

Environment and Tools

Microsoft®

C/C++

Microsoft® C/C++

Version 7.0

Environment and Tools

For MS-DOS® and Windows™ 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 and/or databases described in this document are furnished under a license agreement or nondisclosure agreement. The software and/or databases 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. The licensee may make one copy of the software for backup purposes. No part of this manual and/or databases may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or information storage and retrieval systems, for any purpose other than the licensee's personal use, without the express written permission of Microsoft Corporation.

©1991 Microsoft Corporation. All rights reserved.
Printed in the United States of America.

Microsoft, MS, MS-DOS, CodeView, QuickC, and XENIX are registered trademarks and QuickBasic, QBasic, QuickPascal, and Windows are trademarks of Microsoft Corporation.

OS/2 and Operating System/2 are registered trademarks licensed to Microsoft Corporation.

U.S. Patent No. 4955066

UNIX is a registered trademark of American Telephone and Telegraph Company.

Intel is a registered trademark of Intel Corporation.

BRIEF is a registered trademark of SDC Software Partners II L. P.

Contents Overview

Introduction	xxiii
---------------------------	-------

Part 1 The Programmer's WorkBench

Chapter 1	Introducing the Programmer's WorkBench	5
Chapter 2	Quick Start	9
Chapter 3	Managing Multimodule Programs	41
Chapter 4	User Interface Details.....	65
Chapter 5	Advanced PWB Techniques	85
Chapter 6	Customizing PWB.....	119
Chapter 7	Programmer's WorkBench Reference.....	141

Part 2 The CodeView Debugger

Chapter 8	Getting Started with CodeView	321
Chapter 9	The CodeView Environment.....	345
Chapter 10	Special Topics	377
Chapter 11	Using Expressions in CodeView.....	399
Chapter 12	CodeView Reference.....	417

Part 3 Compiling and Linking

Chapter 13	CL Command Reference.....	485
Chapter 14	Linking Object Files with LINK.....	561
Chapter 15	Creating Overlaid DOS Programs	597
Chapter 16	Creating Module-Definition Files.....	607
Chapter 17	Using EXEHDR.....	629

Part 4 Utilities

Chapter 18	Managing Projects with NMAKE.....	645
Chapter 19	Managing Libraries with LIB	697
Chapter 20	Creating Help Files with HELPMAKE	709
Chapter 21	Browser Utilities	731
Chapter 22	Using Other Utilities	743

Part 5 Using Help

Chapter 23	Using Help	755
------------	------------------	-----

Appendixes

Appendix A	Regular Expressions.....	777
Appendix B	Decorated Names	789

Appendix C	United States ASCII Character Chart (Code Page 437)	793
Appendix D	Multilingual ASCII Character Chart (Code Page 850).....	797
Appendix E	Key Codes	799
Glossary	803
Index	819

Contents

Introduction	xxiii
Scope and Organization of This Book	xxiv
Document Conventions	xxv

Part 1 The Programmer's WorkBench

Chapter 1	Introducing the Programmer's WorkBench	5
1.1	What's in Part 1	6
1.2	Using the Tutorial.....	6
	Conventions in the Tutorial	7
Chapter 2	Quick Start	9
2.1	The PWB Environment	9
	The Microsoft Advisor.....	10
	Entering Text.....	12
	Saving a File.....	13
	Indenting Text with PWB	13
	Copying, Pasting, and Deleting Text.....	15
	Opening an Existing File	18
2.2	Single-Module Builds.....	19
	Setting Build Options.....	19
	Setting Other Options	22
	Building the Program.....	23
	Fixing Build Errors	24
	Running the Program	27
2.3	Debugging the Program	28
	Using CodeView to Isolate an Error	29
	Testing Conditions in the Watch Window	32
2.4	Formatting Text.....	34
	Indenting Lines of Code.....	35
	Searching for Text.....	38
2.5	Where to Go from Here.....	39

Chapter 3	Managing Multimodule Programs	41
3.1	Multimodule Program Example.....	41
	Creating the Project.....	42
	Contents of a Project.....	43
	Dependencies in a Project.....	45
	Building a Multimodule Program.....	45
	Running the Program.....	46
	Project Maintenance.....	47
	Using Existing Projects.....	49
	Adding a File to the Project.....	50
	Changing Compiler Options.....	52
	Changing Options for Individual Modules.....	54
3.2	The Program Build Process.....	56
	Extending a PWB Project.....	58
	Using a Non-PWB Makefile.....	61
3.3	Where to Go from Here.....	63
Chapter 4	User Interface Details	65
4.1	Starting PWB.....	65
	From the Command Line.....	65
	Using the Windows Program Manager.....	66
	Using the Windows File Manager.....	67
4.2	The PWB Screen.....	67
4.3	PWB Menus.....	72
	File.....	72
	Edit.....	73
	Search.....	73
	Project.....	74
	Run.....	74
	Options.....	75
	Browse.....	76
	Window.....	77
	Help.....	78
4.4	Executing Commands.....	78
4.5	Choosing Menu Commands.....	78
	Shortcut Keys.....	79
	Buttons.....	80
	Dialog Boxes.....	80

Chapter 5	Advanced PWB Techniques.....	85
5.1	Searching with PWB	85
	Searching by Visual Inspection	86
	Using the Find Command.....	87
	Using Regular Expressions	90
	Using the Source Browser	96
5.2	Executing Functions and Macros	106
	Executing Functions and Macros by Name	108
5.3	Writing PWB Macros.....	109
	When Is a Macro Useful?	109
	Recording Macros	109
	Flow Control Statements	112
	User Input Statements	114
Chapter 6	Customizing PWB	119
6.1	Changing Key Assignments.....	119
6.2	Changing Settings.....	122
6.3	Customizing Colors.....	124
6.4	Adding Commands to the Run Menu	125
6.5	How PWB Handles Tabs.....	127
6.6	PWB Configuration.....	130
	Autoloading Extensions	131
	The TOOLS.INI File.....	131
	TOOLS.INI Statement Syntax.....	134
	Environment Variables	137
	Current Status File CURRENT.STS.....	138
	Project Status Files.....	138
Chapter 7	Programmer's WorkBench Reference.....	141
7.1	PWB Command Line	141
7.2	PWB Menus and Keys	142
7.3	PWB Default Key Assignments.....	146
	Note on Available Keys	149
7.4	PWB Functions.....	150
	Cursor-Movement Commands.....	154
7.5	Predefined PWB Macros.....	222

7.6	PWB Switches	263
	Extension Switches	265
	Filename-Parts Syntax	265
	Boolean Switch Syntax	266
	Browser Switches	309
	C and C++ Switches	310
	Help Switches	313

Part 2 The CodeView Debugger

Chapter 8	Getting Started with CodeView	321
8.1	Preparing Programs for Debugging	321
	General Programming Considerations	322
	Compiling and Linking	323
8.2	Debugging Strategies	325
	Identifying the Bug	325
	Locating the Bug	326
8.3	Setting up CodeView	327
	CodeView Files	328
8.4	Configuring CodeView with TOOLS.INI	329
	CodeView TOOLS.INI Entries	330
8.5	Memory Management and CodeView	336
8.6	The CodeView Command Line	336
	Leaving CodeView	337
	Command-Line Options	338
8.7	The CURRENT.STS State File	343
Chapter 9	The CodeView Environment	345
9.1	The CodeView Display	345
	The Menu Bar	346
	The Window Area	346
	The Status Bar	347
9.2	CodeView Windows	347
	How to Use CodeView Windows	347
	The Source Windows	350
	The Watch Window	350
	The Command Window	351
	The Local Window	354

The Register Window	354
The 8087 Window.....	355
The Memory Windows	356
The Help Window	357
9.3 CodeView Menus	358
The File Menu	358
The Edit Menu	360
The Search Menu	361
The Run Menu	362
The Data Menu.....	364
The Options Menu	368
The Calls Menu.....	372
The Windows Menu.....	373
The Help Menu	374
Chapter 10 Special Topics	377
10.1 Debugging in Windows.....	377
Comparing CVW with CV	377
Preparing to Run CVW	378
Starting a Debugging Session.....	378
CVW Commands	382
CVW Debugging Techniques.....	386
10.2 Debugging P-Code	389
Requirements	389
Preparing Programs.....	390
P-Code Debugging Techniques	391
P-Code Debugging Limitations	392
10.3 Remote Debugging	393
Requirements	393
Remote Monitor Command-Line Syntax	396
Starting a Remote Debugging Session	397
Chapter 11 Using Expressions in CodeView	399
11.1 Common Elements	399
Line Numbers.....	400
Registers.....	400
Addresses	401
Address Ranges.....	402
11.2 Choosing an Expression Evaluator	403

11.3	Using the C and C++ Expression Evaluators	404
	Additional Operators	405
	Unsupported Operators	405
	Restrictions and Special Considerations	405
	The Context Operator	406
	Numeric Constants	407
	String Literals	408
	Symbol Formats	408
11.4	Using C++ Expressions	409
	Access Control	409
	Ambiguous References	410
	Inheritance	410
	Constructors, Destructors, and Conversions	410
	Overloading	411
	Operator Functions	412
11.5	Debugging Assembly Language	412
	Memory Operators	412
	Register Indirection	414
	Register Indirection with Displacement	414
	Address of a Variable	414
	PTR Operator	414
	Strings	415
	Array and Structure Elements	415
Chapter 12	CodeView Reference	417
12.1	Command-Window Command Format	417
12.2	CodeView Expression Reference	417
12.3	CodeView Command Overview	422
12.4	CodeView Command Reference	424

Part 3 Compiling and Linking

Chapter 13	CL Command Reference	485
13.1	The CL Command Line	485
13.2	How the CL Command Works	486
13.3	CL Options	488
	/A Options (Memory Models)	488
	/batch (Compile in Batch Mode)	490
	/Bm (Increasing Compiler Capacity)	490

/c (Compile Without Linking)	491
/C (Preserve Comments During Preprocessing).....	491
/D (Define Constants and Macros)	491
/E (Copy Preprocessor Output to Standard Output)	493
/EP (Copy Preprocessor Output to Standard Output).....	494
/F (Set Stack Size).....	494
/f (Fast Compile)	494
/Fo, /Fe, /Fs, /Fa, /Fl, /Fc, /Fm, /Fp, /Fr, /FR (Set Alternate Output Files)....	495
/FP Options (Select Floating-Point-Math Package)	508
/G0, /G1, /G2, /G3, /G4 (Generate Processor-Specific Instructions).....	514
/GA, /GD (Optimize Entry/Exit Code for Protected-Mode Windows).....	515
/GE (Customize Windows Entry/Exit Code).....	515
/Gc, /Gd (Use FORTRAN/Pascal or C Calling Convention).....	516
/Ge, /Gs (Turn Stack Checking On or Off)	518
/Gr (Register Calling Convention)	520
/Gn (Remove P-Code Native Entry Points).....	520
/Gp (Specifying Entry Tables).....	521
/Gq (Real-Mode Windows Entry/Exit Code)	521
/Gt (Set Data Threshold).....	522
/Gw, /GW (Generate Entry/Exit Code for Real-Mode Windows Functions)	522
/Gx (Assume That Data Is Near)	523
/Gy (Enable Function-Level Linking)	524
/H (Restricts Length of External Names)	525
/HELP (List the Compiler Options).....	525
/I (Search Directory for Include Files).....	525
/J (Change Default char Type).....	526
/Ld, /Lw (Control Library Selection).....	527
/link (Linker-Control Options).....	527
/Ln (Link Without C Run-Time Startup Code)	528
/Lr (Real Mode Default Library)	528
/MA (Macro Assembler Options).....	528
/Mq (QuickWin Support).....	528
/ND, /NM /NQ, /NT, /NV (Name the Data or Code Segments)	528
/nologo (Suppress Display of Sign-On Banner)	530
/O Options (Optimize Program)	530
/P (Create Preprocessor-Output File).....	540
/qc (Quick Compile)	540
/Sl, /Sp, /Ss, /St (Source-Listing Format Options)	541
/Tc, /Tp, Ta (Specify C, C++ Source File, or Assembly Language).....	541
/U, /u (Remove Predefined Names).....	542

/V (Set Version String)	544
/W, /w (Set Warning Level).....	544
/X (Ignore Standard Include Directory).....	545
/Fp (Specify Precompiled Header Filename)	546
/Yc, /Yd, /Yu (Precompiled Header Options)	546
/Ze, /Za (Enable or Disable Language Extensions)	550
/Zc (Specify Pascal Naming)	552
/Zg (Generate Function Prototypes).....	552
/Zi, /Zd (Compile for Debugging)	553
/Zl (Remove Default-Library Name from Object File).....	553
/Zp (Pack Structure Members).....	554
/Zf (Accept __far Keyword).....	555
/Zn (Turn Off SBRPACK Utility)	555
/Zr (Check Pointers).....	556
/Zs (Check Syntax Only).....	557
Specifying Options with the CL Environment Variable	557

Chapter 14 Linking Object Files with LINK..... 561

14.1 New Features	561
14.2 Overview.....	563
14.3 LINK Output Files.....	563
14.4 LINK Syntax and Input	564
The objfiles Field	565
The exe file Field.....	566
The mapfile Field	567
The libraries Field	567
The deffile Field.....	570
Examples	571
14.5 Running LINK.....	572
Specifying Input with LINK Prompts.....	572
Specifying Input in a Response File	573
14.6 LINK Options	575
Specifying Options.....	575
The /ALIGN Option.....	576
The /BATCH Option.....	576
The /CO Option.....	577
The /CPARM Option	577
The /DOSSEG Option.....	578
The /DSALLOC Option.....	579
The /DYNAMIC Option	579

The /EXEPACK Option.....	580
The /FARCALL Option.....	580
The /HELP Option.....	581
The /HIGH Option.....	581
The /INFO Option.....	582
The /LINE Option.....	582
The /MAP Option.....	583
The /NOD Option.....	583
The /NOE Option.....	584
The /NOFARCALL Option.....	584
The /NOGROUP Option.....	584
The /NOI Option.....	585
The /NOLOGO Option.....	585
The /NONULLS Option.....	585
The /NOPACKC Option.....	586
The /NOPACKF Option.....	586
The /OLDOVERLAY Option.....	586
The /ONERROR Option.....	586
The /OV Option.....	587
The /PACKC Option.....	587
The /PACKD Option.....	588
The /PACKF Option.....	589
The /PAUSE Option.....	589
The /PM Option.....	590
The /Q Option.....	590
The /r Option.....	591
The /SEG Option.....	591
The /STACK Option.....	592
The /TINY Option.....	592
The /W Option.....	593
The /? Option.....	593
14.7 Setting Options with the LINK Environment Variable.....	593
Setting the LINK Environment Variable.....	593
Behavior of the LINK Environment Variable.....	594
Clearing the LINK Environment Variable.....	594
14.8 LINK Temporary Files.....	595
14.9 LINK Exit Codes.....	596

Chapter 15	Creating Overlaid DOS Programs	597
15.1	Overview	597
15.2	How to Create an Overlaid Program	598
	Compiling for Overlays	599
	Creating the Module-Definition File	600
	Linking the Overlaid Program	601
15.3	How MOVE Works.....	602
	Memory Allocation	602
	Limits and Requirements	603
15.4	Dynamic and Static Overlays	604
	Specifying Overlays on the Command Line	604
	Using the Static Overlay Manager	605
	Advantages of MOVE.....	605
Chapter 16	Creating Module-Definition Files	607
16.1	New Features	607
	Overlays	607
	DOS Programs	607
	Statements	608
16.2	Overview.....	608
16.3	Module Statements	609
	Syntax Rules.....	610
	Reserved Words	611
16.4	The NAME Statement	611
16.5	The LIBRARY Statement	612
16.6	The DESCRIPTION Statement.....	613
16.7	The STUB Statement.....	614
16.8	The APPLOADER Statement	615
16.9	The EXETYPE Statement	615
16.10	The PROTMODE Statement.....	616
16.11	The REALMODE Statement.....	617
16.12	The STACKSIZE Statement	617
16.13	The HEAPSIZE Statement	617
16.14	The CODE Statement	618
16.15	The DATA Statement.....	618
16.16	The SEGMENTS Statement.....	619
16.17	CODE, DATA, and SEGMENTS Attributes.....	620
16.18	The OLD Statement.....	622

16.19	The EXPORTS Statement	623
16.20	The IMPORTS Statement	624
16.21	The FUNCTIONS Statement	625
16.22	The INCLUDE Statement	627

Chapter 17 Using EXEHDR 629

17.1	Running EXEHDR	629
	The EXEHDR Command Line	629
	EXEHDR Options.....	630
17.2	Executable-File Format	631
17.3	EXEHDR Output: DOS Executable File	632
17.4	EXEHDR Output: Segmented Executable File	634
	DLL Header Differences.....	635
	Segment Table.....	635
	Exports Table	636
17.5	EXEHDR Output: Verbose Output.....	637
	DOS Header Information	637
	New .EXE Header Information	637
	Tables	638

Part 4 Utilities

Chapter 18 Managing Projects with NMAKE 645

18.1	New Features	645
18.2	Overview	646
18.3	Running NMAKE.....	647
	Command-Line Options	647
	NMAKE Command File.....	650
	The TOOLS.INI File.....	652
18.4	Contents of a Makefile	653
	Using Special Characters as Literals	653
	Wildcards	653
	Comments	654
	Long Filenames.....	654
18.5	Description Blocks	655
	Dependency Line	655
	Targets.....	656
	Dependents.....	659

18.6	Commands	660
	Command Syntax	660
	Command Modifiers	661
	Exit Codes from Commands	662
	Filename-Parts Syntax	663
	Inline Files	664
18.7	Macros	667
	User-Defined Macros	668
	Using Macros	671
	Special Macros	671
	Substitution Within Macros	677
	Substitution Within Predefined Macros	678
	Environment-Variable Macros	678
	Inherited Macros	679
	Precedence Among Macro Definitions	680
18.8	Inference Rules	680
	Inference Rule Syntax	681
	Inference Rule Search Paths	682
	User-Defined Inference Rules	682
	Predefined Inference Rules	684
	Inferred Dependents	685
	Precedence Among Inference Rules	686
18.9	Directives	687
	Dot Directives	687
	Preprocessing Directives	688
18.10	Sequence of NMAKE Operations	692
18.11	A Sample NMAKE Makefile	694
18.12	NMAKE Exit Codes	696
Chapter 19	Managing Libraries with LIB	697
19.1	Overview	697
19.2	Running LIB	698
	The LIB Command Line	698
	LIB Command Prompts	698
	The LIB Response File	699
19.3	Specifying LIB Fields	699
	The Library File	700
	LIB Options	700

LIB Commands	702
The Cross-Reference Listing	705
The Output Library	706
Examples	707
19.4 LIB Exit Codes	708
Chapter 20 Creating Help Files with HELPMAKE	709
20.1 Overview	710
20.2 Running HELPMAKE	711
Encoding	711
Decoding	713
Getting Help	714
Other Options	715
20.3 Source File Formats	716
20.4 Elements of a Help Source File	716
Defining a Topic	716
Creating Links to Other Topics	717
Formatting Topic Text	721
Dot Commands	722
20.5 Other Help Text Formats	725
Rich Text Format	725
Minimally Formatted ASCII	728
20.6 Context Prefixes	729
Chapter 21 Browser Utilities	731
21.1 Overview of Database Building	732
Preparing to Build a Database	732
How BSCMAKE Builds a Database	732
Methods for Increasing Efficiency	733
21.2 BSCMAKE	734
System Requirements for BSCMAKE	734
The BSCMAKE Command Line	735
BSCMAKE Options	736
Using a Response File	738
BSCMAKE Exit Codes	739
21.3 SBRPACK	739
Overview of SBRPACK	739
The SBRPACK Command Line	740
SBRPACK Exit Codes	741

Chapter 22	Using Other Utilities.....	743
22.1	CVPACK.....	743
	Overview of CVPACK.....	744
	The CVPACK Command Line.....	744
	CVPACK Exit Codes.....	745
22.2	IMPLIB.....	745
	About Import Libraries.....	746
	The IMPLIB Command Line.....	746
	Options.....	747
22.3	RM, UNDEL, and EXP.....	747
	Overview of the Backup Utilities.....	747
	The RM Utility.....	748
	The UNDEL Utility.....	749
	The EXP Utility.....	750

Part 5 Using Help

Chapter 23	Using Help.....	755
23.1	Structure of the Microsoft Advisor.....	755
23.2	Navigating Through the Microsoft Advisor.....	756
	Using the Help Menu.....	757
	Using the Mouse and the F1 Key.....	757
	Using Hyperlinks.....	759
	Using Help Windows and Dialog Boxes.....	760
	Accessing Different Types of Information.....	762
	Using Different Help Screens.....	764
23.3	Using Help in PWB.....	765
	Opening a Help File.....	765
	Global Search.....	766
23.4	Using QuickHelp.....	768
	Using the /Help Option.....	768
	Using the QH Command.....	768
23.5	Managing Help Files.....	771
	Managing Many Help Files.....	772

Appendixes

Appendix A	Regular Expressions	777
A.1	Regular-Expression Summaries	778
A.2	UNIX Regular-Expression Syntax	781
A.3	Tagged Regular Expressions	782
	Tagged Expressions in Build:Message	784
A.4	Justifying Tagged Expressions.....	785
A.5	Predefined Regular Expressions	785
A.6	Non-UNIX Regular-Expression Syntax.....	786
	Non-UNIX Matching Method	788
Appendix B	Decorated Names.....	789
B.1	Overview	789
	Format of a Decorated Name.....	789
	Viewing Decorated Names	790
B.2	Getting and Specifying a Decorated Name.....	790
Appendix C	United States ASCII Character Chart (Code Page 437)	793
Appendix D	Multilingual ASCII Character Chart (Code Page 850).....	797
Appendix E	Key Codes.....	799
Glossary		803
Index		819

Figures and Tables

Figures

Figure 2.1	PWB Display	10
Figure 2.2	PWB Build Options.....	22
Figure 3.1	The COUNT Project	42
Figure 3.2	The PWB Build Process.....	57
Figure 4.1	User Interface Elements	68
Figure 4.2	Window Elements	69
Figure 4.3	Status Bar Elements	70
Figure 4.4	PWB Menu Elements	71
Figure 4.5	Dialog Box Elements	81
Figure 4.6	Key Box and Check Box.....	82
Figure 5.1	Regular Expression Example	91
Figure 5.2	Complex Regular Expression Example	92
Figure 6.1	How PWB Displays Tabs	128
Figure 7.1	Arranged Windows	227
Figure 7.2	Cascaded Windows	231
Figure 7.3	Vertical Tiling	299
Figure 7.4	Horizontal Tiling	300
Figure 9.1	CodeView Display	346
Figure 17.1	Format for a Segmented Executable File.....	632
Figure 18.1	NMAKE Description Block.....	655
Figure 23.1	Microsoft Advisor Global Contents Screen.....	756
Figure 23.2	Microsoft Advisor Global Index Screen.....	757
Figure 23.3	Help on the PWB Cut Command.....	758
Figure 23.4	Help for printf in a PWB Window	763
Figure 23.5	PWB Index	765
Figure 23.6	The QuickHelp Window	770

Tables

Table 7.1	File Menu and Keys	143
Table 7.2	Edit Menu and Keys.....	143
Table 7.3	Search Menu and Keys.....	144
Table 7.4	Project Menu and Keys	144
Table 7.5	Run Menu and Keys.....	144
Table 7.6	Browse Menu and Keys	145
Table 7.7	Window Menu and Keys.....	145
Table 7.8	Help Menu and Keys.....	146
Table 7.9	PWB Default Key Assignments.....	146
Table 7.10	PWB Functions	151

Table 7.11	Cursor-Movement Commands.....	155
Table 7.12	PWB Macros	222
Table 7.13	PWB Color Names.....	271
Table 7.14	PWB Color Values.....	273
Table 8.1	CodeView TOOLS.INI Entries	330
Table 8.2	CodeView Command-Line Options.....	338
Table 9.1	Moving Around with the Keyboard.....	349
Table 11.1	Registers.....	401
Table 12.1	Register Names	419
Table 12.2	CodeView Command Summary	422
Table 13.1	Memory Models.....	488
Table 13.2	Customized Memory Model Codes	490
Table 13.3	Optional File Types.....	495
Table 13.4	Floating-Point Options.....	508
Table 13.5	CL Options and Default Libraries	514
Table 13.6	Using the check_stack Pragma	519
Table 13.7	Segment-Naming Conventions	529
Table 13.8	Inline Expansion Control	532
Table 13.9	Predefined Names	543
Table 13.10	Using the pack Pragma	555
Table 16.1	Module Statements.....	609
Table 18.1	Predefined Inference Rules.....	684
Table 18.2	Binary Operators for Preprocessing.....	691
Table 20.1	Formatting Attributes.....	721
Table 20.2	Dot Commands	722
Table 20.3	RTF Formatting Codes	727
Table 20.4	Microsoft Product Context Prefixes	730
Table 20.5	Standard h. Contexts	730
Table A.1	UNIX Regular-Expression Summary	778
Table A.2	UNIX Predefined Expressions.....	778
Table A.3	CodeView Regular Expressions.....	779
Table A.4	Non-UNIX Regular-Expression Summary.....	780
Table A.5	Non-UNIX Predefined Expressions	780
Table A.6	UNIX Regular-Expression Syntax	781
Table A.7	Predefined Regular Expressions and Definitions	785
Table A.8	Non-UNIX Regular Expression Syntax.....	786

Introduction

Microsoft C/C++ includes a full set of development tools—editor, compiler, linker, debugger, and browser—for writing, compiling, and debugging your programs. You can work within the Microsoft Programmer’s WorkBench (PWB) integrated environment, or you can use the tools separately to develop your programs.

