages, LINK, 382
/NONULLDOSSEG option, LINK, 247
/NOP option, LINK, 247
/NOPACKCODE option, LINK, 247
Not-equal-to operator, FORTRAN, 69
.NOT. operator, 69
NUL, 274
Number sign (#)
NMAKE comment character, 290
Tab Set command, 200
Numbers, floating point, 122–124

O

O (Option Command), 201
/o option, BIND, 343
/O option
CodeView, 29, 211
LINK, 247, 259
Object files
extensions, 227–228
naming, default, 228
NMAKE inference rules, used in, 299
object modules, difference from, 269
specifying
LINK command line, 228
LINK prompts, 230
LINK response file, 232
Object modules
defined, 270
libraries
deleting from, 273, 280
extracting and deleting from, 274, 281
including in, 273, 279–280
listing (LIB), 274, 282
object files, difference from, 270
Object ranges, arguments, used as, 82
Octal notation
BASIC, 75
C, 67
FORTRAN, 70
/Od compiler option, 8
OLD statement, 338
Operands, machine instruction, displayed by
CodeView, 128
Operators
BASIC, 73
C, 66
FORTRAN, 69
memory, CodeView, 85
Optimization, and CodeView, 8
Option command, 201
.OR. operator, 69
Ordinal position, 335
Output
Port command, 186
View menu selection, 49
Output screen, CodeView, 26, 195–196
/OVERLAYINTERRUPT option (/O), LINK, 247, 259
Overlays
interrupt number, setting, 247, 259
LINK, specifying, 258
overlay manager prompts, 259
restrictions, 259
search path, 259

P

/P option
CodeView, 29
NMAKE, 288
P (Program Step command), 91
Packed files, and CodeView, 5
Packing executable files, LINK, 241
/PADC option, LINK, 249, 263
/PADCODE option, LINK, 249, 263
/PADD option, LINK, 250, 263
/PADDATA option, LINK, 250, 263
Page size, library, 271, 283
/PAGESIZE option, LIB, 271, 283
Palette registers and CodeView, 30
Paragraph space, 239
Parameters, program, 18
Parentheses ( ), FORTRAN, 69
Pascal
compiling and linking, 14
include files, 13
programs, 13
Pass count, 134–135, 140
PATH command, CodeView, 17
/PAU option, LINK, 251
Pause command, 207
/PAUSE option, LINK, 251

Period (.)
  Current Location command, 166
  operator
    C, 66
    FORTRAN, 69–70
  regular expressions, used in, 354
PGDN key (scroll page down), 37
PGUP key (scroll page up), 36
Physical segment, defined, 262
Plus sign (+)
  LIB command symbol
    Intel, XENIX files, used with, 269
    libraries, combining and specifying, 273, 275, 281
    object files, appending, 279–281
    using, 273
  LINK command symbol, 231–232
  operator, FORTRAN, 69
Point, defined, 41
Pointer, mouse, 35, 41
Port Input command, 127
Port Output command, 186
Precedence of operators
  BASIC, 73
  C, 66
  FORTRAN, 69
.PRECIOUS pseudotarget, 303
Prefixes, with format specifiers, 101
printf type specifiers, 146, 149
PRIVATE combine type, 256
Procedure calls
  Stack Trace command, 167
  stepping over, 91
  tracing into, 89
Process, multiple, debugging, 211–212
Producing a .COM file (/BINARY option), 238
Program arguments, CodeView, 97
Program header, inspection of, 308
Program Step command
  F10 function key, 39, 60
  mouse, executing with, 44
  using, 90
Prompt, CodeView (>), 34, 61
Protected mode
  debugging in, 209–210
  described, 315
Protected-mode (80286) mnemonics, 162, 171
Protected-mode format, executable files, 344
PROTMODE statement, 338
Pseudotargets, 302, 305
PUBLIC combine type, 256
Public names. *See* Public symbols
Public symbols
  LIB, 274, 279, 282
  LINK, 245
  Macro Assembler, 32

Q

Q (Quit command), CodeView, 192
/Q option, NMAKE, 288
/QUICKLIB option, LINK, 251
Quotation marks (""), Pause command, 207

R

R (Register command), 128
/R option, NMAKE, 288
Radix
  command, using, 193
  current
    BASIC, 75
    C, 67–68
    effect on display, 57
    effect on unassemble, 166
    FORTRAN, 70–71
Ranges, arguments, used as, 82
Real mode, 225
REALMODE statement, 339
Redirection
  commands, 203
  start-up commands, used in, 25
Redraw command, 195
References
  long, 258
  near segment relative, 258
  near self relative, 258
  resolving, 245, 257
  short, 257
  unresolved, 257
Register
  argument, used as, 80
  command
    changing register values, 187
    displaying registers, 128
    F2 function key, 38, 59
    mouse, executing with, 44
    View menu selection, 49
  DS, described, 240

Register (*continued*)
variables, 66–67, 149
window, 34
Regular expressions
searches, used in, 50, 196
searching for, 50
specifying, 353
Regular libraries, LINK command line, 229
Relational expressions, 146
Relational operators
BASIC, 74
FORTRAN, 69
Relocation
information, 255
table, 310
Response files
LIB, 277
LINK, 232
NMAKE, 286, 304
Restart command
Run menu selection, 51
using, 96
Restrictions, CodeView, 5
Return codes. *See* Exit codes
ROM (read-only memory), 89
Routines
and CodeView, 167
arguments, value of, 166
calls to, 167
Run menu
Clear Breakpoints, 52, 136
Execute, 51, 96
Restart, 51, 97
Start, 51, 97
Run-time libraries, 269
Running
LIB
command line, 270
prompts, 276
response file, 277
LINK
command line, 226
prompts, 230