Environment and Tools describes the following development tools:

- The Programmer’s WorkBench (PWB). PWB is a comprehensive tool for application development. Within its environment is everything you need to create, build, browse, and debug your programs. Its macro language gives you control over not only editing but also build operations and other PWB functions.
- The Microsoft CodeView debugger. This is a diagnostic tool for finding errors in your programs. Two versions of CodeView are described: one for DOS programs and one for Microsoft Windows. Each CodeView version has specialized commands for its operating environment, as well as other commands for examining code and data, setting breakpoints, and controlling your program’s execution.
- CL, the Microsoft C/C++ Compiler. CL compiles and links your source code.
- LINK, the Microsoft Segmented Executable Linker. The linker combines object files and libraries into an executable file, either an application or a dynamic-link library (DLL).
- EXEHDR, the Microsoft EXE File Header Utility. EXEHDR displays and modifies the contents of an executable-file header.
- NMAKE, the Microsoft Program Maintenance Utility. NMAKE simplifies project maintenance. Once you specify which project files depend on others, you can use NMAKE to automatically execute the commands that will update your project when any file has changed.
- LIB, the Microsoft Library Manager. LIB creates and maintains standard libraries. With LIB, you can create a library file and add, delete, and replace modules.
- HELPMMAKE, the Microsoft Help File Maintenance Utility. HELPMMAKE creates and maintains Help files. You can use HELPMMAKE to create a Help file or to customize the Microsoft Help files.
- BSCMAKE, the Microsoft Browser Database Maintenance Utility, and SBRPACK, the Microsoft Browse Information Compactor. BSCMAKE creates browser files for use with the PWB Source Browser. SBRPACK compresses the files that are used by BSCMAKE.

Environment and Tools also describes these special purpose utilities:

- CVPACK, the Microsoft Debugging Information Compactor. CVPACK compresses the size of debugging information in an executable file.
- IMPLIB, the Microsoft Import Library Manager. IMPLIB creates an import library that resolves external references from a Windows application to a DLL.
- RM, the Microsoft File Removal Utility; UNDEL, the Microsoft File Undelete Utility; and EXP, the Microsoft File Expunge Utility. These utilities manage, delete, and recover backup files.

Scope and Organization of This Book

This book has five parts and five appendixes to give you complete information about PWB, CodeView, CL, and the utilities included with C/C++.

Part 1 is a brief PWB tutorial and comprehensive reference. The first three chapters introduce PWB and provide a tutorial that describes the features of the integrated environment and how to use them. Chapters 4, 5, and 6 contain detailed information on the interface, advanced PWB techniques, and customization. Chapter 7 contains a complete reference to PWB's default keys and all functions, predefined macros, and switches.

Part 2 provides full information on the Microsoft CodeView debugger. Chapter 8 tells how to prepare programs for debugging, how to start CodeView, and how to customize CodeView's interface and memory usage. Chapter 9 describes the environment, including the CodeView menu commands and the format and use of each CodeView window. Chapter 10 explains how to use expressions, including the C and C++ expression evaluators. Chapter 11 describes techniques for debugging Windows programs and p-code. Chapter 12 contains a complete reference to CodeView commands.

The chapters in Parts 3 and 4 describe the compiler and utilities. These chapters are principally for command-line users. Even if you're using PWB, however, you may find the detailed information in Parts 3 and 4 helpful for a better understanding of how each tool contributes to the program development process.

Part 3 provides information about compiling and linking your program. Chapter 13 describes the command-line syntax and options for the CL compiler. LINK command-line syntax and options are covered in Chapter 14. Chapter 15 describes how to overlay a DOS program. The contents and use of module-definition files are explained in Chapter 16. Chapter 17 describes how to use EXEHDR to examine the file header of a program.

Part 4 presents the other utilities. NMAKE, the utility for automating project management, is described in Chapter 18. Chapter 19 covers LIB, the utility to use in

managing standard libraries. Procedures for using HELPMAKE to create and maintain Help files are in Chapter 20. The tools for creating a browser database are discussed in Chapter 21. Finally, Chapter 22 describes how to use the following special purpose utilities: CVPACK, IMPLIB, RM, UNDEL, and EXP.

Part 5 presents the Microsoft Advisor Help system and the QuickHelp program. It describes the structure of the Help files, how to navigate through the Help system, and how to manage Help files.

The appendixes provide supplementary information. Appendix A describes regular expressions for use in PWB and CodeView. Appendix B explains a procedure for getting the decorated name of a C++ function. Appendix C lists United States ASCII codes, Appendix D lists multilingual ASCII codes, and Appendix E lists key codes.

Document Conventions

This book uses the following typographic conventions:

Examples	Description
README.TXT, COPY, LINK, /CO	Uppercase (capital) letters indicate filenames, DOS commands, and the commands to run the tools. Uppercase is also used for command-line options, unless the option must be lowercase.
printf, IMPORT	Bold letters indicate keywords, library functions, reserved words, and CodeView commands. Keywords are required unless enclosed in double brackets as explained below.
<i>expression</i>	Words in italic are placeholders for information that you must supply (for example, a function argument).
[[<i>option</i>]]	Items inside double square brackets are optional.
{ <i>choice1</i> <i>choice2</i> }	Braces and a vertical bar indicate a choice between two or more items. You must choose one of the items unless all the items are also enclosed in double square brackets.
CL ONE.C TWO.C	This font is used for program examples, user input, program output, and error messages within the text.
Repeating elements...	Three horizontal dots following an item indicate that more items having the same form may follow.
while() { . . . }	Three vertical dots following a line of code indicate that part of the example program has intentionally been omitted.

F1, ALT+A

Small capital letters indicate the names of keys and key sequences, such as ENTER and CTRL+C. A plus (+) indicates a combination of keys. For example, CTRL+E means to hold down the CTRL key while pressing the E key.

The cursor-movement keys on the numeric keypad are called ARROW keys. Individual ARROW keys are referred to by the direction of the arrow on the top of the key (LEFT, RIGHT, UP, DOWN). Other keys are referred to by the name on the top of the key (PGUP, PGDN).

Arg Meta Delete
(ALT+A ALT+A SHIFT+DEL)

A bold series of names followed by a series of keys indicates a sequence of PWB functions that you can use in a macro definition, type in a dialog box, or execute directly by pressing the sequence of keys. In this book, these keys are the default keys for the corresponding functions. Some functions are not assigned to a key, and the word “Unassigned” appears in the place of a key. In PWB Help, the current key that is assigned to the function is shown.

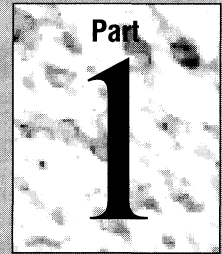
“defined term”

Quotation marks usually indicate a new term defined in the text.

dynamic-link library (DLL)

Acronyms are usually spelled out the first time they are used.

The Programmer's WorkBench



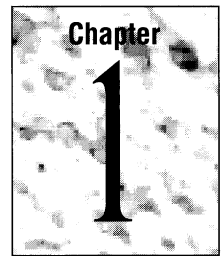
Chapter 1	Introducing the Programmer's WorkBench	5
2	Quick Start	9
3	Managing Multimodule Programs.....	41
4	User Interface Details	65
5	Advanced PWB Techniques.....	85
6	Customizing PWB	119
7	Programmer's WorkBench Reference	141

The Programmer's WorkBench

Microsoft C/C++ provides you with a comprehensive tool for application development. The Programmer's WorkBench (PWB) integrated environment provides an editor, compiler, linker, debugger, and browser.

Chapter 1 presents an overview of PWB and describes how to use the tutorial. Chapter 2 guides you through the tutorial. Chapter 3 continues the tutorial and explains how to build a multimodule project. Chapter 4 explains the PWB user interface—the screen elements, menus, commands, and other features. Chapter 5 describes some advanced features of the PWB environment, including how to use search techniques, regular expressions, functions, macros, and the Source Browser. Chapter 6 explains how to customize PWB. Chapter 7 contains a reference of PWB menu command, functions, and switches.

Introducing the Programmer's WorkBench



The Microsoft Programmer's WorkBench (PWB) is a powerful tool for application development. PWB combines the following features:

- A full-featured programmer's text editor.
- An extensible "build engine" which allows you to compile and link your programs using the PWB environment. The build engine can be extended to support any programming tool.
- Error-message browsing. Once a build completes, you can step through the build messages, fixing errors in your source programs.
- A Source Browser. When working with large systems, it is often difficult to remember where program symbols are accessed and defined. The Source Browser maintains a database that allows you to go quickly to where a given variable, function, type, class, or macro is defined or referenced.
- An extensible Help system. The Microsoft Advisor Help system provides a complete reference on using PWB, your programming language, and the other components of Microsoft C/C++. You can also write new Help files and seamlessly integrate them into the Help system to document your own library routines or naming conventions.
- A macro language that can control editing functions, program builds, and other PWB operations.

For increased flexibility, you can write extensions to PWB. These extensions can perform tasks that are inconvenient in the PWB macro language. For example, you can write extensions to perform file translations, source-code formatting, text justification, and so on. As with the macro language, PWB extensions have full access to most PWB capabilities. For information about how to write PWB extensions, see the Microsoft Advisor Help system (choose "PWB Extensions" from the main Help table of contents).

1.1 What's in Part 1

This part of the book introduces you to the fundamentals of PWB. Chapter 2, “Quick Start,” shows you how to use the PWB editor and build a simple single-module program from PWB. Chapter 3, “Managing Multimodule Programs,” expands upon the information you learned in Chapter 2. It teaches you how to build a more complicated program that consists of several modules. You should be able to work through these two chapters in less than three hours.

As you work through these chapters, you may want to refer to Chapter 4, “User Interface Details,” which explains options for starting PWB, briefly describes all of the menu commands, and summarizes how menus and dialog boxes work. The user interface information is presented in one chapter for easy access.

Chapter 5, “Advanced PWB Techniques,” shows how to use the PWB search facilities (including searching with regular expressions), how to use the Source Browser, how to execute functions and macros, and how to write PWB macros.

Chapter 6, “Customizing PWB,” describes how to redefine key assignments, change PWB settings, add commands to the PWB menu, and use the TOOLS.INI initialization file to store startup and configuration information for PWB.

Chapter 7, “PWB Reference,” contains an alphabetical reference to PWB menus, keys, functions, predefined macros, and switches. It contains the essential information you need to know to take the greatest advantage of PWB's richly customizable environment.

Chapters 4 and 5 are not as tutorial as Chapters 2 and 3. Chapters 6 and 7 describe advanced features that you probably don't need to learn right away. You may want to come back to these chapters after you are comfortable with PWB.

1.2 Using the Tutorial

You probably want to get right to work with Microsoft C/C++. The tutorial chapters 2 and 3 can help you become productive very quickly. To get the most out of this material, here are a few recommendations:

- Follow the steps presented in the tutorial. It is always tempting to explore the system and find out more about the product through independent research. However, just as programming requires an orderly sequence of steps, some aspects of PWB also require sequenced actions.
- If you complete a step and something seems wrong—for example, if your screen doesn't match what is in the book—back up and try to find out what's wrong. Troubleshooting tips will help you take corrective actions.

- When working through this tutorial, consider how you might use these techniques in your own work. PWB is like a full tool chest. You probably won't learn (or even want to learn) all of PWB's capabilities right away. But as time goes on, you'll have uses for many of the tools you don't use immediately.

Conventions in the Tutorial

To help you move through the tutorial quickly, there are two navigation aids: the tip and the procedure heading.

Tips

TIP Tips like this are useful tidbits of information, such as a keyboard shortcut.

Information that is handy but not essential is included in the left margin. Tips offer additional information to help you make the most efficient use of PWB. They should not be confused with margin summaries, which appear in other sections of this book and are used to summarize information presented in the text.

Procedure Headings

Procedure headings are denoted by a triangular symbol. These headings always precede a list of steps. For example:

► **To open a file:**

1. From the File menu, choose Open.
PWB displays the Open File dialog box.
2. In the File List list box, select the file that you want to open.
3. Choose OK.

In procedures, the heading gives you a capsule summary of what the steps will accomplish. Each numbered step is an action you take to complete the procedure. Some steps are followed by an explanation, an illustration, or both.

This chapter gets you started with PWB. You'll learn the basics by building and debugging a program that calculates payments on a loan given the principal, interest rate, and term.

To start PWB in Windows for this tutorial, click the PWB icon in the Microsoft C/C++ Program Group.

In DOS, type

PWB

at the prompt.

► **To leave PWB at any time:**

- From the File menu, choose Exit, or press ALT+F4.

2.1 The PWB Environment

If this is the first time you have used PWB, you see the menu bar, the status bar, and an empty desktop. If you have used PWB before, it opens the file you last worked with.

PWB uses a windowed environment to present information, get information from you, and allow you to edit programs. The environment has the following components:

- An editor for writing and revising programs
- A “build engine”—the part of PWB that helps you compile, link, and execute your programs from within the environment
- A source-code browser

- Commands for program execution and debugging
- The Microsoft Advisor Help system

The browser and the Help system are dynamically loaded extensions to the PWB platform. Microsoft languages and the utilities are also supported in PWB by extensions. Other extensions are available, such as the Microsoft Source Profiler. PWB presents all of these components through menus and dialog boxes.

Before continuing, look at the following figure, which introduces some parts of the PWB interface.

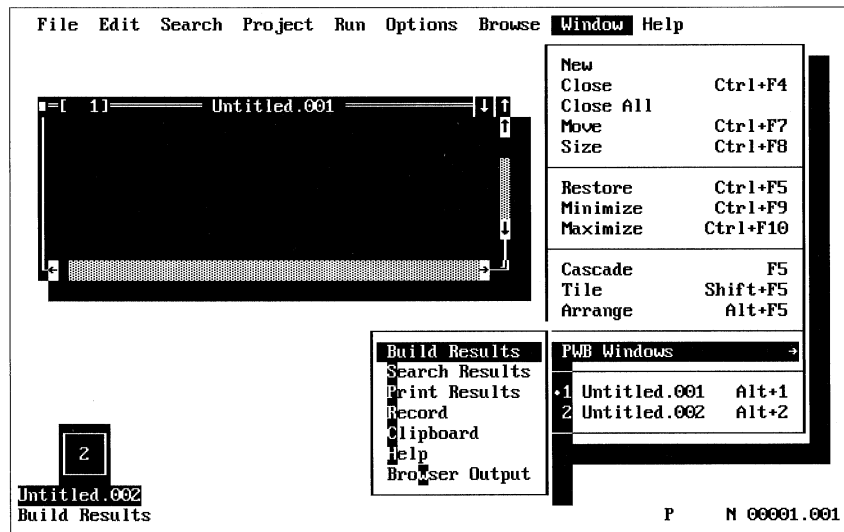


Figure 2.1 PWB Display

Chapter 4, “User Interface Details,” contains a thorough description of these elements and the rest of the PWB environment. Refer to this chapter when you need specific information about an unfamiliar interface element.

The Microsoft Advisor

PWB makes programming easier by providing the Microsoft Advisor Help system, which contains comprehensive information about:

- PWB editing functions
- PWB advanced features
- PWB menus and dialog boxes

- CodeView debugger
- C and C++ languages
- C and C++ compiler options
- C run-time library
- P-code
- Microsoft utilities (such as NMAKE, LINK, and so on)
- Windows API (application programming interface)
- Microsoft class libraries

TIP To get immediate help on any subject, point to the item in question and press the right mouse button.

The Advisor provides context-sensitive Help and general Help. Context-sensitive Help provides information about the menu, dialog box, or language element at the cursor. To see context-sensitive Help, press the F1 key. PWB displays the Help window to show the requested information. You can also get context-sensitive Help and more general Help by using the Help menu.

To answer questions of a less specific nature, you can access the Contents screen by choosing Contents from the Help menu or by pressing SHIFT+F1. From the Advisor contents, you can access Help on any other subject in the database.

► **To get started using the Microsoft Advisor:**

- From the Help menu, choose the Help on Help command.

Help on Help teaches you how to use the Microsoft Advisor Help system. For more information on using Help, see Chapter 23.

► **To close the Help window:**

- Click the upper-left corner of the Help window (the Close box), press ESC, choose Close from the File menu, or press CTRL+F4.

Note Click the Close box, choose Close from the File menu, or press CTRL+F4 to close any open window in PWB.

The following sections explain basic editing procedures. If you're already familiar with these, you can skip to "Opening an Existing File" on page 18.

Entering Text

In this section, you'll learn basic PWB procedures by entering a simple C program, ANNUITY.C.

TIP Press the highlighted key in a menu or command name to open the menu or execute the command.

► **To start a new file:**

1. Move the mouse cursor ("point") to the File menu on the menu bar and click the left button.
PWB opens the File menu.
2. Point to the New command and click the left button.

You can also do this from the keyboard:

1. Press ALT+F to open the File menu.
2. Press N to choose New.

PWB opens a window with the title `Untitled.001`.

Starting with your cursor in the upper-left corner of the edit window, type the following comments:

```
//  
// annuity.c - Program to generate a simple annuity table  
//
```

Your screen should appear as follows:

The screenshot shows a window titled "Untitled.001" with a menu bar containing "File", "Edit", "Search", "Project", "Run", "Options", "Browse", "Window", and "Help". The text entered in the window is:

```
File Edit Search Project Run Options Browse Window Help  
-I 11 Untitled.001  
//  
// annuity.c - Program to generate a simple annuity table  
//
```

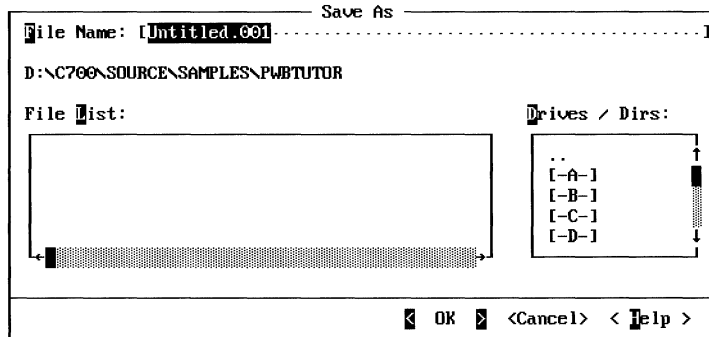
At the bottom of the window, there is a status bar with the text: "<General Help> <F1=Help> <Alt=Menu> MP 00003.003".

Saving a File

Now that you've entered some of your program, save your work before proceeding.

► **To save a file:**

- From the File menu, choose Save, or press SHIFT+F2.
PWB displays the Save As dialog box.



This dialog box has several options that you use to pass information to PWB. PWB indicates the active option—in this case, the File Name text box—by highlighting the area in which you can enter text. For more information about dialog boxes, see Chapter 4, “User Interface Details.”

Because you have not yet saved the file, it still has the name `Untitled.001`. Type `ANNUITY.C` in the File Name text box. Then click OK or press ENTER to save the file.

Note Now that you have named your file, choosing Save from the File menu does not bring up a dialog box. Your file is immediately saved to disk.

Indenting Text with PWB

Type the following program fragment:

```
#include <stdio.h>
#include <math.h>
main()
{
    float pv, rate, pmt, fv, ratepct;
    int nper, actnper;
```

```
// get input from the user
printf( "Enter present value:\n" );
scanf( "%f", &pv );
printf( "Enter interest rate in percent:\n" );
scanf( "%f", &rate );
printf( "Enter number of periods in years:\n" );
scanf( "%i", &nper );
}
```

Often you will add several lines indented to the same column. PWB saves you time by automatically indenting new lines when you press the ENTER key.

When the PWB C extension is loaded, PWB automatically indents new lines in C and C++ source files as appropriate for the C and C++ languages. When the language extension is not loaded (or the extension's autoindentation is turned off), PWB uses its default indentation rules as follows:

- If there is no line or a blank line immediately below the new line, PWB matches the indentation of the line above it.
- If there is a line immediately below the new line, PWB matches the indentation of the line below it.

You'll now type some text after the last **scanf** statement.

► **To insert space for a new line:**

1. Position the mouse cursor anywhere past the end of the **scanf** statement. Precise positioning of the cursor is not critical because (by default) PWB trims trailing spaces from the end of your lines.
2. Click the left mouse button.
3. Press ENTER to make a new line.

If you are in overtyping mode, change to insert mode by pressing the INS key. Otherwise, pressing ENTER simply moves the cursor to the beginning of the next line. PWB displays the letter O on the status bar and shows the cursor as an underscore to signal that you are in overtyping mode.

► **To insert the new line using the keyboard:**

1. Move the cursor to the **scanf** statement just above the closing brace by pressing the UP ARROW key.
2. Press END to move the cursor to the end of the line.
3. Press ENTER to make a new line.

Type the following lines:

```

ratepct = rate / 1200.0;
actnper = nper * 12;
// calculate the payment amount
pmt = pv * (ratepct / (1.0 - (1.0 /
    (pow((1.0 + ratepct), actnper )))));
printf( "Principal: %f\n", pv );
printf( "Interest rate: %f\n", rate );

```

TIP To move the cursor directly to the first column, press F9, HOME.

When you enter the forward slash for the first comment line (the third line of this section), PWB's automatic indentation feature positions the cursor in column 5. To move the cursor to column 1, use the LEFT ARROW key or the BACKSPACE key.

Copying, Pasting, and Deleting Text

The remainder of the program consists of the following `printf` statements. Don't type them in yet. You will copy and paste to enter these lines.

```

printf( "Number of years: %i\n", nper );
printf( "Monthly Payment: %f\n", pmt );
printf( "Total Payments: %f\n", pmt * nper * 12.0 );
printf( "Total Interest: %f\n", pmt * nper * 12.0 - pv );

```

Since these lines are similar, you can save time by typing only the first one, then copying and pasting text using PWB's clipboard (a temporary storage place for text).

► To copy and paste text:

1. Place the cursor on the line:

```
printf( "Interest rate: %f\n", rate );
```

TIP You can use the keys CTRL+INS for Copy and SHIFT+INS for Paste.

2. From the Edit menu, choose Copy. This action places the entire line on the clipboard for later reference.
3. To insert the copied line, choose Paste from the Edit menu.
4. Paste the same line three more times to create enough **printf** statements for the remainder of the program.

Your screen should look like the following figure:

```

File Edit Search Project Run Options Browse Window Help
D:\C700\SOURCE\SAMPLES\PWBTUTOR\amuity.c
// get input from the user
printf( "Enter present value:\n" );
scanf( "%f", &pv );
printf( "Enter interest rate in percent:\n" );
scanf( "%f", &rate );
printf( "Enter number of periods in years:\n" );
scanf( "%i", &nper );

ratepct = rate / 1200.0;
actnper = nper * 12;
// calculate the payment amount
pmt = pv * (ratepct / (1.0 - (1.0 /
(pow((1.0 + ratepct), actnper )))));
printf( "Principal: %f\n", pv );
printf( "Interest rate: %f\n", rate );
printf( "Interest rate: %f\n", rate );
printf( "Interest rate: %f\n", rate );
printf( "Interest rate: %f\n", rate );
printf( "Interest rate: %f\n", rate );
}
<F1=Help> <Alt=Menu> <F6=Window> M N 00027.027

```

There are now five copies of the same **printf** statement. Next, you'll modify the **printf** statements so that each corresponds to the preceding example. The cursor should be on the first copy of the **printf** statement.

Before you modify the text, you must select what you are going to modify.

► **To select text:**

1. Point to the I in "Interest Rate" in the second **printf** statement.
2. While holding down the left mouse button, drag the mouse until it is over the colon.

```

File Edit Search Project Run Options Browse Window Help
D:\C700\SOURCE\SAMPLES\PWB\TUTOR\annuity.c
// get input from the user
printf( "Enter present value:\n" );
scanf( "%f", &pv );
printf( "Enter interest rate in percent:\n" );
scanf( "%f", &rate );
printf( "Enter number of periods in years:\n" );
scanf( "%i", &nper );

ratepct = rate / 1200.0;
actnper = nper * 12;
// calculate the payment amount
pmt = pv * (ratepct / (1.0 - (1.0 /
(pow((1.0 + ratepct), actnper ))));
printf( "Principal: %f\n", pv );
printf( "Interest rate: %f\n", rate );
printf( "Interest rate: %f\n", rate );
printf( "Interest rate: %f\n", rate );
printf( "Interest rate: %f\n", rate );
printf( "Interest rate: %f\n", rate );
}
<F1=Help> <alt=Menu> <F6=Window> M N 00027.027

```

The text `Interest Rate` is now selected and highlighted on your screen.

► **To select this text with the keyboard:**

1. Move the cursor to the first character of text you want to select.
2. Press **SHIFT+RIGHT ARROW** until the cursor is on the colon.

Now that the text is selected, type: `Number of Years`. When you type the first character, you'll notice that the selected text is deleted.

Important If you select an area of text and type, PWB replaces the selected text and does not save it on the clipboard. You can recover the text by choosing **Undo** from the **Edit** menu.

Now change the variable `rate` on the same line to `nper`, as follows:

1. Select the word `rate` and press **DEL**.

The word is removed from the file and placed on the clipboard. Pressing **DEL** is a direct way to delete text.

2. Type the word `nper`

Use the techniques you've learned to make the rest of the corrections, and then save the file by choosing **Save** from the **File** menu.

TIP Double-click a word to select it.

Note You can turn on automatic file saving by setting the **Autosave** switch to yes with the Editor Settings command on the Options menu. When **Autosave** is turned on, PWB automatically saves your file before executing certain commands such as running your program or switching to another file. For example, if you run a program that is not yet stabilized, PWB ensures that your file is stored safely in case you have to reboot.

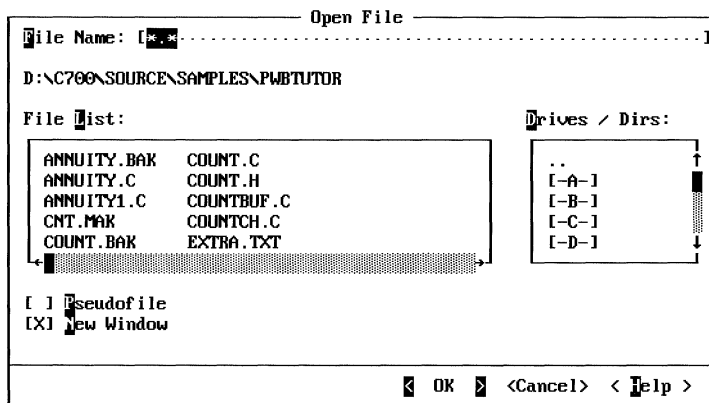
Opening an Existing File

The remainder of this chapter uses another program, ANNUITY1.C, which you can now open in PWB. This program is a slightly different version of the program you just entered. It has several errors you will correct as you follow the tutorial.

► To open ANNUITY1.C:

1. From the File menu, choose Open (press ALT+F, O).

PWB displays the Open File dialog box.



TIP If you know the name of the file, you can type it into the File Name text box.

PWB uses *.* as the default filename. This causes PWB to display all files in the current directory in the File List box.

2. If you are not in the directory where the sample programs are located, press the TAB key twice to move to the Drives/Dirs box. The example files are in \C700\SOURCE\SAMPLES\PWBTUTOR if you accepted the default directory suggested by SETUP.

You'll notice that the cursor is a blinking underline. That means that although you have activated the list box, you haven't yet selected an item.

TIP Double-click the drive or directory to move to that location.

3. Use the arrow keys to move to the drive or directory where the files are located.

As you press the arrow keys, you'll notice that the cursor changes to a bar that highlights the whole selection. This is called the "selection cursor." The text of the selected item also appears in the File Name box.

4. When you have highlighted the drive or directory you want, press **ENTER** to move there.
5. Use the **TAB** key to move to the File List box.
6. Use the arrow keys to move to `ANNUITY1.C`.

7. When you have highlighted `ANNUITY1.C`, press **ENTER** or click **OK** to accept your selection and open the file.

PWB opens `ANNUITY1.C` for editing.

TIP Double-click the filename to open the file.

2.2 Single-Module Builds

Now that you have opened your file, you probably want to compile and run it to see if it works. Compiling the source files and linking them with the run-time libraries is called "building the project." It results in an executable file. A project build can also:

- Create and update the browser database.
- Create a Windows dynamic-link library (DLL).
- Build a library of routines.

Setting Build Options

Before you build a program, you must tell PWB what sort of file to create by using the commands on the Options menu. Use the commands from the Options menu to specify:

- The run-time support for your program. This is important for mixed-language program development, where you have some source files in C and some in another language. With Basic, for example, the run-time support must be Basic's run-time support.

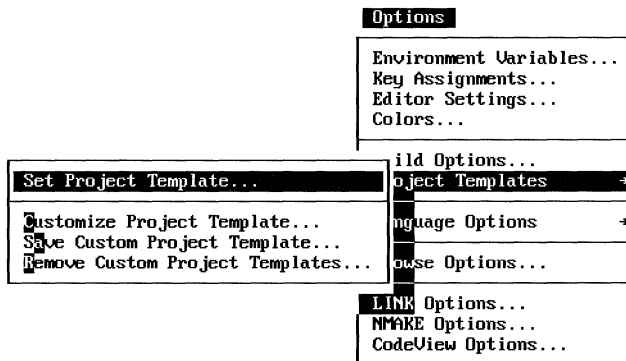
The run-time support you choose determines the run-time libraries that are used and the types of target environments that can be supported.

- Project template. The template describes in detail how PWB is to build a project for a specific type of file (`.EXE`, `.COM`, `.DLL`, `.LIB`) and the operating environment for the target file (DOS, Windows, and so on).
- Either a debug or release build. Debug options normally specify low levels of optimization and the inclusion of CodeView debugging information. Release options specify higher levels of optimization and no CodeView information.

- A build directory. PWB builds your object and executable files in your current directory unless you specify otherwise. (This option is reserved for projects that use explicit project files, which are described in Chapter 3.)

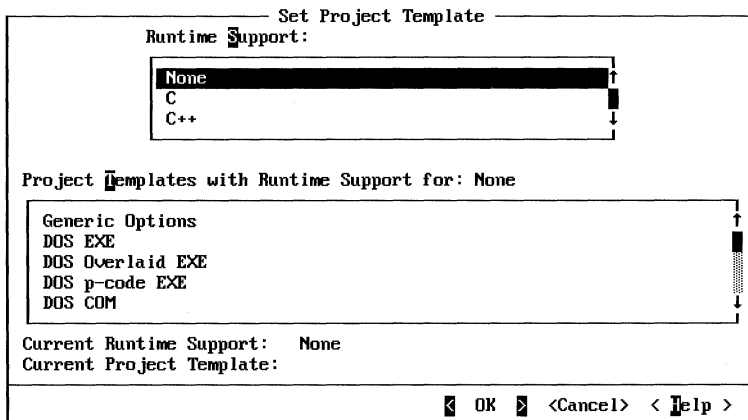
► **To set the project template for ANNUITY1.C:**

1. From the Options menu, choose Set Project Template from the Project Templates cascaded menu.



Note that the actual order of the menu items may differ from the illustration because PWB's extensions can be loaded in any order.

2. PWB displays the Set Project Template dialog box.



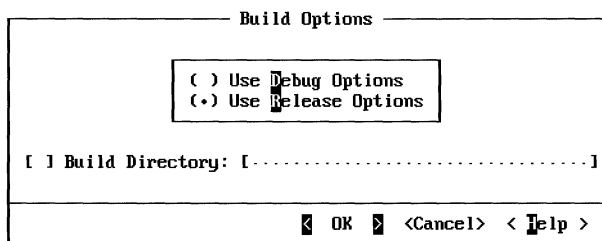
This dialog box typically has the entries `None`, `C`, and `C++` in the Runtime Support list box. If you have installed other languages, their names appear as well.

If the current run-time support is not `C`, you need to choose `C` as the run-time support, and you must also select a project template.

3. Click `C`, or press the `DOWN ARROW` key until `C` is highlighted.
4. Move to the Project Templates list box by clicking in the box, pressing the `TAB` key the appropriate number of times, or by pressing `ALT+T`.
5. Select `DOS EXE`.
6. Choose `OK` to set the new project template.

► **To set the build options for ANNUITY1.C:**

1. From the Options menu, choose Build Options.
PWB displays the Build Options dialog box.



2. Turn on Use Debug Options by clicking the option button or by pressing `ALT+D`.
This option tells PWB that you are building a debugging version of the program. PWB uses debug options when you build or rebuild until you use the Build Options dialog box to choose Use Release Options.
3. Choose `OK`.

PWB saves all the options that you specify. You don't have to respecify them each time you work on your project.

Figure 2.2 shows the three sets of options that PWB maintains for each project. Global options are used for every build. Debug options are used when Use Debug Options is turned on in the Build Options dialog box. Release options are used when Use Release Options is turned on.

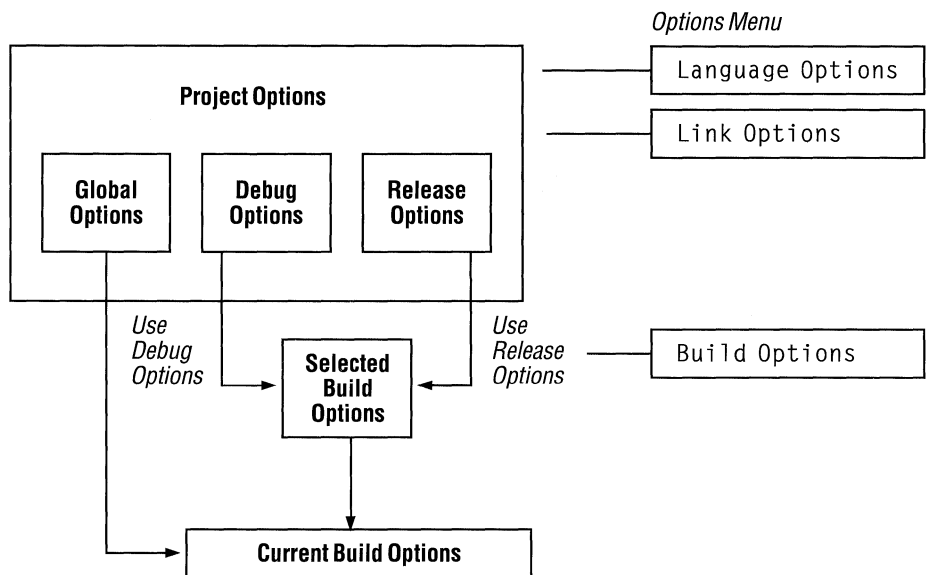


Figure 2.2 PWB Build Options

You can set compiler and linker options for both types of builds by using the Language Options commands and the LINK Options command. These commands do not determine which set of options are used when you build the project. Only the Build Options command determines which set of options (debug or release) are used when you build the project or compile a file in the project.

Global options typically include settings for warning level, memory model, and language variant. These are options that do not change between debug and release versions of a project. The debug and release sets control options that differ between the two types of builds, such as the level of optimization and the inclusion of CodeView debugging information. Debug options normally specify low levels of optimization and the inclusion of debugging information. Release options usually specify high levels of optimization and no debugging information.

Setting Other Options

The Options menu also contains commands that allow you to describe the desired project build more completely. You don't need to use any of these options to build ANNUITY1.C because the default values supplied by the template are correct for the type of program you choose.

The Options menu contains the following commands:

- **C Compiler Options** and **C++ Compiler Options** in the Language Options cascaded menu. These commands let you specify compiler options specific to either debug or release builds and general options common to both types of builds. Use the Compiler Options command to customize the options given by your project template. You can specify memory model, warning level, processor type, and so on.

If you have more languages installed, their Compiler Options commands also appear in the Languages Options cascaded menu.

- **LINK Options**. This command parallels the Compiler Options commands. You can specify options specific to debug or release builds and general options common to both debug and release builds.

Use LINK Options to specify items such as stack size and additional libraries. You can also select different libraries for debug and release builds. This is handy if you have special libraries for debugging and fast libraries for release builds.

- **NMAKE Options**. This command lets you specify NMAKE command-line options for all builds. This option is particularly useful if you have an existing makefile that was not created by PWB or if you have modified your PWB project makefile. For more information about these subjects, see “Using a Non-PWB Makefile” on page 61.
- **CodeView Options**. This command allows you to set options for the CodeView debugger.

Building the Program

Now that you’ve set your options, you can build the program. Note that the sample program contains intentional errors that you will correct.

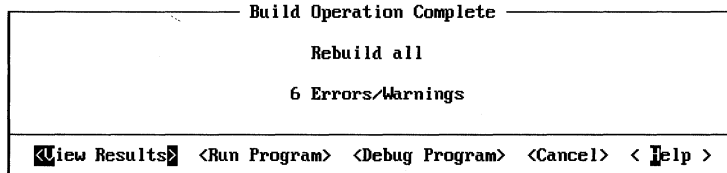
► To start the project build:

1. From the Project menu, choose Build.

PWB tells you that your build options have changed and asks if you want to Rebuild All.

2. Choose Yes to rebuild your entire project.

After the build is completed, PWB displays the following dialog box:



You can choose one of several actions in this dialog box:

- View the complete results of the build by opening the Build Results window.
- Run the program if building in DOS. You can run a DOS program right away if the build succeeds. If the build fails, you should fix the errors before you attempt to run the program.

To run a successfully built Windows program, you must return to the Program Manager and use the Run command on the File menu.

- Debug the program if building in DOS. If the build succeeds but you already know the program is not producing the intended results, you can debug your DOS program using CodeView.

To debug a Windows program, you must return to the Program Manager and start CodeView for Windows from the Microsoft C/C++ Program Group.

- Get Help by choosing the Help button or by pressing F1 (as in every PWB dialog box).
- Cancel the dialog box. This returns you to normal editing.

Choose Cancel to dismiss the dialog box (press ESC). PWB keeps the results of the build so that you can view the build messages later or step through them to view the location of each error. The next section describes how to do this.

Fixing Build Errors

For each build, PWB keeps a complete list of build errors and messages in the Build Results window. The ANNUITY1.C program that you just built contains several errors that you'll identify and fix in this section.

► **To go to the first error:**

- From the Project menu, choose Next Error, or press SHIFT+F3.

PWB positions the cursor on the location of the first error or warning in your program; in this case, the keyword **int** is misspelled. The message from the compiler is displayed on the last line of the window.

```

File Edit Search Project Run Options Browse Window Help
D:\C700\SOURCE\SAMPLES\PWB\TUTOR\ANNUITY1.C
//
// ANNUITY1.C - Generate annuity table.
// Contains intentional errors for use with the PWB Tutorial
//
#include <stdio.h>
#include <math.h>

void main( void )
{
    float Principal, Rate, Pmt, RatePct, PerInterest, PerPrin;
    int Nper;
    int ActNper;
    ont Period;

    //
    // Get input from the user.
    //

    printf( "\nEnter Present Value: " );
    scanf ( "%f", &Principal );
ANNUITY1.C(13) : error C2065: 'ont' : undeclared identifier
<F1=Help> <Error Help> <F6=Window> N 00013.005

```

Whenever a message is displayed on the bottom line of the window, you can get Help on that message by clicking the Help button on the status bar or by pressing F1.

► **To get Help on a message that is not currently displayed:**

1. Press ALT+A. This executes the PWB function **Arg** to begin a text argument.
2. Type the error number with its alphabetic prefix. In this example it would be C2065. For the C and C++ compiler, be sure to use the exact letter case of the message number.
3. Press F1.

When you use one of these techniques to get Help on the message, PWB opens the Help window and displays information about the error.

```

File Edit Search Project Run Options Browse Window Help
Help: C2065 [286]
  <Up> <Contents> <Index> <Back>

Compiler error C2065

'identifier' : undeclared identifier

The specified identifier was not declared.

float Principal, Rate, Pmt, RatePct, PerInterest, PerPrin;
int Nper;
int ActNper;
ont Period;

//
// Get input from the user.
//

printf( "\nEnter Present Value: " );
scanf ( "%f", &Principal );
ANNUIITY1.C(13) : error C2065: 'ont' : undeclared identifier
<F1=Help> <Error Help> <F6=Window> P N 00001.001

```

When you are finished reading the Help, close the Help window by clicking the close box in the upper-left corner of the Help window, by pressing ESC, or by pressing CTRL+F4.

Correct the first error by changing `ont` to `int`.

The compiler reports two additional errors that are side effects of the misspelling of **int**. You could continue choosing Next Error to skip these additional messages, but there is another way to go directly to a selected error in Build Results.

► **To go to a selected error:**

1. From the Window menu, open the PWB Windows cascaded menu and choose Build Results.

PWB opens the Build Results window, which contains the complete results of the build.

```

File Edit Search Project Run Options Browse Window Help
D:\C700\SOURCE\SAMPLES\PWB_TUTOR\ANNUITY1.C
//
// ANNUITY1.C - Generate annuity table.
// Contains intentional errors for use with the PWB Tutorial
//
#include <stdio.h>
#include <math.h>

void main( void )
{
    float Principal, Rate, Pmt, RatePct, PerInterest, PerPrin;
}
Build Results
+++ PWB [D:\C700\SOURCE\SAMPLES\PWB_TUTOR] Rebuild all

NMAKE /a /f c:\temp\PWB08993.mak all

Microsoft (R) Program Maintenance Utility Version 1.20.0053
Copyright (c) Microsoft Corp 1988-91. All rights reserved.

cl /c /W2 /BATCH /qc /Gi.\ANNUITY1.mdt /Zr /Zi /Od /Fo.\ANNUITY1.obj ANNUI
Microsoft (R) C/C++ Optimizing Compiler Version 7.00.252
Copyright (c) Microsoft Corp 1984-1991. All rights reserved.
<F1=Help> <Error Help> <F6=Window> P N 00001.001

```

- Find the next message that is not a side effect of the first error, and move the cursor to that line in the Build Results window.

Move the cursor to the message:

```
error C2001: newline in constant.
```

- From the Project menu, choose Goto Error.

PWB jumps to the location of the second error in the program.

Correct the second error in the program (an unterminated string) by adding the missing double quotation mark (") one space beyond the colon (:) in the prompt string.

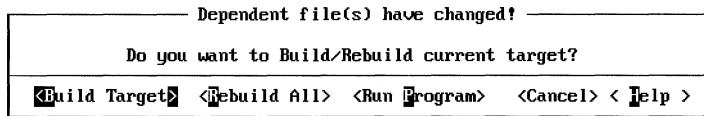
Running the Program

Now that all the errors are corrected, you can run the program.

► To run the program:

- From the Run menu, choose Execute.

PWB detects that you've changed the source and displays a dialog box with the following options:

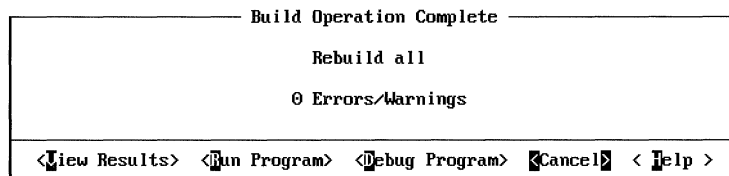


Option	Description
Build Target	Build the program by compiling only the modified files. For more information about building specific targets, see “Using Non-PWB Makefiles” on page 61.
Rebuild All	Build the program by compiling all program files. For this single-module program, Build Target and Rebuild All are equivalent.
Run Program	Run the program without rebuilding it.
Cancel	Cancel the Execute command.

Since you’ve corrected errors, you want to build the target.

2. Choose Build Target to build the program.

When the build completes, PWB displays the following dialog box where you can choose Run Program to run the finished program.



The following sections describe debugging with the Microsoft CodeView debugger. If you’re already familiar with CodeView, you can skip ahead to “Formatting Text” on page 34 or go directly to Chapter 3, “Managing a Multimodule Program.”