S

S (Set Mode command), 160
/S option
CodeView, 26
NMAKE, 288
Screen
buffer, 143
exchange
command, 195
F4 function key, 38, 59
method, 26
movement commands, 36
two, using, 22
Scroll bar, defined, 35
/SE option, LINK, 252
Search
command
menu selections, 49
regular expressions, used with, 353
using, 196
menu
Find, 49, 196
Label, 51, 197
Next, 50, 197
Previous, 50, 197
Search Memory command, 126
Search paths
dependents, 289
libraries, 233
overlays, 259
Segment attributes
CODE statement, 325
DATA statement, 327
SEGMENTS statement, 331
Segment, defined, 261
Segmented-executable file format, 344
Segments
alignment types, 255–256
class names, 256
class types, 256
combine types, 256
combining, 256
number allowed, 252
order, 239, 256
/SEGMENTS option, LINK, 252
SEGMENTS statement, 331
Semicolon (;)
LIB command symbol, 270–271, 277, 282
LINK command symbol, 229, 231–232
Separator line, 34
Sequential mode
CodeView, 33
redirection, used with, 205
starting, 30
Set Block, DOS function call (#4A), 198

Set Mode command
dialog command, 48
F3 function key, 38, 59
using, 159
View menu selection, 48
SETENV
error messages, 413
exit codes, 359
utility, 311
Shell Escape command
File menu selection, 47
using, 198
Short reals
Codeview, entering with, 182
dumping, 122–123
.SILENT pseudotarget, 303
Source
file, line-number arguments, used with, 79
mode, 159–160
Source-module files, location, 18
/ST option
EXEMOD, 308
LINK, 253
STACK
8087 register, 131
class name, 240
combine type, 256
/STACK option
EXEMOD, 308
LINK, 253
Stack size
controlling, 308
setting, 253
Stack Trace command
display contents, 57
using, 166
STACKSIZE statement, 334
Standard places, libraries, 233
Start-up
code, 20, 198
file configuration, CodeView, 17
Startup routine, 239
Statements, module, 321–322
Stopping
library manager, LIB, 271, 278
linker, LINK, 226
Strings
arguments
BASIC, 76
C, 68
FORTRAN, 71
Strings (*continued*)
concatenation, BASIC, 74
mnemonics, 173
operators, BASIC, 74
STUB statement, 337
Subprogram calls
Stack Trace command, 167
stepping over, 91
tracing into, 89
.SUFFIXES pseudotarget, 303
Swapping
disks, during linking, 251
screen, 26–27
Symbols
BASIC, 74
C, 67
examining, 111
FORTRAN, 70
underscore (_), in names, 67, 70
SYMDEB, 33
Syntax
CodeView summary, 191
error, 8
SYSTEM-REQUEST key, 39, 88

T

T (Trace command), 89
/T option
CodeView, 30
NMAKE, 288
Tab Set command, 200
Targets
defined, 289
macros for, 295
specifying
description blocks, 286, 289
multiple blocks, 292
Text strings, finding, 49, 196, 353
Threads, multiple, debugging, 213–215
TMP environment variable, used by LINK, 235
TOOLS.INI file
ignoring inference rules and macros in, 288
inference rules, used in, 298
precedence of macros, 297
Trace command
dialog command, 88
F8 function key, 39, 59
Tracepoint command
sequential mode, 60
setting, 148
Watch menu selection, 53

Tracepoint, defined, 148
Two-color graphics display, CodeView, 24
Type specifiers, 144, 146

U

U (Unassemble command), 161–162
!UNDEF directive, NMAKE, 300
Underscore (_), symbol names, 67, 70
Unsigned integers, dumping, 121
UP ARROW key (cursor up), 36
Utilities. *See* EXEMOD, LIB, LINK

V

V (View command), 163–164
/v option
  EXEHDR, 347
  /LINK, 265
Variables
  local, 8, 142
  special
    _edata, 240
    _end, 240
Verbose mode, 349
Video-display pages, 27
View
  command, 163–164
  menu
    Assembly, 48, 160
    Mixed, 48
    Output, 49
    Registers, 49
    Source, 48, 160
VM.TMP file, 235, 251

W

W commands
  Watch, 143–144
  Watch List, 60, 154
/W option
  CodeView, 30
  LINK, 253
WAIT instruction, 174
/WARNFIXUP option, LINK, 253
Watch
  command
    menu selections, 52
    sequential mode, 60
    setting Watch statement, 142
Watch (*continued*)
  expression statement, 143–144
  memory statement, 143–144
  menu
    Add Watch, 52
    Delete All, 54
    Delete Watch, 53–54
    Tracepoint, 53
    Watchpoint, 53
  statements
    commands, 141
    defined, 35
    deletion, 152
    listing, 154
    summary, 141
  window, 35, 142
Watch Delete All command, 54
Watch Delete command, 53–54, 152
Watch List command, 60, 154
Watchpoint command
  sequential mode, 60
  setting, 146
  Watch menu selection, 53
Watchpoint, defined, 146
Wild-card characters, 290
Window commands, 35, 61
Window mode
  CodeView, 33
  starting, 30
WO operator, 84
Words (units of memory), 84
WP (Watchpoint command), 147

X

X (Examine Symbols command), 111
/X option, NMAKE, 288

Y

Y (Watch Delete command), 153

Z

/Zd compiler option, 8
/Zi compiler option, 8

Microsoft Corporation
16011 NE 36th Way
Box 97017
Redmond, WA 98073-9717

Microsoft®

0689 Part No. 07824