2.3 Debugging the Program

PWB integrates several Microsoft tools to produce a complete development environment. Among those tools are NMAKE, a program maintenance utility, and CodeView, a symbolic debugger. You saw how PWB uses NMAKE to build programs; now you can use PWB as a gateway to CodeView.

Earlier, you chose Use Debug Options in the Build Options dialog box. A debug build typically includes the compiler options that generate CodeView information. Therefore, the program is ready to debug with the CodeView debugger.

Using CodeView to Isolate an Error

In addition to the typographical errors that you just corrected, ANNUITY1.C contains a logic error: when the program prints the annuity table, the year numbers start at 0 instead of 1. You can use CodeView to isolate the errors in program logic.

This program calculates the payment on a loan, so you can use the following test case:

```
Present Value:      $14,500
Interest Rate:     14%
Period:            5 Years
```

The expected result is a monthly payment of \$337.39.

► To start CodeView:

- From the Run menu, choose Debug.

TIP You can always return to PWB from CodeView by choosing Exit from CodeView's File menu.

If anything in your program is out-of-date, PWB asks if you want to build or rebuild the current target. If you modified the source file to correct errors or change text, PWB considers it out-of-date relative to the executable file that you built earlier. If this happens, build the program and choose Debug from the Run menu.

CodeView starts, showing you the source line of the program's starting point. In this case, the starting point is the opening brace of the function `main`.

```
File Edit Search Run Data Options Calls Windows Help
[1] local
-----
[3] source1 CS:IP ANNUITY1.C
4: //
5: #include <stdio.h>
6: #include <math.h>
7:
8: void main( void )
9: {
10:     float Principal, Rate, Pmt, RatePct, PerInterest, PerPrin;
11:     int Nper;
12:     int ActNper;
13:     int Period;
14:
-----
=[9] command
>
-----
<F8=Trace> <F10=Step> <F5=Go> <F3=Src1 Fmt> DEC
```

The first step in debugging a program is to verify input values. You will know what values have been supplied after the last `scanf` statement has executed, so run the program up to that point as follows:

1. If the Source window (the window displaying your program) is not the active window, press F6 until it is. You can tell that a window is active when the title bar is highlighted and it has scroll bars.
2. Move the cursor to line 32, `RatePct = Rate / 1200.0`
3. Press F7 (continue execution to cursor).

TIP Click the right mouse button on line 32 or type `G.32` to continue execution to line 32.

The program runs, asking for input. Supply the values you are using as a test case:

```
Present Value:      14500
Interest:          14
Number of Periods: 5
```

CodeView stops your program at line 32.

The screenshot shows the CodeView interface. At the top is a menu bar with 'File', 'Edit', 'Search', 'Run', 'Data', 'Options', 'Calls', 'Windows', and 'Help'. Below the menu bar is a window titled 'local' containing the following variable values:

```
[BP-0006] float PerPrin = 1.56184e-008
[BP-000A] float Rate = 14.0000
[BP-000E] float PerInterest = 4.05925e-038
[BP-0010] short Period = -32557
[BP-0012] short Nper = 5
[BP-0016] float Pmt = 4.62134e-041
[BP-001A] float Principal = 14500.0
[BP-001E] float RatePct = 2.41561e+007
```

Below the Local window is a window titled 'command' with a cursor on line 32:

```
[9] command
>
```

At the bottom of the window is a status bar with the following text: `<F8-Trace> <F10-Step> <F5-Go> <F3-Src1 Fmt> <ENTER-Expand> DEC`

TIP You can resize the window with the mouse by dragging the lower-right corner to the desired location.

4. Resize the Local window until you can see all the variables:
 - a. Press F6 until the Local window is active. (The active window is the window with the scroll bar.)
 - b. From the Window menu, choose Size.
 - c. Use the DOWN ARROW key to enlarge the Local window.
When the window is the desired size, press ENTER to accept that size.
 - d. Press F6 to move back to the Source window.

Now you can verify that the initial data used by your program is correct by examining the values of `Pv`, `Rate`, and `Nper`. They should have the values 14500, 14, and 5, respectively.

TIP The CodeView interface and menu commands are similar to PWB, so techniques you use in PWB are often useful in CodeView.

Note For case-insensitive languages such as Basic or FORTRAN, CodeView displays all variables, subroutine names, and function names in uppercase.

The next step is to execute the program until the initial calculations are done. The calculations are complete prior to the `for` loop. If you let the program execute that far, the program produces some screen output that's useful for debugging.

► **To execute to the for loop:**

1. From the Search menu, choose Find.
2. Type `for` in the Find Text box.
3. Press ENTER to move the cursor to the `for` statement.

Although your cursor is now on line 62, the program has not executed the statements between where it stopped (on line 32) and the current cursor position.

4. Press F7 to execute all code up to but not including this location.

Your program has now displayed the summary information on the screen. To switch to the output screen, press F4. To switch back to the CodeView screen, press F4 again (or any key).

On the output screen, you should observe the following results:

Monthly Payment:	337.39
Total Payments:	20243.38
Total Interest:	5743.38

These results are correct. You know that the program works properly up to the beginning of the `for` loop. Therefore, you can ignore all code up to this point and focus on discovering why the year number is incorrect.

► **To step through one cycle of the loop and examine your data:**

1. Press F10 three times to step three lines.

This calculates values for `PerInterest` (interest for the current period) and `PerPrin` (contribution to principal for the current period).

2. Examine the values of `PerInterest` and `Prin` in the Local window to see if they are correct. The formula for simple interest is:

$$\text{Interest} = \text{Outstanding Principal} * \text{Periodic Interest Rate}$$

Similarly, the formula for the contribution principal is:

$$\text{Contribution} = \text{Payment} - \text{Interest}$$

TIP You can type P 3 in the Command window to step three statements.

In the test case, the correct values for `PerInterest` and `PerPrin` are 169.167 and 168.223, respectively. These values should appear on your screen.

3. Press F10 again to step one more program statement—the **printf** statement.
4. Press F4 to examine the screen output. The year number is still incorrect.

You have reduced the problem to the arithmetic in the **printf** call itself. The argument list for **printf** contains the expression `Period / 12`. Integer truncation causes all values of `Period` that are less than 12 to yield the result 0.

Now that you have identified the apparent bug, you can test the solution. The proposed solution to this problem is to replace the expression `Period / 12` with `Period / 12 + 1`. You can test this solution by using the CodeView expression evaluator.

► **To test the new expression:**

1. Activate the Command window.
2. Type the Display Expression (?) command with the test expression:

```
? Period / 12 + 1
```

CodeView evaluates the expression and prints 1, the correct result.

Testing Conditions in the Watch Window

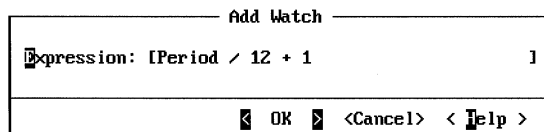
You know the solution works for one iteration of the loop but not if it works for other conditions. You can test a larger range of possible conditions by using breakpoints and the Watch window—a window in which you can view the value of selected data or expressions during a debugging session.

► **To test conditions using the Watch window:**

1. From the Data menu, choose Add Watch.
2. Type the proposed expression:


```
Period / 12 + 1
```
3. Choose OK to put this expression in the Watch window.

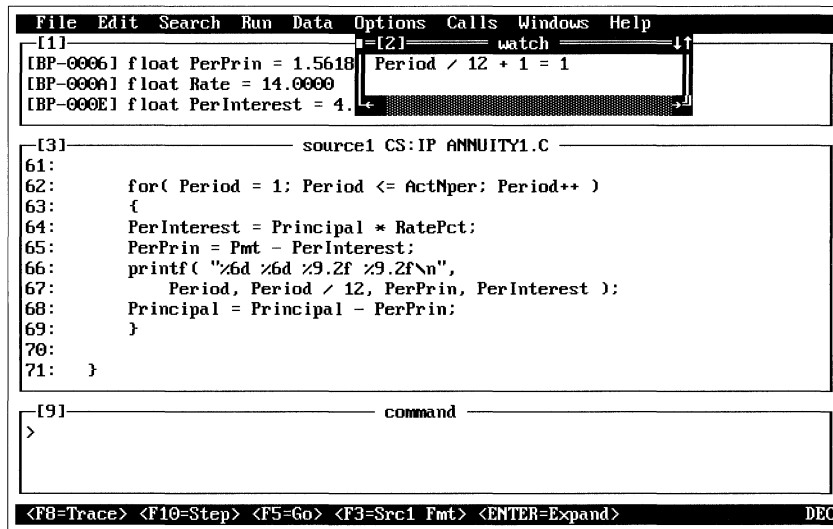
TIP You can use **CTRL+W** to add a watch expression quickly.



TIP To set a breakpoint on a line, double-click the line.

Now you can see the results of the expression as your program executes, but you have to stop the program at places where the results of this expression are informative. You do this by setting breakpoints. A breakpoint is a location to stop execution or a condition when you want your program to stop.

1. Move to the Source window by clicking it or by using the F6 key.
2. Move the cursor to the **printf** statement and press F9 to set a breakpoint there.



This breakpoint stops execution and returns control to CodeView each time the **printf** statement is about to be executed. You can then examine the values of the variables in the Local window and the results of your expression in the Watch window.

3. Press F5 to run the program.

TIP The status bar reminds you about commonly used actions and keys. You can click the buttons with the mouse to carry out the actions.

Now each time you press F5, the program executes all statements up to but not including the **printf** statement. Repeat this until `Period` equals 12. At this point, you are in the last month of the first year. Notice that the expression you specified does not handle this boundary condition correctly. It changes from 1 to 2 one period too early.

► **To adjust for the boundary condition:**

1. Add the following expression to the Watch window:

```
(Period - 1) / 12 + 1
```

2. Restart the program by choosing Restart from the Run menu.

3. Press F5 to start execution.

The program asks for input again. These values are:

```
Present Value:      14500
Interest:          14
Number of Periods: 5
```

Your program stops at the breakpoint you set on the **printf** statement.

On the first iteration, you'll notice that the watch expression from the last run, `Period / 12 + 1`, is still in the Watch window.

► **To remove the incorrect expression:**

1. From the Data menu, choose Delete Watch.
2. Select the expression you want to delete.
3. Press ENTER to delete the Watch expression from the window.

Run your test; press F5 to step through the loop one iteration at a time. This time, you should get the correct results.

► **To switch from CodeView back to PWB:**

- Choose Exit from the CodeView File menu.

Now that you've built and debugged your program, you may want to reformat your code to make it easier to read. The following section describes how to do this using PWB's editing functions.

2.4 Formatting Text

Well-formatted code is more readable and easier to maintain. ANNUITY1.C is not very well formatted, but PWB can help you indent blocks of code to make them more readable. For example, the **printf** statement at line 41 is continued across multiple lines. Indenting continued lines of a statement lends clarity to the code.

TIP CodeView numbers watch expressions starting at zero. To remove this expression, you can type `WC0` in the Command window.

Indenting Lines of Code

► To indent lines:

TIP The line and column of the cursor appear in the status bar.

1. Go to the statement where you want to indent text (on line 41). Press ALT+A, type 41, then press CTRL+M to jump to line 41.

This sequence of keystrokes is pronounced “Arg 41 Mark.” The PWB function **Arg** begins an argument (41) that is passed to the **Mark** function. When you pass a number to **Mark**, PWB moves the cursor to that line.

You can also do this from the menu by typing the line number in the Goto Mark dialog box from the Search menu.

2. Move the cursor to the double quotation mark (") in column 5.
3. Hold down the SHIFT key and press the RIGHT ARROW key eight times so that the cursor is in column 13, under the opening double quotation mark in the **printf** statement.

```

File Edit Search Project Run Options Browse Window Help
#l 11 D:\C700\SOURCE\SAMPLES\PWB\TUTOR\ANNUITY1.C
// Print a summary of the annuity
//
printf( "\n\n"
"Principal:      %13.2f\n"
"Interest Rate: %13.2f\n"
"Number of Years: %13i\n"
"Monthly Payment: %13.2f\n"
"Total Payments: %13.2f\n"
"Total Interest: %13.2f\n\n",
Principal, Rate, Nper, Pmt,
Pmt * (float)Nper * 12.0,
Pmt * (float)Nper * 12.0 - Principal );

//
// Print headings of the amortization table.
//
printf( "Period Year  Principal Interest\n"
"-----\n\n" );

//
// Loop on the number of periods, printing the period, year,

```

<General Help> <F1=Help> <Alt=Menu> N 00041.013

4. Press SHIFT+DOWN eight times to select the rest of the statement.

```

File Edit Search Project Run Options Browse Window Help
D:\NC700\SOURCE\SAMPLES\NWBUTUTOR\ANNUITY1.C
// Print a summary of the annuity
//
printf( "\n\n"
"Principal:      %13.2f\n"
"Interest Rate:  %13.2f\n"
"Number of Years: %13i\n"
"Monthly Payment: %13.2f\n"
"Total Payments: %13.2f\n"
"Total Interest: %13.2f\n\n",
Principal, Rate, Nper, Pmt,
Pmt * (float)Nper * 12.0,
Pmt * (float)Nper * 12.0 - Principal );

//
// Print headings of the amortization table.
//
printf( "Period Year  Principal Interest\n"
"-----\n" );

//
// Loop on the number of periods, printing the period, year,
<General Help> <F1=Help> <Alt=Menu> N 00049.013

```

By default, the editor starts in stream selection mode. This mode allows selection to begin at any point and selects all characters in a stream between the beginning and end of a selection, as shown above. You will need to change the selection mode to perform the block indent.

The Edit menu lets you choose from three selection modes:

- Stream mode. The default, as explained previously.
- Line mode allows you to select complete lines of text.

```

File Edit Search Project Run Options Browse Window Help
D:\NC700\SOURCE\SAMPLES\NWBUTUTOR\ANNUITY1.C
// Print a summary of the annuity
//
printf( "\n\n"
"Principal:      %13.2f\n"
"Interest Rate:  %13.2f\n"
"Number of Years: %13i\n"
"Monthly Payment: %13.2f\n"
"Total Payments: %13.2f\n"
"Total Interest: %13.2f\n\n",
Principal, Rate, Nper, Pmt,
Pmt * (float)Nper * 12.0,
Pmt * (float)Nper * 12.0 - Principal );

//
// Print headings of the amortization table.
//
printf( "Period Year  Principal Interest\n"
"-----\n" );

//
// Loop on the number of periods, printing the period, year,
<General Help> <F1=Help> <Alt=Menu> N 00049.013

```

- Box mode allows you to select a rectangular section of text.

```

File Edit Search Project Run Options Browse Window Help
D:\C700\SOURCE\SAMPLES\PWB\TUTOR\ANNUITY1.C
// Print a summary of the annuity
//
printf( "\n\n"
Principal:  %13.2f\n"
Interest Rate:  %13.2f\n"
Number of Years: %13i\n"
Monthly Payment: %13.2f\n"
Total Payments: %13.2f\n"
Total Interest: %13.2f\n\n",
Principal, Rate, Nper, Pmt,
Pmt * (float)Nper * 12.0,
Pmt * (float)Nper * 12.0 - Principal );

//
// Print headings of the amortization table.
//
printf( "Period Year  Principal Interest\n"
"-----\n" );

//
// Loop on the number of periods, printing the period, year,

```

The screenshot shows a DOS-style command prompt window with a menu bar (File, Edit, Search, Project, Run, Options, Browse, Window, Help) and a title bar (D:\C700\SOURCE\SAMPLES\PWB\TUTOR\ANNUITY1.C). The main window contains C code. A rectangular selection box is drawn around the following lines of code:

```

printf( "\n\n"
Principal:  %13.2f\n"
Interest Rate:  %13.2f\n"
Number of Years: %13i\n"
Monthly Payment: %13.2f\n"
Total Payments: %13.2f\n"
Total Interest: %13.2f\n\n",
Principal, Rate, Nper, Pmt,
Pmt * (float)Nper * 12.0,
Pmt * (float)Nper * 12.0 - Principal );

```

The status bar at the bottom of the window displays "<General Help> <F1=Help> <Alt=Menu>" on the left and "N 00049.013" on the right.

When the starting column of the selection is the same as the ending column, PWB selects the range of lines, just as it does for line selection mode.

TIP To change selection modes with the mouse, click the right button while holding down the left button.

5. Choose Box Mode from the Edit menu. Your screen should look like the preceding picture.
6. Press CTRL+N to indent the lines.

Pressing CTRL+N executes the **Linsert** function. When you have a box selected, **Linsert** inserts spaces into the selected area. With no selection, **Linsert** inserts a line above the cursor.

Now the **printf** format string and arguments are neatly aligned.

```

File Edit Search Project Run Options Browse Window Help
D:\NC700\SOURCE\SAMPLES\PWB\TUTOR\ANNUITY1.C
// Print a summary of the annuity
//
printf( "\n\n"
        "Principal:      %13.2f\n"
        "Interest Rate:   %13.2f\n"
        "Number of Years: %13i\n"
        "Monthly Payment: %13.2f\n"
        "Total Payments:  %13.2f\n"
        "Total Interest:  %13.2f\n\n",
        Principal, Rate, Nper, Pmt,
        Pmt * (float)Nper * 12.0,
        Pmt * (float)Nper * 12.0 - Principal );

//
// Print headings of the amortization table.
//
printf( "Period Year  Principal Interest\n"
        "-----\n" );

//
// Loop on the number of periods, printing the period, year,

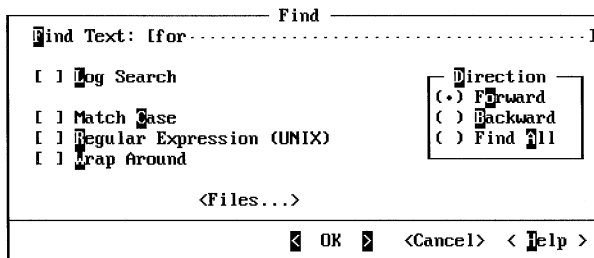
```

Searching for Text

You can improve readability by indenting statements within loops. You'll now use PWB's search menu to find a `for` loop and indent it.

► To find the `for` loop:

1. From the Search menu, choose Find.
PWB displays the Find dialog box.
2. Type `for` in the Find Text text box.



3. Click OK or press ENTER to locate the `for` statement.
4. You're still in box selection mode, so select the area between the `for` and the terminating brace. Make the selection four characters wide.
5. Indent the block by pressing CTRL+N to execute the **Linsert** function.

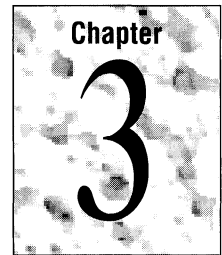
For more information on searching, see “Searching with PWB,” on page 85.

You’ve now learned the basics of editing and reformatting text. PWB has many more commands for manipulating text. See Chapter 7, “Programmer’s WorkBench Reference,” for details on all PWB functions.

2.5 Where to Go from Here

Now that you’ve created, built, and debugged a simple program, you’ve begun to discover the power of PWB. In Chapter 3, “Managing Multimodule Programs,” you learn how to create and manage projects with more than one source file.

Managing Multimodule Programs



This chapter expands on the work you did in Chapter 2 and explains how to build and maintain multimodule programs using PWB's integrated project-management facilities. PWB offers a new, more efficient way to manage complex projects. You organize and build your project entirely within PWB, using convenient menus and dialog boxes instead of makefiles or batch files.

PWB stores the information needed to build and manage your program in two files, the project makefile and the project status file. These are called the "project." When you open the project, PWB automatically configures itself to build your program. To move from one project to another, you close one project and open another.

3.1 Multimodule Program Example

In this chapter, you'll learn to set up a multimodule project in PWB by building COUNT.EXE, a three-module program. The COUNT program analyzes text files and produces a statistical profile of the text.

The following modules make up COUNT.EXE:

Module	Function
COUNT.C	Program driver; contains main and calls all other routines.
COUNTBUF.C	Analyzes text in the input buffer.
COUNTCH.C	Analyzes a character.

The program also contains a common header file COUNT.H in addition to these three source modules. Figure 3.1 shows the components of COUNT and how they combine to build the executable file. Later in the tutorial, you will add the SETARGV.OBJ object file.

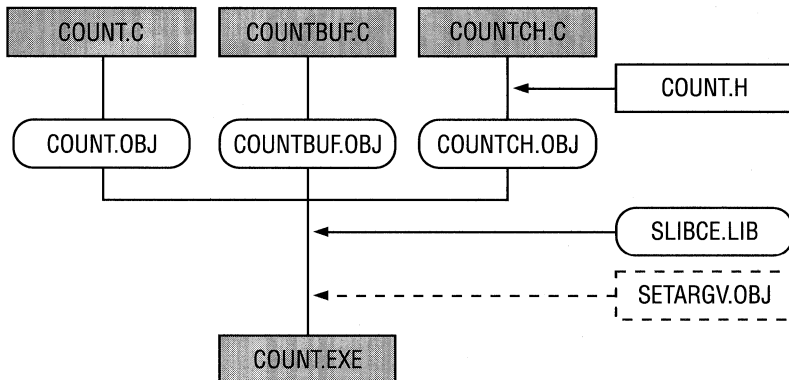


Figure 3.1 The COUNT Project

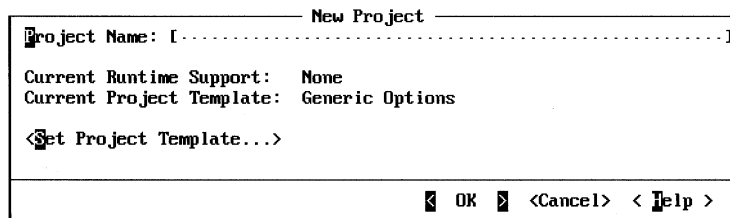
To build COUNT.EXE, you need to compile the three source files and link the correct libraries. You also need to specify various options, such as the target operating environment. All this information is contained in the COUNT project.

Creating the Project

Start by creating a new project for COUNT. (If you have not started PWB, do so now.)

► To create a new project:

1. From the Project menu, choose New Project.
PWB displays the New Project dialog box.

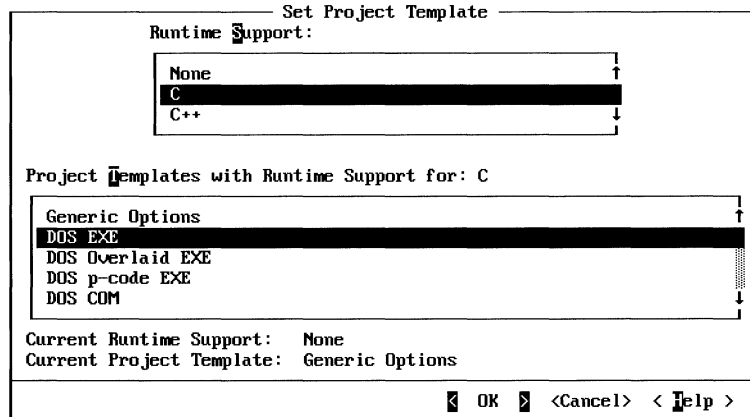


TIP The executable file you create takes on the base name of the project.

2. Type COUNT in the Project Name text box.
3. Choose Set Project Template.
PWB displays the Set Project Template dialog box.
4. Select the following options:
 - Runtime Support: C.

- Project Template: DOS EXE.

At this point, the Set Project Template dialog box should appear as follows:



This initial specification tells PWB what you intend to build and is saved as part of the project.

5. Choose OK to return to the New Project dialog box, and then choose OK.

PWB displays the Edit Project dialog box for adding files to your new project.

The next section describes the types of files that can be added to the project. The tutorial then continues by listing the example files to add to the list.

Contents of a Project

A project file list can contain the following files:

- Source code files (.C, .CPP).
- Object files (.OBJ).
- Library files (.LIB).
- Module-definition files (.DEF).
- Resource-compiler source files (.RC).

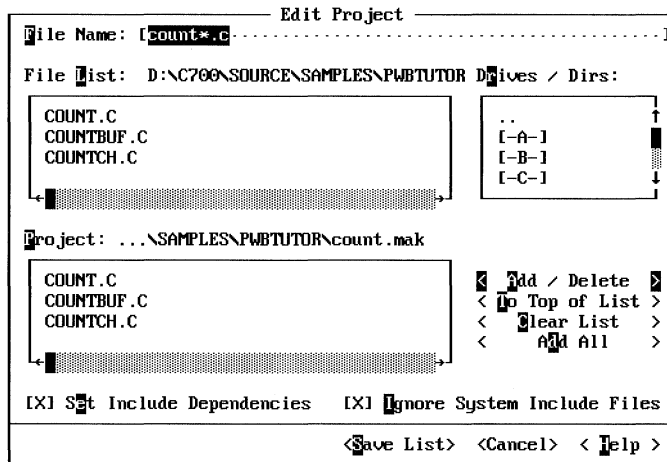
These file types are all that are needed to create most DOS and Windows applications. Include files, such as `STDIO.H`, are not put in a project because they are not primary components of a program build. PWB automatically adds the necessary include files to your project. For more information on include files, see “Dependencies in a Project” on page 45.

When you select the type of run-time support, PWB automatically specifies standard library files such as SLIBCE.LIB. Therefore, you do not need to add standard library files to the project list.

► **To add the COUNT files to your project:**

TIP To add the files in one step, type `CO*.C` in the File Name text box and press ENTER. Then choose Add All.

1. Choose the files you want to add to the project from the File List box. In this case, you'll add `COUNT.C`, `COUNTBUF.C`, and `COUNTCH.C`. These files are located in the `\C700\SOURCE\SAMPLES\PWBTUTOR` directory. If you installed Microsoft C/C++ in a directory other than `C700`, adjust the path accordingly.



You can scroll the File List box by clicking the scroll bars or by pressing the arrow keys. For more information about using list boxes and other elements of the PWB interface, see Chapter 4, “User Interface Details.”

TIP Double-click a file to add or remove it from the list.

2. For each file, select it and choose Add / Delete to add the file to the Project list box.
3. Choose Save List when you have added all three files.

PWB uses the rules in the project template along with the list of files that you just specified to scan the sources for include dependencies and to create the project makefile. This process is described in the next section.

Now your project completely describes what you want to build (the project template), the component source files, and the commands used to build the project.

Dependencies in a Project

When you save the project, PWB generates a makefile from the project template, files, and options you specified. This file also contains a list of instructions that are interpreted by NMAKE. In addition, PWB generates the project status file, which saves the project template, the editor state, and the build environment for the project. For more information on the project status file, see “Project Status Files” on page 138.

When you build the project, NMAKE examines the build rules in the project makefile. These are rules that specify targets (such as an object or an executable file) and the commands required to build them. For example, a rule for making a .OBJ file from a .C file can be expressed as follows:

```
.c.obj:  
    CL /c $<
```

To reduce the amount of time builds take, NMAKE compiles or links only the targets that are out-of-date with respect to their corresponding source file. This process is simple if there is a one-to-one correspondence between sources and targets. However, most programs use the **#include** directive to include definitions or other program text. The object files must be made dependent not only on the source file but also on the files that are used by the source file.

In the preceding section, you learned that you don't add include (.H) files to your project. When you save the project, PWB scans your source files looking for **#include** directives and builds dependencies on these files. Therefore, NMAKE recompiles a source file if you change a file that it includes.

Scanning for include files can take some time, especially when using the Windows include files. Because these system include files rarely change, you can turn on the Ignore System Include Files check box in the Edit Project dialog box. This prevents PWB from scanning these include files for dependencies.

Building a Multimodule Program

Now that the project files are complete, you can build the program in the same way you built the single-module program.

► To build a multimodule program:

1. You are starting on a new project, so you will want to use debug options for the initial builds. Turn on the Use Debug Options option button in the Build Options dialog box, as you did in “Setting Build Options” on page 19.

2. From the Project menu, choose Build.

PWB displays a dialog box to inform you that build information has changed because you altered the build options.

3. Choose Yes to rebuild your entire project.

As the program is built, PWB shows status messages about the progress of the build. When the build completes, a dialog box displays a summary of any errors encountered during the build process.

Note The Next Error command on the Project menu works the same for a multi-module build as for a single-module build. Because errors in a multimodule build can occur in different files, PWB automatically switches to the file that contains the error.

In some cases, you will want to force a complete rebuild of your project by choosing Rebuild All from the Project menu. The difference between Build and Rebuild All is that Build compiles and links only out-of-date targets and Rebuild All compiles all targets, regardless of whether they are current.

Running the Program

Now that your program is built, you can test it from inside PWB.

► To run COUNT:

1. From the Run menu, choose Program Arguments.
2. Type the name of a text file to pass to the COUNT program. The COUNT.C source file is a good file to use.
3. Choose OK to set the program arguments. PWB saves the arguments so that you can run or debug the program many times with the same command line.
4. From the Run menu, choose Execute.

The results look like this:

File statistics for extra.txt

Bytes:	1029
Characters:	770
Letters:	649
Vowels:	233
Consonants:	416
Words:	141
Lines:	45
Sentences:	13
Words per sentence:	10.8
Letters per word:	4.6
Estimated symbols per word:	1.8

Press ENTER to return to PWB.

You have successfully created a multimodule project, built the program, and run it, all from within the Programmer's WorkBench. You can now leave PWB.

► **To leave PWB:**

- From the File menu, choose Exit or press ALT+F4.

PWB saves your project and returns to the operating-system prompt. If you ran PWB from Windows, PWB returns to Windows.

Creating a PWB project is an important first step. However, most of the time you will be maintaining projects. The next section provides an overview of project maintenance. The tutorial then continues with the COUNT project.

Project Maintenance

Once you have created a project, you may have to change it to reflect the changes in your project organization. You can:

- Add new file-inclusion directives to your source files.
- Add new source, object, or library files.
- Delete obsolete files.
- Move modules within the list.
- Change compiler and linker options.
- Change options for individual modules.

When you add a new include directive to a source file, you add a new dependency between files. For the most accurate builds, you need to regenerate include dependencies for the project.

► **To regenerate include dependencies:**

1. From the Project menu, choose Edit Project.
2. Turn on the Set Include Dependencies check box.
3. Choose Save List.

PWB regenerates the include dependencies for the entire project and rewrites the project makefile.

► **To add new files to an existing project:**

1. From the Project menu, choose Edit Project.
2. For each file that you want to add to the project:
 - a. Select the file from the File List box, or type the name of the file in the File Name text box.
 - b. Choose the Add / Delete button to add the file.
3. Choose Save List to rewrite the project makefile, set up the dependencies, and add the commands for the new files.

To see how to add the SETARGV.OBJ file to the COUNT project, see “Adding a File to the Project” on page 50.

► **To delete files from a project:**

1. From the Project menu, choose Edit Project.
2. For each file that you want to remove from the project:
 - a. Select the file from the File List box, or type the name of the file in the File Name text box.
 - b. Choose the Add / Delete button to remove the file from the list.
3. Choose Save List.

With most programming languages, you won't need to move modules within a project. However, some languages or custom projects require files to be in a specific order. If you're programming in Basic, for example, you must place the main module of your program at the top of the list. Unlike other languages, Basic does not define an explicit name where execution begins. Entry to a Basic program is defined by the first file in the list.

► **To move a file to the top of the project file list:**

1. From the Project menu, choose Edit Project.
2. Select the file you want to move to the top of the list.
3. Choose the To Top of List button.

Using Existing Projects

You'll now use the COUNT project that you just created for further work.

During a PWB session, the project you open remains open unless you explicitly change it. If you have not already started PWB, you should do so now. In Windows, click the PWB icon in the Microsoft C/C++ program group.

If you are not compiling from within Windows, you can start PWB and open the COUNT project from the operating-system command line by typing the command:

```
PWB /PP COUNT
```

TIP To automatically reopen the last project whenever you start PWB, set the Lastproject switch to yes.

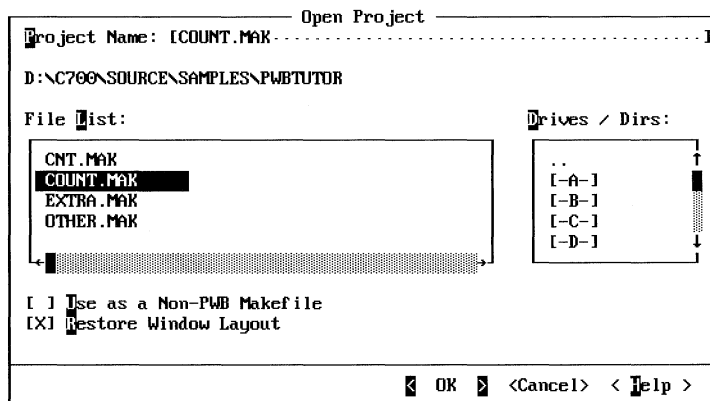
If the COUNT project is the last project you had open in PWB, type the following command:

```
PWB /PL
```

If you have already started PWB, open the project now.

► **To open the project from within PWB:**

1. From the Project menu, choose Open Project.
2. Choose COUNT.MAK from the File List box or type COUNT in the Project Name text box.



3. Choose OK

When you open the project, PWB restores the project's environment, including:

- The window layout with the window style, size, and position for each window.
- The file history—a list of open files for each window and the last cursor position in each file.
- The last find string.
- The last replace string.
- The options that you used for the last find or find-and-replace operation, such as regular expressions. See “Using Regular Expressions” on page 90 for more information about regular expressions.
- The project template (for example, `DOS EXE`) and any customizations you have made to the template such as changing the build type or a compiler or linker option.
- The command-line arguments for your program.
- All environment variables, including `PATH`, `INCLUDE`, `LIB`, and `HELPPFILES`.

Note that you can customize the way PWB handles environment variables by changing the `Envcursave` and `Envprojsave` switches. For more information, see “Environment Variables” on page 137.

Note When you turn off the Restore Window Layout option, PWB does not restore the window layout, the find strings and options, or the file history. PWB opens the project but keeps the same editor state as it had before you opened the project.

Adding a File to the Project

As you develop a project, you will occasionally add new modules. For example, you can add the object file `SETARGV.OBJ` to the `COUNT` project so that the `COUNT` program accepts wildcards on the command line.

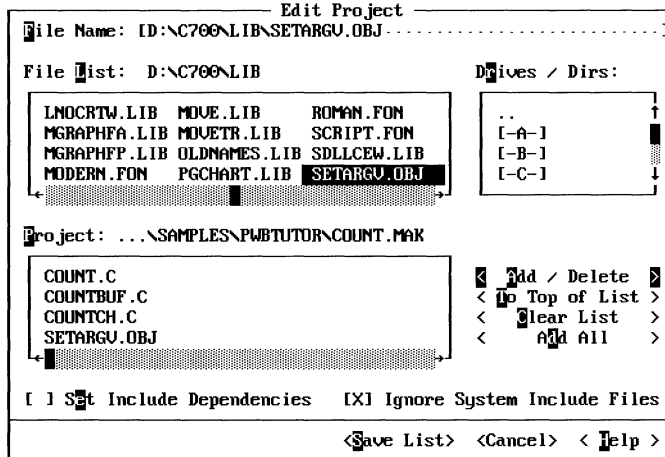
► To add `SETARGV.OBJ` to your project:

1. From the Project menu, choose Edit Project.

The file and directory navigation lists in this dialog box work in exactly the same way as those in the Open File dialog box. Choose the parent directory symbol (`.`) in the Drives / Dirs list box to move up the directory tree. To move down the tree, choose the destination directory.

TIP To specify a directory listed in an environment variable such as LIB, type \$LIB: in the File Name text box and press ENTER.

2. Change to the directory that contains your C libraries.



Notice that the directory displayed after the label `File List` reflects the directory change.

3. Make sure the File Name text box contains `*.*` or `*.OBJ`.
4. Select `SETARGV.OBJ` in the File List box.
5. Choose the Add / Delete button to add the file to the project.
6. Since `SETARGV.OBJ` is not a source file and cannot have include dependencies, you can turn off the Set Include Dependencies check box. If this check box is left on, PWB regenerates the dependencies for all the files in the project.
7. Choose Save List.

`SETARGV.OBJ` is now part of the project. However, if you build the program now, the linker displays the message:

```
error L2044: __setargv : symbol multiply defined, use /NOE
```

The linker produces this error because `SETARGV.OBJ` redefines entry points in the standard run-time library. You must change linker options to disable extended dictionary searching (that is, use the `/NOE` option).

► To change the linker options:

1. From the Options menu, choose LINK Options.
2. Choose Additional Global Options.
3. Turn on the No Extended Dictionary Search in the Library check box.

4. Choose OK to close the Additional Global LINK Options dialog box.
5. Choose OK to close the LINK Options dialog box and use the new options.

You are now ready to build COUNT with the new command-line processing.

► **To build the modified project:**

1. From the Project menu, choose Rebuild All.

PWB displays the message:

```
Current directory is not the project directory.  
Change to project directory?
```

You received this message because you changed the current directory to the directory with the C libraries when you added SETARGV.OBJ.

2. Choose OK to switch to the project directory and build the project.

You can run the COUNT program as before by choosing Execute from the Run menu. To see how the program works with the new command-line processing, you can specify *.C as the argument.

Changing Compiler Options

Up to this point, you have used PWB's default build options for all the examples. These options are sufficient for many cases, but occasionally you will want to adjust them.

Suppose you decide to optimize the COUNT program for size to get the smallest code possible regardless of the execution speed. Ordinarily, you don't consider optimizations until your code has stabilized and you are ready to try a release build. (A release build is normally a build with optimizations turned on and debugging information turned off.)

► **To specify a release build:**

1. From the Options menu, choose Build Options.
PWB displays the Build Options dialog box.
2. Choose Use Release Options.
3. Choose OK to accept your choice.

When you specify a release build, PWB does not change your debug options. For more information on global options, debug options, and release options, see "Setting Build Options" on page 19.

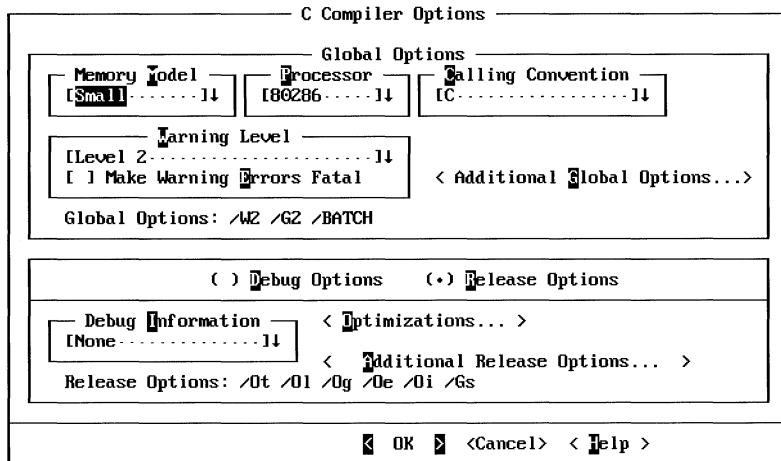
Now that you have chosen a release build, you can set specific options that PWB uses to create a release version of the program.

► **To change compiler options to optimize for space:**

1. From the Language Options cascaded menu on the Options menu, choose C Compiler Options.

The C Compiler Options dialog box displays the following options that are common to both the release and debug builds (Global Options):

- Memory model
- Processor (type of CPU)
- Calling convention
- Warning level



At the bottom of the dialog box is a panel that shows options that are specific to the current type of build. In this case, release options are being used. The default settings for a build were determined when you chose the project template.

Note You can choose the Debug Options button to view and set the options for debug builds. However, this does not change the type of build that is performed when you build the project. To set the type of build, choose Build Options from the Options menu.

TIP Press ALT+O to choose Optimizations.

1. Choose Optimizations.

PWB displays a dialog box in which you can specify release optimizations.

Release Optimization Options

<p style="text-align: center;">General</p> <p><input type="radio"/> Optimize for Time</p> <p><input checked="" type="radio"/> Optimize for Space</p> <p><input type="radio"/> Disable Optimization</p>	<p style="text-align: center;">Specific</p> <p><input checked="" type="checkbox"/> Loop Code Optimization</p> <p><input checked="" type="checkbox"/> Merge Global Expressions</p> <p><input type="checkbox"/> Merge Local Expressions</p> <p><input type="checkbox"/> Improve Float Consistency</p> <p><input checked="" type="checkbox"/> Global Register Allocation</p> <p><input type="checkbox"/> Generate Intrinsic Functions</p>
<p style="text-align: center;">Inlining</p> <p><input type="radio"/> Suppress Inlining</p> <p><input checked="" type="radio"/> Allow Explicit Inlining</p> <p><input type="radio"/> Allow Automatic Inlining</p>	
<p style="text-align: center;">Aliasing</p> <p><input type="radio"/> Assume No Aliasing</p> <p><input type="radio"/> Aliasing Only Across Calls</p> <p><input checked="" type="radio"/> Allow Aliasing</p>	
<input type="checkbox"/> OK <input type="checkbox"/> <Cancel> < <input type="checkbox"/> Help >	

2. Turn on the Optimize for Space option. PWB automatically turns off the Optimize for Time option.
3. Choose OK to return to the C Compiler Options dialog box.
4. Choose OK to set the new options that you have selected.

The procedure you have just completed causes PWB to build an executable file that is optimized for space the next time you choose Build or Rebuild from the Project menu.

Changing Options for Individual Modules

Most of the modules in your system use the same build options. However, you will occasionally need to modify the options for a single module. For example, if the code size is critical on most modules but one module needs to be optimized for speed, you can set your compiler options to optimize for space, which handles the predominant case. You can then modify the options for the module that you want to optimize for speed.

The example that follows shows how to customize your project to change the compiler options to optimize only COUNTCH.C for speed.

First, set the compiler options for the most general case. For COUNT in this example, the most general case is to optimize for space. (If you have been following the tutorial, you did this in the previous section.)

Once you have set the options for the general case, you have to customize the project to compile only COUNTCH.C with optimizations for time. To do this, you manually edit the instructions in the project makefile for compiling COUNTCH.C.

► **To open COUNT.MAK for editing:**

1. If the COUNT project is open, choose Close Project from the Project menu.
This step is important because you cannot edit a PWB makefile for a project that is currently open.
2. Choose the Open command from the File menu and open the COUNT.MAK file in the editor.

Find the rule for compiling COUNTCH.C:

```
COUNTCH.obj : COUNTCH.C
!IF $(DEBUG)
    $(CC) /c $(CFLAGS_G) $(CFLAGS_D) /FoCOUNTCH.obj COUNTCH.C
!ELSE
    $(CC) /c $(CFLAGS_G) $(CFLAGS_R) /FoCOUNTCH.obj COUNTCH.C
!ENDIF
```

This rule contains a conditional statement with two commands. The first command is for debug builds, and the second command is for release builds. You will edit the second (release) command. The release command uses the following macros defined earlier in the makefile:

Macro	Definition
CC	The name of the C compiler
CFLAGS_G	Global options for C compiles
CFLAGS_R	Release options for C compiles

To optimize only COUNTCH.C for time, place the /Ot compiler option after \$(CFLAGS_R). The resulting command is:

```
$(CC) /c $(CFLAGS_G) $(CFLAGS_R) /Ot /FoCOUNTCH.obj COUNTCH.C
```


There is no way to predict if the C option macros contain the /Os option, which would turn off /Ot, or if they contain any other option. To handle this potential problem, the new option must be placed at the end because the option specified last takes precedence. The compiler options, such as /Ot, and NMAKE macros, such as CFLAGS_G, are case sensitive and must appear exactly as shown.

Warning After this modification, PWB can still understand this makefile as a PWB makefile. However, if you make changes beyond adding options to individual command lines, PWB may no longer recognize the file as a PWB makefile. If this happens, you can delete the makefile and re-create it, or you can use it as a non-PWB makefile. For more information on using non-PWB makefiles, see “Using a Non-PWB Makefile” on page 61.

Save your changes to the makefile by choosing Save from the File menu. You can now reopen the project and rebuild COUNT with the custom options.

3.2 The Program Build Process

This section explains the correspondence between projects and makefiles. This process is relatively automatic. If you do nothing out of the ordinary, you will never have to modify its default operation.

Most programmers encounter situations that require customized build options. Read this section to understand how the utilities work with PWB. You can return to this material when you have special requirements that are not handled by PWB’s default build rules.

Figure 3.2 illustrates the PWB build process.

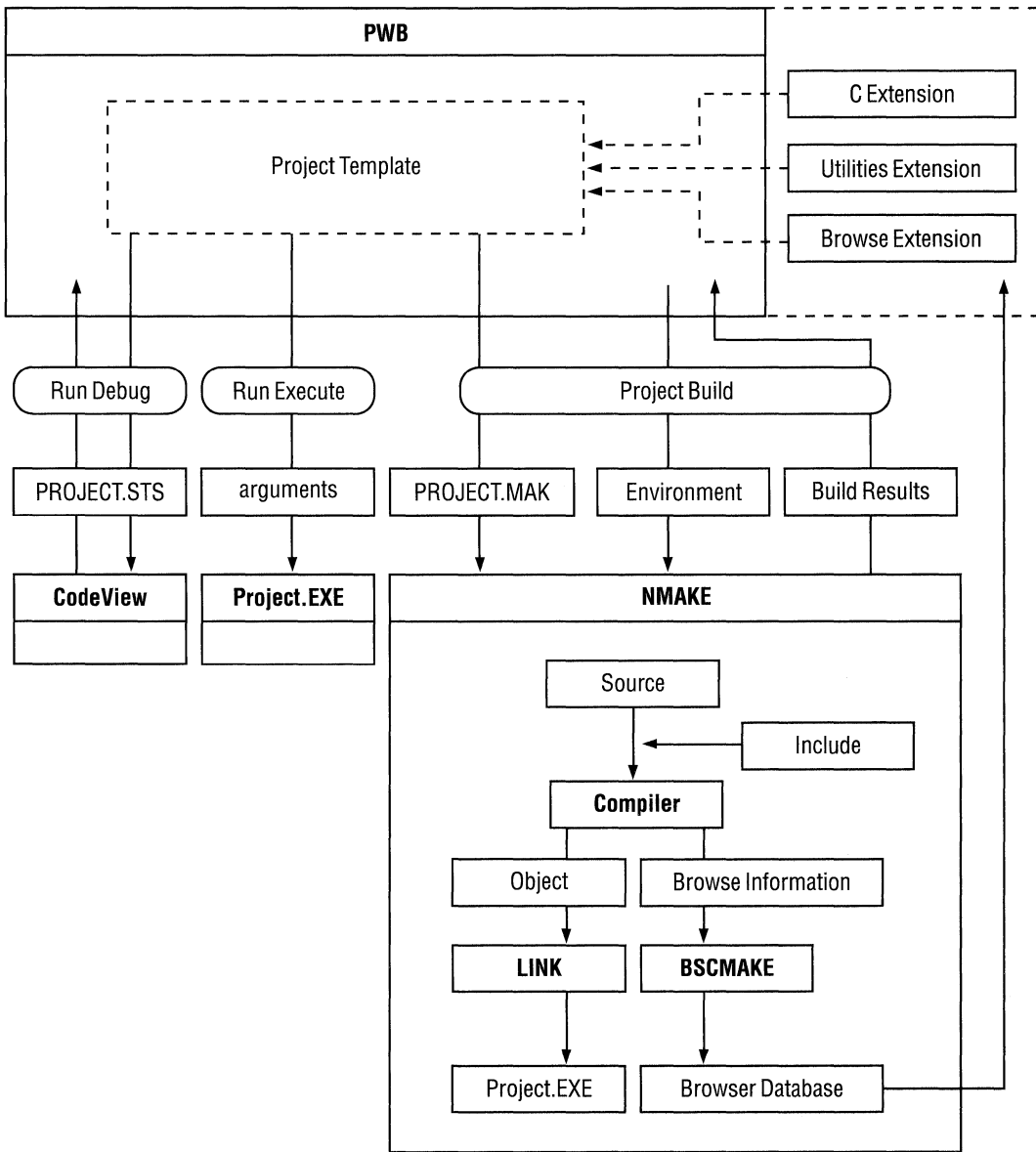


Figure 3.2 The PWB Build Process

When you save your project by choosing the Save button in the Edit Project dialog box, PWB uses the list of files along with the rules in the selected project template to scan for dependencies and write the project makefile.

When you choose the Build or Rebuild All command from the Project menu, PWB releases as much memory as possible and passes the makefile to NMAKE, which builds the project.

NMAKE stops at the end of the first build step that produces an error (as opposed to a warning) or at the end of a successful build. In either case, NMAKE returns the results of the build to PWB along with a log of any errors and warnings. For more information about NMAKE, see Chapter 18, “Managing Projects with NMAKE.”

PWB saves the output of the build for you to view in the Build Results window or to step through when you choose the Next Error (SHIFT+F3), Previous Error (SHIFT+F4), and Goto Error commands on the Project menu. You can run the program, set program arguments, and debug the program by choosing commands in the Run menu.

If you have turned on the generation of browser information, PWB builds the browser database when you build the program. Once you have a browser database, you can use the commands in the Browse menu to navigate your program’s source files and examine the structure of your program. For more information, see “Using the Source Browser” on page 96.

Extending a PWB Project

Makefiles that are not written by PWB often contain utility targets that are not used in the process of building the project itself. These targets are used to clean up intermediate files, perform backups, process documentation, or automate other tasks related to the project. You can extend a PWB makefile to perform these kinds of tasks by adding new rules. These additional rules must be placed in a special section of the project makefile.

In the following example you will add a section that creates a file with information about the project. This file has the same base name as the project and the extension .LST. It lists the files in the project and the major options used for the build. This example section can be used with any PWB project.

Use the COUNT project to see how to add a custom section. If you have been following the tutorial, this project is already open in PWB.

► To add a custom section to the PWB makefile:

1. From the Project menu, choose Close Project.

This step is crucial because PWB disables modification of the project makefile until the project is closed or a different project is opened. (This restriction does not apply to non-PWB project makefiles.)

2. From the File menu, choose the Open command and open the COUNT.MAK file in the editor.
3. Press CTRL+END to move the cursor to the end of the makefile.
4. Type this comment line *exactly* as shown:

```
# << User_supplied_information >>
```

TIP You can copy this line from help. Type ALT+A USI F1, and then copy and paste into the makefile.

You must put the number sign (#) in column one and type the contents of the line exactly as shown, including capitalization. Failing to type this line accurately will make the project unrecognizable to PWB or allow PWB to change your custom build information in unexpected ways.

NMAKE requires space between rules. Therefore, you should separate this line from the lines above it by one blank line. Similarly, you should leave at least one line between the separator and your custom build rules. For more information about NMAKE and the syntax of makefiles, see Chapter 18, “Managing Projects with NMAKE.”

This comment line is used by PWB as a separator. Anything above this comment is regarded as belonging to PWB, and you should not edit the information there. The exception is to add options to individual command lines, as described in “Changing Options for Individual Modules” on page 54. Anything in the makefile after the separator is your information, and PWB ignores it. NMAKE, however, processes the entire file.

TIP This section is in the example file EXTRA.TXT.

Now that you have a separator to show PWB where your custom information starts, you can add the custom information. The separator and custom section is included in the following text:

```
# << User_supplied_information >>

# Example 'user section' for PWB project makefiles,
# used in the PWB Tutorial.
#
# NOTE: This is not a standalone makefile.
#       Append this file to makefiles created by PWB.
#
# This user section adds a new target to build a project
# listing that shows the build type, options, and a list
# of files in the project.
#

!IFDEF PROJ
!ERROR Not a standalone makefile.
!ENDIF
```

```
!IF $(DEBUG)
BUILD_TYPE = debug
!ELSE
BUILD_TYPE = release
!ENDIF

# Project files and information-list target
#
$(PROJ).lst : $(PROJFILE)
    @echo < Project Name:      $(PROJ)
Build Type:      $(BUILD_TYPE)
Program Arguments: $(RUNFLAGS)
Project Files
    $(FILES: =^
    )
C Compiler Options
    Global: $(CFLAGS_G)
    Debug:  $(CFLAGS_D)
    Release: $(CFLAGS_R)
Link Options
    Global: $(LFLAGS_G)
    Debug:  $(LFLAGS_D)
    Release: $(LFLAGS_R)
    Debug Libraries: $(LLIBS_D)
    Release Libraries: $(LLIBS_R)
<<KEEP
```

The custom section of a PWB makefile can use any of the information defined by PWB. This example takes advantage of many macros defined by PWB. For example, the PROJFILE macro, which contains the name of the project makefile, is used as the dependent of the listing file so that the listing is rebuilt whenever the project makefile changes.

In addition, this custom section uses many features of NMAKE including macros, macro substitution, preprocessing directives, and inline files. For more information about NMAKE and makefiles, see Chapter 18, “Managing Projects with NMAKE.”

► **To rebuild using the custom options:**

TIP If PWB fails to recognize your customized project, you may have typed the separator comment incorrectly.

1. Choose Open Project from the Project menu and reopen the COUNT project.
2. From the Project menu, choose Build Target.
3. Type the name of the new target COUNT.LST in the Target text box, and then choose OK.

PWB informs you that the build options have changed and asks if you want to rebuild everything.

4. Choose Yes to confirm that you want to rebuild everything.

TIP To open a file from the list, put the cursor on the first character of the name and type ALT+A F10.

The project information file that is created shows the project name, indicates whether the build is a debug or release build, lists the files in the project, and lists the compiler and linker options used for the build.

Using a Non-PWB Makefile

PWB makefiles are highly structured and stylized makefiles that are generated from the rules in the project template and a list of files that you supply. Many projects have existing makefiles that PWB can't read because they do not have this stylized structure. These makefiles are called non-PWB or "foreign" makefiles.

You can still take advantage of many of PWB's project features with non-PWB makefiles. The features that cannot be used are shown as unavailable menu items. Note that a PWB makefile is not required to use the Source Browser—all you need to have is a browser database. For information on building a browser database, see "Building Databases for Non-PWB Projects" on page 104 and Chapter 21.

Before continuing, consider the following makefile, which builds a version of the COUNT project:

```
#
# CNT.MAK - A simple non-PWB makefile for building
# the PWB tutorial example program COUNT.EXE .
#
# NOTE: The LIBS macro assumes the default
# library name. If you have installed with different
# names, you must change the LIBS macro.
#

#
# Macros
#
CC = cl
CFLAGS = /Ox /qC
LFLAGS = /NOD:SLIBCE.LIB /NOE /NOI /EXE /FAR /PACKC
LINKER = link
OBJS = COUNT.OBJ COUNTBUF.OBJ COUNTCH.OBJ
STDOBJS = SETARGV.OBJ
LIBS = SLIBCE

#
# The "all" target.
# Building 'all' builds COUNT.EXE.
#
all: COUNT.EXE
```

```
#
# The file suffixes NMAKE needs to "know" about
# for this project.
#
.SUFFIXES:
.SUFFIXES: .obj .c

#
# An inference rule to make an object file from a
# C source file.
#
.c.obj :
    $(CC) /c $(CFLAGS) /Fo$@ $<

#
# The description block for building COUNT.EXE
# from the object files and libraries.
#
COUNT.exe : $(OBJS)
    $(LINKER) $(LFLAGS) $(OBJS) $(STDOBJS),@,,$(LIBS);

# The 'clean' target. Delete intermediate files
# that might be clutter after a release build
#
clean :
    -del *.obj
    -del *.bak
    -del *.tmp
    -del *.map
```

This makefile is written for NMAKE. Even though PWB cannot read it as a PWB makefile, you can use CNT.MAK as a project makefile in PWB without having to change it.

CNT.MAK defines two primary targets, `all` and `clean`. By default, NMAKE builds the first target in your makefile. The first target is commonly called `all` and is used to build the main targets of a project. Other targets in the makefile are used to build the `all` targets or describe additional functionality. For example, the `clean` target in this makefile deletes some intermediate files from disk.

► **To use CNT.MAK in PWB:**

1. From the Project menu, choose Open Project.
2. Select CNT.MAK.
3. Turn on the Use as a Non-PWB Makefile check box.
The Open Project dialog box appears.
4. Choose OK.

Note A PWB makefile cannot be edited or modified when it is the open project. However, PWB does not disable modification of non-PWB makefiles. You can edit a non-PWB makefile, even when it belongs to the currently open project.

The `LIBS` macro in `CNT.MAK` assumes the default library name. If you have installed with different names, or you want to use a different library, you must change the `LIBS` macro to contain the name of the library you are using.

You can now use the Build, Rebuild All, and Build Target commands from the Project menu. The Build and Rebuild All commands work as they do with a PWB makefile by building the first target. However, the Language Options commands and the LINK Options command on the Options menu are unavailable. You set options by editing the makefile.

► **To build the `clean` target:**

1. From the Project menu, choose Build Target.
PWB displays the Build Target dialog box where you can specify the target name(s).
2. Type `clean` in the Target text box.
3. Choose Build.
PWB builds the `clean` target instead of the first target in the makefile (in this case, `all`).

When you close a non-PWB project, PWB saves the environment, window layout, and file history just as it does for a PWB project.

3.3 Where to Go from Here

This concludes the PWB tutorial section of this manual. If you wish, you can leave PWB by choosing Exit from the File menu (or by pressing `ALT+F4`).

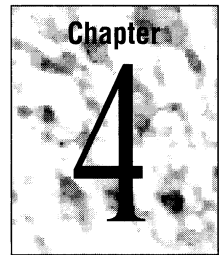
Chapter 4, “User Interface Details,” explains how to start PWB, describes the elements of the user interface, and gives you an overview of the menus.

Chapter 5, “Advanced PWB Techniques,” explains search techniques (including regular-expression searching), describes how to use the browser, and shows how to write PWB macros.

Chapter 6, “Customizing PWB,” describes how to change the behavior of PWB to suit your needs.

Chapter 7, “PWB Reference,” contains an alphabetical reference to PWB menus, keys, functions, predefined macros, and switches.

User Interface Details



This chapter summarizes the PWB user interface. It contains:

- General information on starting PWB.
- Instructions on how to use elements of the PWB screen.
- A description of the indicators on the status bar.
- A summary of every PWB menu command.
- Instructions on how to use menus and dialog boxes.

4.1 Starting PWB

You can start PWB in either of the following ways:

- From the Windows Program Manager
- From the operating-system command line

From the Command Line

► **To start PWB from the command line:**

- At the operating-system prompt, type:

PWB *[[options]]* *[[filename]]*

PWB starts with its default startup sequence.

For a complete list of PWB options and their meanings, see “PWB Command Line” on page 141. Sometimes, you will want to modify the default startup sequence. The following procedures are examples of how you can start PWB to accommodate different circumstances.

► **To start PWB with an existing PWB project:**

- Type `PWB /PP project.mak`

PWB opens the specified project and the files that you were working on with the project.

► **To start PWB with the project you used in your last session:**

- Type `PWB /PL`

As with the previous option, the `/PL` option opens a project and arranges your screen as it was when you left PWB.

► **To start PWB quickly for editing a file such as CONFIG.SYS:**

- Type `PWB /DAS /t CONFIG.SYS`

This command suppresses autoloading of extensions and status files (`/DAS`). It also tells PWB not to remember `CONFIG.SYS` for the next PWB session (`/t CONFIG.SYS`).

Using the Windows Program Manager

Microsoft Windows offers features that can enhance program development, particularly if you plan to develop Windows applications. You can edit and build your application in a “DOS Box” and then immediately run it under Windows.

When you install Microsoft C/C++ on a computer running Windows, the `SETUP` program provides a PWB icon in the Microsoft C/C++ 7.0 Program Group and a `.PIF` file for running PWB successfully under Windows. These files, `PWB.ICO` and `PWB.PIF`, are located in the `\C700\BIN` directory (assuming you accepted the default root directory name supplied by `SETUP`).

To start PWB under Windows, double-click the PWB icon.

You can add a Program Item to the Program Manager for each project you are working on. Use the PIF editor to open `PWB.PIF`, and then choose `Save As` on the `File` menu to create a `.PIF` file with the same base name as your project. Next, use the `Optional Parameters` text box to specify the `/PF` or `/PP` options and the name of the project makefile.

Using the Windows File Manager

When programming, you are often concentrating on which file or project you want to work on and would prefer that the computer provide the right tool for the job. With the Windows File Manager, you can associate certain types of files with the commands that operate on those files. Therefore, when you double-click the file-name in the File Manager, the right tool starts with the correct command-line options.

You can associate project makefiles (.MAK files) with the PWB .PIF file. Double-clicking a project makefile then starts PWB and opens that project, source files and all.

► **To associate PWB with .MAK files:**

1. Select any file in the File Manager with the extension .MAK.
2. From the File menu, choose Associate.
3. Type the command `PWB.PIF /PP` in the dialog box. (Make sure that your PWB.PIF file specifies a question mark (?) in the Optional Parameters text box.)

Now when you double-click a project makefile, the File Manager automatically starts PWB, and PWB opens that project.

Note Be sure you have set your PATH, INIT, and TMP environment variables prior to starting Windows so PWB can find all its files.

4.2 The PWB Screen

Figure 4.1 shows the PWB display. The table which follows it describes each of the user interface elements.

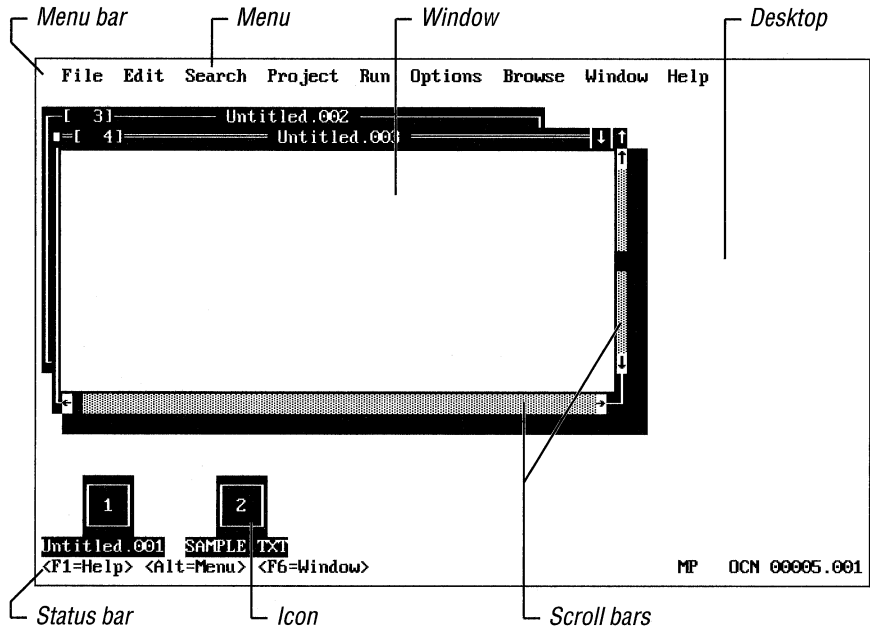


Figure 4.1 User Interface Elements

Name	Description
Menu bar	Lists available menus.
Menu	Lists PWB commands.
Desktop	Background area.
Icon	Displays a window in compact form.
Window	Contains source code; displays Help, browser results, build results, or error messages.
Scroll bars	Change position in file or list.
Status bar	Shows command buttons for the mouse and shortcut keys; summarizes commands and file and keyboard status.

Figure 4.2 shows a PWB window. The table which follows it describes each of a window's elements.

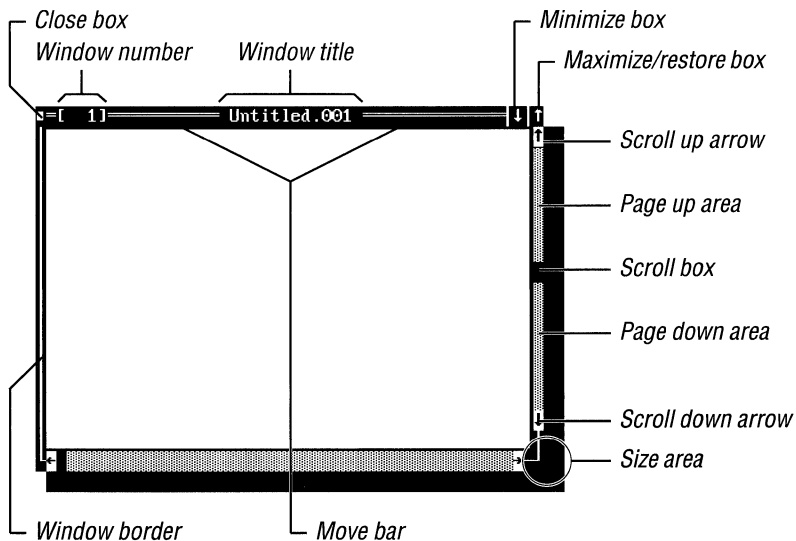


Figure 4.2 Window Elements

Name	Description
Window border	Moves window. Drag to move the window.
Close box	Closes the window. Click to close the window.
Window number	Identifies window. Press ALT+ <i>number</i> to move to that window.
Window title	Indicates window contents, a filename, or pseudofile title.
Minimize box	Shrinks window to an icon. Click to minimize the window.
Maximize/Restore box	Enlarges window to maximum size or restores window to its original size.
Scroll up arrow	Scrolls up by lines. Click to scroll up.
Page up area	Scrolls up by pages. Click to page up.
Scroll box	Indicates relative position in the file. Drag to change position.
Page down area	Scrolls down by pages. Click to page down.
Scroll down arrow	Scrolls down by lines. Click to scroll down.
Size area	Sizes window. Drag to size the window.
Move bar	Moves window. Drag to move the window.

Figure 4.3 shows the PWB status bar. The table which follows it describes each of the status bar's elements.

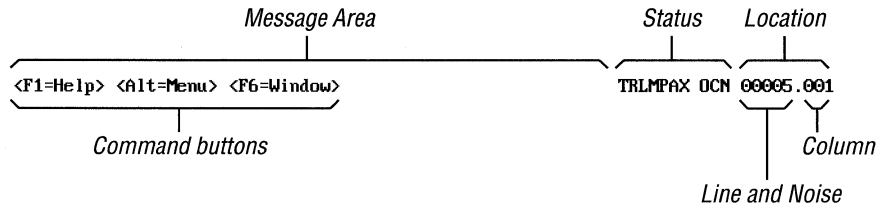


Figure 4.3 Status Bar Elements

Name	Description
Message area	Shows command buttons for the mouse and shortcut keys, and summarizes commands.
Status	Indicates current file, editor, and keyboard status, as described in the following table.
Location	Shows the location of the cursor in the file.
Command buttons	Show common commands and shortcut keys. Click the button or press the key to execute the command.
Line	Indicates the line at the cursor. When scanning a file during a search or when loading a file, PWB displays the current line in the line indicator as specified by the Noise switch.
Column	Indicates the column at the cursor.

The status area of the status bar displays one of the following letters to indicate the corresponding status.

Letter	Description
T	File is temporary and is not recorded in the PWB status file.
R	File is no-edit (read-only); modification is disabled.
L	Line endings in the file are linefeed characters only.
M	File is modified.
P	File is a pseudofile.
A	Meta prefix (F9) is active.
X	Macro recording is turned on.
O	Overtime mode is enabled. In insert mode, no indicator appears.
C	CAPS LOCK is on.
N	NUM LOCK is on.

Figure 4.4 shows the Window menu with the PWB Windows cascaded menu pulled down. The table which follows it describes each element of a menu.

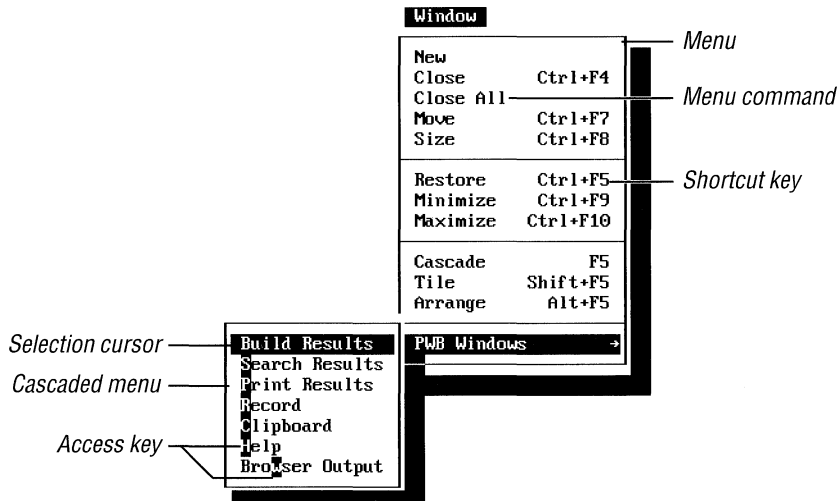


Figure 4.4 PWB Menu Elements

Name	Description
Menu	Displays a list of commands.
Menu command	Executes the command. When the command is dimmed, it is unavailable.
Shortcut key	Executes the command directly and bypasses the use of the menu. Press the key to execute the command.
Cascaded menu	Lists a group of related commands. The command for a cascaded menu has a small right arrow after the command. To open a cascaded menu, click the command or move the selection cursor to the command and press the RIGHT ARROW key. To close an open cascaded menu, press the LEFT ARROW key.
Access key	Executes the command. Press the highlighted letter key to execute the command.
Selection cursor	Indicates the selected command. Press the UP ARROW and DOWN ARROW keys to move the selection cursor. Press ENTER to execute the command.

4.3 PWB Menus

PWB commands are organized into menus; the menu names appear along the menu bar at the top of the screen. When a menu or command is selected, PWB displays a brief description of the selected menu on the status bar. To get more information about a menu or command, point the mouse cursor to the name and click the right mouse button, or highlight the name by using the arrow keys and then press F1.

File

The File menu provides commands to open, close, and save files. You can switch to any open PWB file or find a specific file on your disk. You can also print a selection, a file, or a list of files.

Command	Description
New	Start a new file
Open	Open an existing file
Find	Locate a file or list of files on disk
Merge	Merge one or more files into the current file
Next	Open the next file in the list of files specified on the command line
Save	Save the current file
Save As	Save the current file with a different name
Save All	Save all modified files
Close	Close the current file
Print	Print a selection, the current file, or a list of files
DOS Shell	Temporarily exit to the operating-system
All Files	List all open files in PWB
Exit	Leave PWB

Edit

The Edit menu provides commands to manipulate text, set the selection mode, and record macros.

Command	Description
Undo	Reverse the effect of your recent edit
Redo	Reverse the effect of the last Undo
Repeat	Repeat the last edit
Cut	Delete selected text and copy it to the clipboard
Copy	Copy selected text to the clipboard
Paste	Insert text from the clipboard
Delete	Delete selected text without copying it to the clipboard
Set Anchor	Save the current cursor position
Select To Anchor	Select text from the anchor to the cursor
Stream Mode	Set stream selection mode
Box Mode	Set box selection mode
Line Mode	Set line selection mode
Read Only	Toggle the PWB no-edit state (to prevent accidental modification or to allow modification)
Set Record	Define a macro name and its shortcut key
Record On	Record commands for a macro

Search

The Search menu provides commands to perform single-file and multifile text and regular-expression searches. You can do single-file and multifile find-and-replace operations. You can define and jump to marks or go to specific lines.

Command	Description
Find	Search for an occurrence of a text string or pattern
Replace	Search for a string or pattern and replace it with another
Log	Turn multifile searching on and off
Next Match	Move to the next match
Previous Match	Move to the previous match
Goto Match	Go to the match at the cursor in the Search Results window
Goto Mark	Move to a mark or line number
Define Mark	Set a mark at the cursor
Set Mark File	Open or create a mark file

Project

The Project menu provides commands to open and create projects, build a project or selected targets in the project, and determine the location of build errors and messages.

Command	Description
Compile File	Compile the current source file
Build	Build the project
Rebuild All	Build all files in the project (even those that have not been modified)
Build Target	Build specific targets in the project
New Project	Create a new project
Open Project	Open an existing project
Edit Project	Change the list of files in the project
Close Project	Remove the current project from memory without changing its contents
Next Error	Move to the next error
Previous Error	Move to the previous error
Goto Error	Move to the error at the cursor in the Build Results window

Run

The Run menu provides commands to set arguments for the project's program, run and debug the program, run operating-system commands, and add or run custom Run menu commands.

Command	Description
Execute	Run the current program
Program Arguments	Specify commands passed to your program for Execute or Debug
Debug	Run CodeView for the current program
Run DOS Command	Perform any single DOS task without exiting PWB
Customize Run Menu	Add commands to the Run menu

The custom commands that you add to the Run menu appear after the Customize Run Menu command.

Options

The Options menu provides commands to set environment variables for use within PWB, customize the look and behavior of PWB, and assign keys to commands. For projects, you can set the build type, customize the project template, and set compiler and utility options.

Command	Description
Environment Variables	View and modify environment variables
Key Assignments	Assign keys that invoke functions and macros
Editor Settings	Change the setting of any PWB switch
Colors	Change screen colors
Build Options	Specify whether the program is built as a debug or release version; specify a build directory
Project Templates	Cascaded menu of commands for project templates
Language Options	Cascaded menu of compiler options commands

The Project Templates cascaded menu provides the following commands to manage project templates:

Command	Description
Set Project Template	Changes the run-time support and project template
Customize Project Template	Modify the current project template
Save Custom Project Template	Save the current template as a new, custom template
Remove Custom Project Template	Delete custom project templates

The Language Options cascaded menu provides the following commands for setting compiler options:

Command	Description
C Compiler Options	Set C compiler options
C++ Compiler Options	Set C++ compiler options

Note Additional languages are listed when their PWB extension is loaded.

The following commands appear when the utilities extension (PWUTILS) is loaded:

Command	Description
LINK Options	Set linker options for your project
NMAKE Options	Set options for NMAKE when it builds the project
CodeView Options	Set options for CodeView when debugging the project

The following command appears when the browser extension (PWBROWSE) is loaded:

Command	Description
Browse Options	Define the way the Source Browser database is built

Browse

The Browse menu provides the commands for the PWB Source Browser. You can select a browser database. You can jump to specific definitions or symbols in your project and view complex relationships among program symbols. You can also view your program as an outline, function-call tree, or class-inheritance tree.

Command	Description
Open Custom	Open a custom browser database, open the project database, or save the current database
Goto Definition	Locate the definition of any symbol in your source code
Goto Reference	Locate the references to any name in the browser database
View Relationship	Query the browser database
List References	Display a list of functions that call each function and show the use of each variable, type, macro, or class
Call Tree (Fwd/Rev)	View which functions call other functions
Function Hierarchy	Display a program outline
Module Outline	Display an outline of program modules
Which Reference?	Display a list of possible references for the ambiguous reference at the cursor
Class Tree (Fwd/Rev)	View the class inheritance tree
Class Hierarchy	View the hierarchy of classes
Next	Find the next definition or reference
Previous	Find the previous definition or reference
Match Case	Define whether or not browse queries are case sensitive

Window

The Window menu provides commands to manipulate and navigate windows in PWB.

Command	Description																
New	Duplicate the active window																
Close	Close the active window																
Close All	Close all windows																
Move	Start window-moving mode for the active window																
Size	Start window-sizing mode for the active window																
Restore	Restore a minimized or maximized window to normal size																
Minimize	Shrink the active window to an icon																
Maximize	Enlarge windows to maximum size																
Cascade	Arrange windows to show all their titles																
Tile	Arrange windows so that none overlap																
Arrange	Organize windows in a useful configuration for viewing Help, source code, and Build Results																
PWB Windows	Cascaded menu that lists the following special PWB windows:																
	<table border="1"> <thead> <tr> <th>PWB Window</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>Build Results</td> <td>View the results of builds</td> </tr> <tr> <td>Search Results</td> <td>View the results of logged searches</td> </tr> <tr> <td>Print Results</td> <td>View the results of print operations</td> </tr> <tr> <td>Record</td> <td>View, edit, save recorded macros</td> </tr> <tr> <td>Clipboard</td> <td>View the PWB clipboard</td> </tr> <tr> <td>Help</td> <td>Access the Help system</td> </tr> <tr> <td>Browser Output</td> <td>View the results of browser queries</td> </tr> </tbody> </table>	PWB Window	Description	Build Results	View the results of builds	Search Results	View the results of logged searches	Print Results	View the results of print operations	Record	View, edit, save recorded macros	Clipboard	View the PWB clipboard	Help	Access the Help system	Browser Output	View the results of browser queries
PWB Window	Description																
Build Results	View the results of builds																
Search Results	View the results of logged searches																
Print Results	View the results of print operations																
Record	View, edit, save recorded macros																
Clipboard	View the PWB clipboard																
Help	Access the Help system																
Browser Output	View the results of browser queries																
1 window1	Move to window <i>n</i> .																
...																	
5 window5																	
All Windows	View a list of all open windows																

The All Windows command does not appear until the full list of open windows is too long to fit on the menu.

Help

The Help menu contains commands to access the Microsoft Advisor Help system. You can see the index or table of contents for the system, get context-sensitive Help, and perform global plain-text searches in the Help.

Command	Description
Index	Display a list of available indexes
Contents	Display a table of contents for each component of the Help system
Topic	Display Help about the item or keyword at the cursor
Help on Help	Display information on how to use Help
Next	Display the next Help screen that has the same name as the topic you last viewed
Global Search	Search all open Help files for a string or regular expression
Search Results	View the results of the last global Help search
About	Display the PWB copyright and version number

4.4 Executing Commands

PWB commands appear in menus and as “buttons.” You can execute these commands in two ways:

- With a Microsoft Mouse or any fully compatible pointing device
You perform mouse operations by “clicking”—moving the mouse cursor to the specified item and briefly pressing the left mouse button. “Double-click” by pressing the left button twice, quickly. Always use the left mouse button unless specifically instructed otherwise.
- With the keyboard

4.5 Choosing Menu Commands

► **To choose a menu command with the mouse:**

1. Click the menu name to open the menu.
2. Click the command.

► **To choose a menu command from the keyboard:**

1. Press the ALT key to activate the menu bar.
2. Press the highlighted character in the menu name (such as F for File).

An alternative is:

1. Press the ALT key to activate the menu bar.
2. Use the RIGHT ARROW and LEFT ARROW keys to select a menu.
3. Press ENTER to open the menu.
4. Press the highlighted character in the command name (such as S for Save in the File menu), or use the UP ARROW and DOWN ARROW keys to select the command and then press ENTER.

There are several ways to close an open menu without executing a command.

► **To close a menu without executing a command:**

- Click outside of the menu.
- Press ESC.
- Press ALT twice.

When a menu command is dimmed (rather than black), it is unavailable. For example, when no windows are open, the Close command on the File menu is unavailable. If a command you want to use is unavailable, you must perform some other action or complete a pending action before you can invoke that command.

Shortcut Keys

Some commands are followed by the names of keys or key combinations. Press the shortcut key to execute the command immediately. You don't have to go through the menu. For example, press SHIFT+F2 to execute the Save command, which saves the current file.

All menu commands with shortcut keys and many other menu commands invoke predefined PWB macros to carry out their action. You can change the key or add new shortcut keys for these commands by assigning a key to the predefined macro. For a complete list of predefined macros and their corresponding menu commands, see "Predefined PWB Macros" on page 222. For more information on assigning keys, see "Changing Key Assignments" on page 119.

Many PWB functions have been assigned to keys besides those listed on the menus. Choose the Key Assignments command on the Options menu to view a list of functions and macros and their assigned keys.

Buttons

You can often execute commands by using buttons or boxes, which are areas of the screen that perform an action when you click them or select them from the keyboard. For example, the rectangle at the upper-left corner of a window is the “close box.” Clicking this box with the mouse closes the window.

A command name surrounded by angle brackets (< >) appearing on the status bar or in a dialog box is a button. The following buttons are on the status bar when you first start PWB:

<General Help> <F1=Help> <Alt=Menu>

The General Help button brings up a screen that explains how to use the Help system. The other two buttons remind you of PWB functions: F1 summons Help, and ALT activates the menu bar. Clicking one of these buttons with the mouse performs the same function as pressing the key.

When you have opened more than one window, PWB displays the following buttons:

<F1=Help> <Alt=Menu> <F6=Window>

Click the Window button or press F6 to move to the next window.

When a menu is selected or a dialog box is displayed, an informative message appears on the status bar. While PWB displays this message, no buttons are available and clicking the status bar does nothing.

Dialog Boxes

When a menu command is followed by an ellipsis (...), PWB needs more information before executing the command. You enter this information in a dialog box that appears when you choose the command.

Dialog boxes can contain any of the items in Figure 4.5.

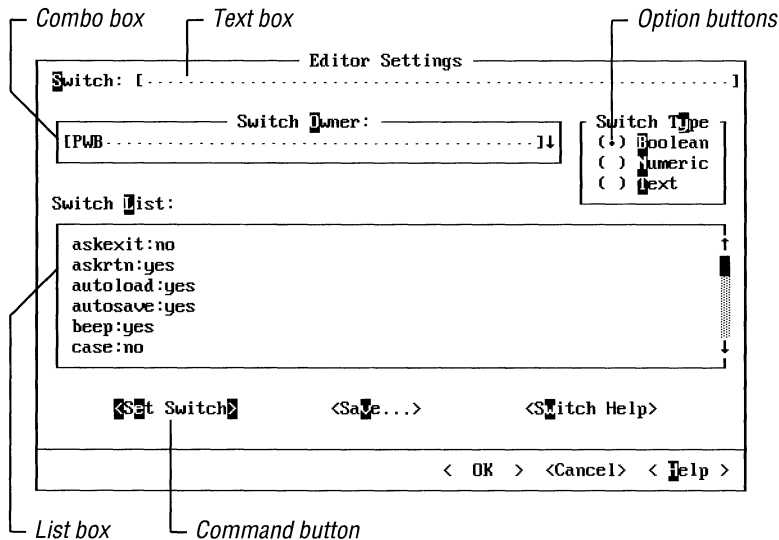


Figure 4.5 Dialog Box Elements

Option Button

A button that you select from a list of mutually exclusive choices. Click the one you want, press its highlighted letter, or use the arrow keys to move among the choices.

Text Box

An area in which you can type text. You can move the cursor within the text box by clicking the location with the mouse or by pressing the LEFT ARROW and RIGHT ARROW keys. You can toggle between insert and overtype mode by pressing the INS key. To select text for deletion, click and drag the mouse over the text or press SHIFT plus an arrow key. Press DEL to delete the text, or type new text to replace the highlighted text.

List Box

A box displaying a list of information (such as the contents of the current disk directory). If the number of items exceeds the visible area, click the scroll bar to move through the list or press PGUP, PGDN, or the arrow keys. You can move to the next item in the list that starts with a particular letter by typing that letter.

Combo Box

The combination of a text box and a drop-down list box. You can type the name of an item in the text box or you can select it from the list.

To open the list, click the highlighted arrow, or press ALT+DOWN ARROW or ALT+UP ARROW. You can then click the item or press the arrow keys to select the item you want. You can also press the first letter of an item to select it. When

you have selected an item, click the highlighted arrow or press ALT+DOWN ARROW or ALT+UP ARROW to close the list.

Command Button

A button within angle brackets (< >) that invokes a command. Choose the OK button to accept the current options, or choose the Cancel button to exit the dialog box without changing the current options. Choose the Help button to see Help on the dialog box.

If one of the command buttons in a dialog box is highlighted, press ENTER to execute that command. You can also click a command button to execute that command. If a button contains an ellipsis (...), it indicates that another dialog box will appear when you choose it.

Check Box

An on/off switch. If the box is empty, the option is turned off. If it contains the letter X, the option is turned on. Click the box with the mouse, or press the SPACEBAR or the UP ARROW and DOWN ARROW keys to toggle a check box on and off.

Key Box

A pair of braces ({ }) that allows you to enter a key by pressing the key. The key box is always followed by a text box where you can type the name of the key.

When the cursor is in the key box (between the braces), most keys lose their usual meaning, including ESC, F1, and the dialog box access keys. The key you press is interpreted as the key to be specified. Only TAB, SHIFT+TAB, ENTER, and NUMENTER retain their usual meaning. To specify one of these keys, type the name in the text box.

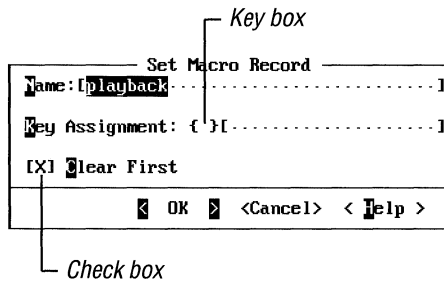


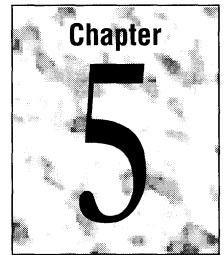
Figure 4.6 Key Box and Check Box

Clicking a dialog-box item either selects it (a text box, for example) or toggles its value (a check box or option button). You can also move among items with the TAB and SHIFT+TAB keys.

Dialog boxes usually contain access keys, identified by highlighted letters. Pressing an access key is equivalent to clicking that item with the mouse or moving to it by pressing `TAB` or `SHIFT+TAB`, and then pressing `ENTER`. Although some access keys are uppercase and others lowercase, dialog boxes are not case sensitive. Therefore, you can type either an uppercase or a lowercase character.

Note When the cursor is in a text box, access keys are interpreted as text. You must press `ALT` along with the highlighted letter. Pressing `ALT` is also required in list boxes because typing a letter by itself moves the cursor to the next item that starts with that letter.

Advanced PWB Techniques



This chapter introduces you to some of the powerful features in the Programmer's WorkBench. It is not an exhaustive discussion of all the ways to use PWB. However, it can provide a starting point for you to begin your own investigation of PWB using the information in the Microsoft Advisor and in Chapter 7, "Programmer's WorkBench Reference."

This chapter contains:

- Find and search-and-replace techniques, including marks and regular expressions.
- How to use the PWB Source Browser.
- How to execute PWB functions and macros.
- An overview of PWB macros, macro recording, and the macro language.

5.1 Searching with PWB

PWB offers the following ways to search your files for information:

- Visually inspecting code, moving with the cursor or the PGUP and PGDN keys. This method is most effective either when you are familiarizing yourself with some old code or after you have switched from CodeView back to PWB and want to examine the local impact of a proposed change.
- Searching with text strings. When you have a specific string in mind (for example, `FileName`), you can find, in sequence, all the references to the name in your file.
- Searching with regular expressions. Regular expressions describe patterns of text. This is useful when you have a number of similarly named program symbols and you'd like to see them all in succession.

For example, Windows API (application programming interface) names are made up of multiple words; the start of each new word is shown as a capital letter (for example, `GetTextMetrics`). With this in mind, you might search for

a string optionally starting with spaces and the letters “GetText” followed by any uppercase letter. This is expressed with a regular expression such as `*GetText[A-Z]`, which means zero or more spaces (using the `*` operator after a space), followed by `GetText`, followed by any uppercase letter (using a character class).

- Searching multiple files with text strings or regular expressions. A multifile search is called a “logged search.” Instead of finding one match, PWB finds all matches in one operation. You can then browse the results of the search.
- Using the Source Browser. The Source Browser enables you to perform faster and more sophisticated searches than plain text searches because it maintains a complete database of relationships between program symbols. For example, to find all occurrences of `FileName` in your entire program (regardless of the number of files in the program), you can use the View References command from the Browse menu.

The Source Browser has many more capabilities than just finding symbols. It can also produce call trees and perform sophisticated queries on the use-and-definition relationships among functions, variables, and classes in your program.

These searching techniques are discussed in detail in the following sections.

Searching by Visual Inspection

If you think you’re close to the text you want to see, don’t try a fancy search—use the `PGUP` or `PGDN` key. It’s often faster. One trick you can use to speed up this method of searching is to leave a trail in the form of marks (names associated with file locations).

Using Marks

PWB lets you set named marks at any location in your file by using the Define Mark command from the Search menu or by using the **Mark** function. You can access these locations by name using the Goto Mark command or the **Mark** function.

TIP You can also use marks when you are writing new code and want to come back and fill in sections.

For example, if you are revising a preexisting program and don’t fully understand all the algorithms, you might leave a mark at each location in the code you want to examine more closely. That way, you can revise the sections of the program that you do understand, get a feel for the flow of the program, and then come back to the marked areas for further research.

To save marks between PWB sessions, create a mark file using the Set Mark File command from the Search menu.

Using the Find Command

The Find command on the Search menu allows you to search a file using text strings and regular expressions.

TIP The searching functions are named **Psearch** (F3) and **Msearch** (F4).

Find can help you locate any string of text in any file you specify. PWB usually searches the file you are currently editing. However, it can also search a list of files. This is particularly useful for finding all occurrences of a string in an entire project.

TIP The multifile searching function is named **Mgrep** (not assigned).

The results of a multifile search are logged—that is—put into the Search Results window. To see the logged results of a search, choose Search Results from the PWB Windows cascaded menu. There are two ways to use the information that PWB puts into Search Results:

- You can look at the matches in sequence by choosing Next Match and Previous Match from the Search menu.
- You can go directly to a specific match by moving the cursor to the match listed in the Search Results window and choosing Goto Match from the Search menu. PWB then jumps to the location of the match.

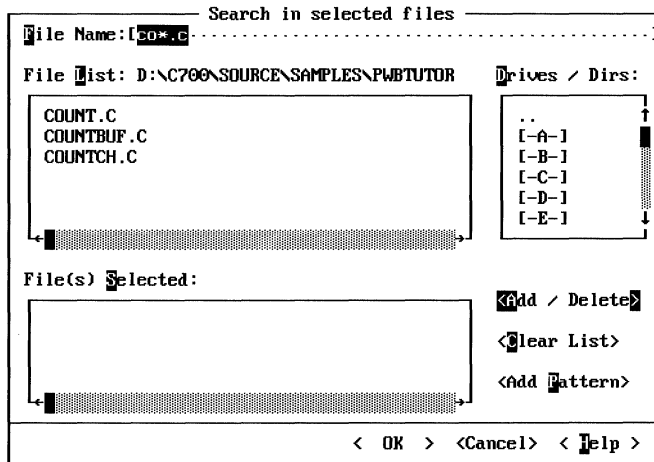
The Match commands on the Search menu work with the Search Results window in exactly the same way that the Project menu's Next Error, Previous Error, and Goto Error commands work with the Build Results window. These Project menu commands are described in "Fixing Build Errors" on page 24.

To illustrate the logged-search technique, suppose you want to locate all functions returning an **int** in the COUNT project's source files.

► **To search all the source files in this project:**

1. From the Search menu, choose Find.
PWB brings up the Find dialog box.
2. Turn on Log Search check box.
3. Type `int` in lowercase.
4. Select the Match Case check box to exclude uppercase or mixed case occurrences of the word.
5. Choose the Files button.

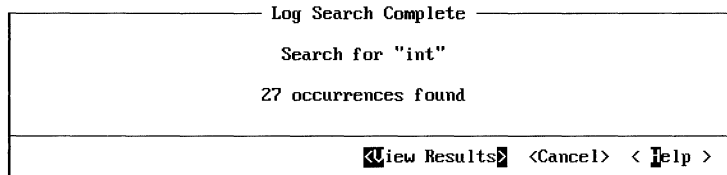
PWB brings up the Search In Selected Files dialog box.



TIP To specify a path from your environment, such as INCLUDE, specify \$INCLUDE: (the environment variable must be in all caps.)

6. Type CO*.C in the File Name text box.
This wildcard specifies all filenames beginning with CO and having the .C extension.
7. Choose the Add Pattern button to add the wildcard to the file list.
8. Return to the File Name text box by clicking the box or by pressing ALT+F.
9. Type COUNT.H in the File Name text box.
Because the default button is Add / Delete, you can press ENTER to add COUNT.H to the file list.
10. Add COUNT.H to the list.
11. Choose OK to start the search.

When PWB finishes the search, it displays the Log Search Complete dialog box.



From this dialog box you can:

- Choose View Results to open the Search Results window.
- Choose Cancel to close the dialog box.

Choose Cancel now (you will open the Search Results window later).

► **To go to the first match:**

- From the Search menu, choose Next Match.

You can step sequentially through all occurrences of the string using the Next Match command. Choose Previous Match to move to the previous occurrence of the string. When you reach the end of Search Results, PWB displays the following message:

End of Search Results

Sometimes, you cannot focus the search narrowly enough to make a sequential scan of Search Results profitable. In this example, you wanted only functions returning **int**, but PWB found many more occurrences of **int**. In these cases, you can examine the results of the search and skip the matches that aren't relevant.

► **To view the Search Results:**

- From the PWB Windows cascaded menu on the Window menu, choose Search Results. PWB opens the Search Results window.

In this window, PWB displays the file, line, and column where the string was located. It also shows as much of the matching line as will fit in the window.

TIP Open the Search Results window to see an overview of all matches from the search.

```

File Edit Search Project Run Options Browse Window Help
Search Results
\700\SOURCE\SAMPLES\PWB\TUTOR Search int
C:\700\SOURCE\SAMPLES\PWB\TUTOR\COUNT.C 6 33: // (A character is defined as printable A
C:\700\SOURCE\SAMPLES\PWB\TUTOR\COUNT.C 11 31: FLAG CountWords( FLAG InWord, int nChars
C:\700\SOURCE\SAMPLES\PWB\TUTOR\COUNT.C 13 5: int Scan;
C:\700\SOURCE\SAMPLES\PWB\TUTOR\COUNT.C 25 1: int main( int argc, char *argv[] );
C:\700\SOURCE\SAMPLES\PWB\TUTOR\COUNT.C 25 11: int main( int argc, char *argv[] );
C:\700\SOURCE\SAMPLES\PWB\TUTOR\COUNT.C 26 1: int CountFile( char *name );
C:\700\SOURCE\SAMPLES\PWB\TUTOR\COUNT.C 29 1: int main( int argc, char *argv[] )
C:\700\SOURCE\SAMPLES\PWB\TUTOR\COUNT.C 29 11: int main( int argc, char *argv[] )
C:\700\SOURCE\SAMPLES\PWB\TUTOR\COUNT.C 32 5: int curArg;
C:\700\SOURCE\SAMPLES\PWB\TUTOR\COUNT.C 54 11: printf( "\n\nEnter file name: " );
C:\700\SOURCE\SAMPLES\PWB\TUTOR\COUNT.C 64 1: int CountFile( char *name )
C:\700\SOURCE\SAMPLES\PWB\TUTOR\COUNT.C 67 5: int nMax;
C:\700\SOURCE\SAMPLES\PWB\TUTOR\COUNT.C 73 11: printf( "\nCan't open %s\n", name )
C:\700\SOURCE\SAMPLES\PWB\TUTOR\COUNT.C 82 24: // Calculate and print the results.
C:\700\SOURCE\SAMPLES\PWB\TUTOR\COUNT.C 83 7: printf( "\n\nFile statistics for %s\n\n"
C:\700\SOURCE\SAMPLES\PWB\TUTOR\COUNT.C 84 7: printf( "\tBytes: %6ld\n", Bytes );
C:\700\SOURCE\SAMPLES\PWB\TUTOR\COUNT.C 85 7: printf( "\tCharacters: %6ld\n", Characte
C:\700\SOURCE\SAMPLES\PWB\TUTOR\COUNT.C 86 7: printf( "\tLetters: %6ld\n", Letters
C:\700\SOURCE\SAMPLES\PWB\TUTOR\COUNT.C 87 7: printf( "\tVowels: %6ld\n", Vowels );
C:\700\SOURCE\SAMPLES\PWB\TUTOR\COUNT.C 88 7: printf( "\tConsonants: %6ld\n", Letters
<F1=Help> <Alt=Menu> <F6=Window> R MP N 00001.02B

```

For example, if the declaration you want is the one that declares `CountWords`, you can jump directly to that location.

► **To jump directly to a match:**

1. Put the cursor on the match.
2. From the Search menu, choose Set Match.

PWB opens the correct file if it is not already open and positions the cursor on the text you located.

You can use multifile searching regardless of whether the files that you want to search are open in PWB. For example, you can search `$INCLUDE:*.H` (all the headers on the INCLUDE path) for a particular prototype.

Using Regular Expressions

The PWB searching capabilities that you have used so far are useful when you know the exact text you are looking for. Sometimes, however, you have only part of the information that you want to match (for example, the beginning or end of the string), or you want to find a wider range of information. In such cases, you can use regular expressions.

Regular expressions are a notation for specifying patterns of text, as opposed to exact strings of characters. The notation uses literal characters and metacharacters. Every character that does not have special meaning in the regular-expression syntax is a literal character and matches an occurrence of that character. For example,

letters and numbers are literal characters. A metacharacter is an operator or delimiter in the regular-expression syntax. For example, the backslash (\) and the asterisk (*) are metacharacters.

PWB supports two syntaxes for regular expressions: UNIX and non-UNIX. Each syntax has its own set of metacharacters. The UNIX metacharacters are `.\[\]**+^$`. The non-UNIX metacharacters are `?\[]**+^$@#(\){}`. Because it uses fewer metacharacters, the UNIX form is a little more verbose. However, it is more familiar to programmers who have experience with UNIX tools such as **awk** and **grep**. This book uses the UNIX syntax, but any expression that can be written with this syntax can also be written with the non-UNIX syntax.

The regular-expression syntax used by PWB depends on the setting of the **Unixre** switch (UNIX is the default). You can change the **Unixre** switch by using the Editor Settings dialog box.

Note PWB switches that take regular expressions always use UNIX syntax. They are independent from the **Unixre** switch.

Finding Text

In the multifile searching example, you learned how to locate every occurrence of **int** in the COUNT project. In a large project, finding every **int** would yield too many matches. To narrow the search, you can use a regular expression.

For this example, you want to match declarations of functions returning **int**. You can specify this with a regular expression. This expression matches text that:

- Begins at the start of the line
- Followed by the keyword **int**
- Followed by white space
- Followed by an identifier
- Followed by any text within parentheses

The syntax for this regular expression is shown in Figure 5.1.

```
^int\b[a-zA-Z0-9_]+(.*)
```

Figure 5.1 Regular Expression Example

It illustrates the following important features of regular expressions:

1. Regular expressions can contain literal text. In this example, `int` is literal text and is matched exactly.
2. Regular expressions can contain predefined regular expressions. Here, `\:b` is shorthand for a pattern that matches one or more spaces or tabs (that is, white space). For a complete list of predefined regular expressions, see Appendix A.
3. You can use *classes* of characters in regular expressions. A class matches any one character in the class. For example, the class `[a-zA-Z0-9_]` is the class of characters that contains all lowercase and uppercase letters and all digits plus the underscore. The dash (`-`) defines a range of characters in a class.
4. The plus sign (`+`) after the class instructs PWB to look for one or more occurrences of any of the characters in the class. This is the key to regular expressions. You don't have to know exactly what appears between `int` and the left parenthesis; all you have to do is describe what *can* be there.

The pattern `^int\:b[a-zA-Z0-9_]+(.*)` matches strings such as

```
int CountWords( void )
int    2BadCIdentifiers()
```

but not the strings

```
int ( char *t )
integer(val)
```

Figure 5.2 shows a more detailed way to write an expression that matches the declaration of a function returning an `int`.

`^ *int\:b[a-zA-Z_][a-zA-Z_0-9$]+ *(.*)`

The diagram shows the regular expression `^ *int\:b[a-zA-Z_][a-zA-Z_0-9$]+ *(.*)` with ten numbered vertical lines pointing to specific parts of the expression:

- 1: `^` (caret)
- 2: `*` (space)
- 3: `int` (literal text)
- 4: `\` (backslash)
- 5: `b` (backspace)
- 6: `[a-zA-Z_][a-zA-Z_0-9$]+` (character class)
- 7: `*` (space)
- 8: `(` (left parenthesis)
- 9: `.` (dot)
- 10: `)` (right parenthesis)

Figure 5.2 Complex Regular Expression Example

This expression is close to the C-language definition for the syntax of the declaration. It is more precise than most searches require, but it is useful as an illustration of how to write a complex regular expression.

You can interpret this expression as follows:

1. Start at beginning of line, which is specified by a caret (`^`) at the beginning of the regular expression.

2. Skip leading optional spaces. To specify optional items, this expression matches *zero or more* occurrences by using the asterisk (*) operator. The expression “*” means “match zero or more spaces.”
3. Look for the **int** keyword as literal text.
4. Skip white space. There must be at least one space or tab.
5. Look for exactly one alphabetic character or underscore.
6. Look for any characters that are alphabetic, numeric, an underscore (_), or a dollar sign (\$). This and the previous part of the expression guarantee that the identifier conforms to the Microsoft C definition of an identifier.
7. Skip optional spaces.
8. Look for a left parenthesis.
9. Skip zero or more of any character.
10. Look for a right parenthesis.

TIP To match a C identifier, use the `\i` predefined expression.

This expression is exact to the point that it takes longer to write than the time it saves. The key to using regular expressions effectively is determining the minimal characteristics that make the text qualify as a match. For example, it's probably not necessary that the text between the space and the left parenthesis be a valid C identifier to qualify as a match. Any sequence of alphanumeric characters or underscores is usually sufficient.

► **To find all function declarations that return an int:**

1. From the Search menu, choose Find.
2. In the Find Text box, type `^int\b\:\i(`.
3. Select the Regular Expression check box.
4. Choose the Files button.
5. Add the pattern `CO*.C` and the file `COUNT.H` to the file list.
6. Choose OK to start the search.

When the search is complete, choose View Results. You can see in the Search Results window that PWB matched only the function declarations.

Replacing Text

You can use regular expressions when changing text to achieve some extremely powerful results. A regular expression replacement can be a simple one-to-one replacement, or it can use “tagged” expressions. A tagged expression marks part of the matched text so that you can copy it into the replacement text.

For example, you can manipulate lists of files easily using regular expressions. This exercise shows how to get a clean list of files that is stripped of the size and time-stamp information.

► **To get a clean list of C files in the current directory:**

TIP A simpler way to get a list of files is to type **Arg wildcard Openfile** (ALT+A wildcard F10).

1. From the File menu, choose New.

This gives you a new file for the directory listing.

2. Execute the function sequence **Arg Arg !dir *.c Paste**.

The default key sequence for this command is to press ALT+A twice, type !dir *.c, then press SHIFT+INS.

Arg Arg introduces a text argument to the **Paste** function with an **Arg** count of two. The exclamation point (!) designates the text argument to be run as an operating-system command. Without the exclamation point, the text is the name of a file to be merged. If only one **Arg** is used, PWB inserts the text argument.

PWB runs the **DIR** command and captures the output. When the **DIR** command is complete, PWB prompts you to press a key. When you press a key, PWB then inserts the results of the command at the cursor. For more information about this and other forms of the **Paste** function, see “Paste” in Chapter 7, “Programmer’s WorkBench Reference.”

3. From the Search menu, choose Replace.
4. In the Find Text box, type \:b\:z \:z-.*\$

This pattern means:

- White space followed by
- A number followed by
- Exactly one space followed by
- A number followed by
- A dash (–) followed by
- Any sequence of characters, then
- End of the line

This string must be tied to the end of the line to prevent the search from finding anything too close to the beginning of the line.

5. Make sure there are no characters in the Replace Text text box.
6. Choose Replace All.

PWB prompts you to verify that you want to replace text with an empty string.

7. Choose OK to confirm that you want to perform the empty replacement.

TIP Any time you need a quick reference to regular expressions, type ALT+A regex F1.

All the file-size, date, and time-stamp information is removed. Because you did not reuse any of the original text in the replacement, this is a simple regular expression replacement.

Choose Close from the File menu to discard the text you created in the previous exercise.

A more complicated task is backing up the C files to a directory called LAST, which is assumed to be a subdirectory of the current directory. A batch file makes this easier. You can create such a batch file using regular expressions.

► **To create a batch file that copies the C files to a subdirectory:**

1. Create a list of C files in the current directory as described in the previous example, but do not remove the file sizes, dates, and times.
2. Delete the heading printed by the **DIR** command.
3. From the Search menu, choose Replace.
4. In the Find Text text box, type:

```
^\([^\ ]+\)[ ]+\([^\ ]+\).*
```

5. This expression finds a string that starts at the beginning of the line (^). Placing parts of the expression inside the delimiters \ (and \) is called “tagging.”

The first tagged expression (\([^\]+\)) matches one or more characters that are not spaces. A leading caret in a class means “not.”

The pattern then matches one or more spaces ([]+), followed by the second tagged expression which matches one or more characters that are not spaces.

The remainder of the line is matched by the wildcard (.), which matches any character, and the repeat operator (*). Matching the rest of the line is important because that is how this pattern removes everything after the filename. It discards these portions of the matched text.

6. In the Replace Text text box, type

```
COPY \1.\2 .\LAST
```

7. Select Replace All and click OK to begin the find-and-replace operation.

PWB transforms each directory entry into a command to copy the file to the LAST subdirectory.

TIP To type a literal tab character in a dialog box, use the Quote function by pressing CTRL+P TAB.


```

File Edit Search Project Run Options Browse Window Help
#=[ 2] Untitled.002
copy COUNTBUF.C .\LAST
copy COUNTCH.C .\LAST
copy COUNT.C .\LAST
copy ANNUITY1.C .\LAST
4 file(s) 6437 bytes
20635648 bytes free

4 occurrences replaced
OK

F1=Help Enter Esc=Cancel Tab=Next Field MP N 00001.001

```

The word `COPY` is inserted literally. The text matched in the first tagged expression (the base name) replaces the expression `\1`. The period is inserted literally. The text matched by the second tagged expression (the filename extension) replaces the expression `\2`. The space is inserted literally. The text `.\LAST` is inserted as `.\LAST`. Be sure to use two backslashes to indicate a literal backslash; otherwise, PWB expects a reference to a tagged expression such as `\1` and displays an error message.

You'll notice that the last two lines of the file are not useful in your batch file. They are the remnants of the summary statistics produced by the **DIR** command. Delete these two lines and you have a finished batch file.

Using the Source Browser

Another search technique is "browsing." Browsing uses information generated by the compiler to help you find pieces of code quickly. This section introduces you to some of the capabilities of the Source Browser. The browser is a handy tool for moving about in projects, large and small.

In addition to navigating through your program, you can use the browser to explore the relationships between parts of the project. The browser database contains full information about where each symbol is defined and used and about the relationships among modules, constants, macros, variables, functions, and classes. Note that the browser files can be very large.

Note This section uses the COUNT project you created in Chapter 3. If you did not create this project or if you have since deleted it, you must create it now. For instructions on how to create the COUNT project, see “Creating the Project,” on page 42.

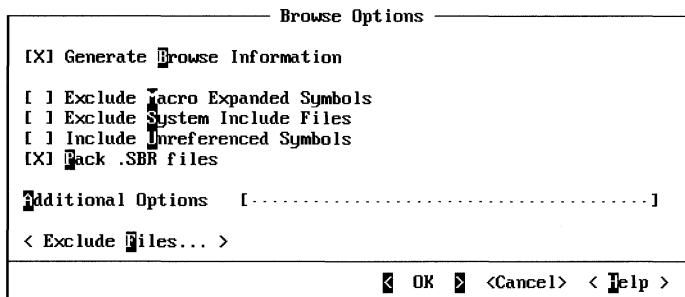
Creating a Browser Database

Before you can use the PWB Source Browser, you must build a browser database. PWB helps you maintain this database automatically as a part of a normal project build.

► To build a browser database:

1. Open the COUNT project using the Open Project command from the Project menu.
2. From the Options menu, choose Browse Options.

PWB displays the Browse Options dialog box.



3. Turn on the Generate Browse Information check box.
4. Choose OK.

The browser changes the project makefile to build the project. It adds compiler options for creating browser information (.SBR files). It includes a BSCMAKE command which combines the .SBR files and creates a browser database (a .BSC file).

5. From the Project menu, choose Rebuild All.

Rebuilding the entire project ensures that the database contains up-to-date information for all files in your program.

When the build completes, the following new files are on your disk:

- COUNT.BSC, the browser database
- COUNTBUF.SBR, a zero-length “placeholder” for COUNTBUF.
- COUNTCH.SBR, a placeholder for COUNTCH.
- COUNT.SBR, a placeholder for COUNT.

After adding each .SBR file’s contribution to the database, BSCMAKE truncates it and leaves the empty .SBR file on disk to provide an up-to-date target for later builds. Leaving these files on the disk ensures that a browser database is not rebuilt unless it is out-of-date with respect to its source files.

A PWB project is not required to create a browser database (although it is convenient). For information on how to build a browser database for non-PWB projects, see “Building Databases for Non-PWB Projects” on page 104.

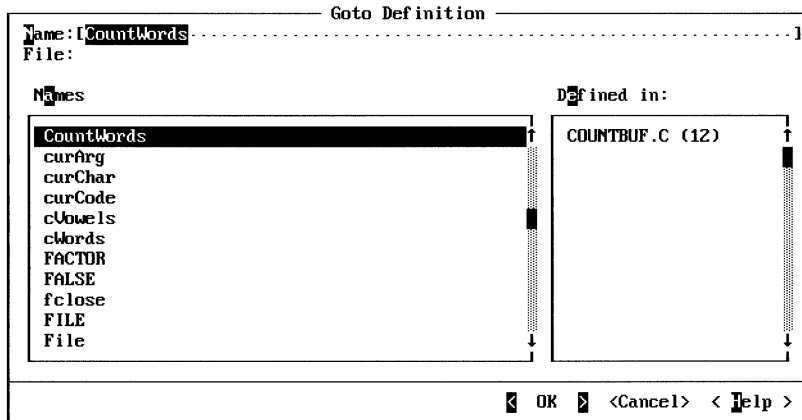
Finding Symbol Definitions

When you are working on a program, it’s easy to forget where a particular variable, constant, or function is defined. You can use the Find command to locate occurrences of a symbol, but that offers little information about which one is the definition. To make such searches easier, you can choose Goto Definition from the Browse menu to jump directly to the definition of any symbol in your program.

The following procedure uses the COUNT project to demonstrate how powerful the browser can be.

► To go to the definition of CountWords:

1. From the Window menu, choose Close All.
2. Open COUNT.C.
3. Move the cursor to the `CountWords` call on line 80.
4. From the Browse menu, choose Goto Definition.
PWB displays the Goto Definition dialog box.



Notice that `CountWords` is highlighted and the defining file's name is displayed in the list box to the right. More than one defining file is listed if a name is defined in several scopes.

5. Choose OK.

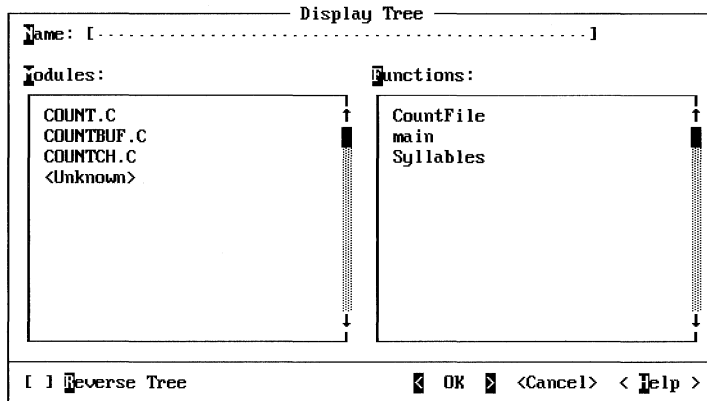
PWB opens `COUNTBUF.C` and shows the definition of `CountWords`.

Showing the Call Tree

Often when analyzing an existing program's flow, or when looking for opportunities for optimization, it's useful to refer to a "call tree." A call tree is a view of your program that provides, for each function, a list of other functions called.

► To generate a call tree of COUNT:

1. From the Browse menu, choose Call Tree.
PWB displays the Display Tree dialog box.



2. Choose COUNT.C from the Modules list box.
Notice that the Functions list box changes to show only the functions in COUNT.C.
3. Choose OK to see the call tree.

The call tree for COUNT.C is as follows:

```

CountFile
+-fopen?
+-printf[13]?
+-fread?
+-CountWords
| +-Analyze[2]
| | +-strchr[3]?
+-fclose?
main
+-CountFile[2]...
+-printf?
+-gets?
Syllables

```

TIP To jump directly to the definition of a name, place the cursor on the name and choose Goto Definition from the Browse menu.

Three kinds of annotations appear in the call tree:

?

A symbol followed by a question mark is used by your program but not defined in any of the program files in the browse database. These are often library functions.

[*n*]

The number *n* between square brackets shows symbols that are used more than once. In the preceding example, CountFile is shown as:

```
CountFile[2]
```

This means that there are two references to `CountFile` in `main`.

... (ellipsis)

The ellipsis means that the full information for the function appears elsewhere in the call tree.

Finding Unreferenced Symbols

As you write your program, you will occasionally remove function calls or references to global variables. This can leave unused code in your program or make the program's data larger than it needs to be. The browser database contains information about where every function and variable is referenced, so you can easily find the ones that are not used.

The `COUNT` project that you have been working with contains an unused function and an unreferenced global variable. This section shows how to use the Source Browser to find and remove the extra code and data.

The system include files define many more functions than many programs use. Therefore, unreferenced functions in your program are easiest to find when using a browser database that does not contain the system include files. This example begins by building a browser database for `COUNT` that does not contain information defined by system include files.

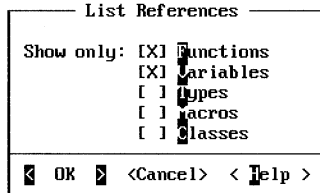
► To build the `COUNT` browser database:

1. Make sure that debug options are turned on. Debug options select the fast compiler and do not generate intrinsic functions. If you perform a release build which generates intrinsics, you will find many unused intrinsic functions defined by the compiler. For information on how to select debug options, see "Setting Build Options" on page 19.
2. From the Options menu, choose Browse Options.
PWB displays the Browse Options dialog box.
3. In the Browse Options dialog box, turn on the Exclude System Include Files and the Include Unreferenced Symbols check boxes.
4. Choose OK.

Now that the browse options are set, rebuild the project and browser database by choosing Rebuild All from the Project menu. With the updated browser database, you can obtain a list of references for functions and variables.

► **To get a list of references for function and variables:**

1. From the Browse menu, choose List References.
PWB displays the List References dialog box.



2. Turn on the Functions and Variables options, and then choose OK.

PWB opens the Browser Output window and creates the list of references. Each name is followed by a colon and a list of functions that refer to the name.

FUNCTION	CALLED BY LIST	
-----	-----	-----
Analyze:	CountWords[2]	
CountFile:	main[2]	
CountWords:	CountFile	
fclose:	CountFile	
fopen:	CountFile	
fread:	CountFile	
gets:	main	
main:		
printf:	main	CountFile[13]
strchr:	Analyze[3]	
Syllables:		

► **To find an unreferenced symbol:**

- Search for the regular expression `:$` (colon, dollar sign).

This pattern specifies a colon at the end of the line. It finds names that are followed by an empty list of references.

PWB positions you at the first unreferenced name (`main`) in the Browser Output window. The function `main` must be kept in the program, so you want the next unreferenced name.

To find all unreferenced items with one search, you can perform a logged search and add only `<browse>` (the Browser Output pseudofile) to the file list. This is especially useful for large projects. Because there are only two unused symbols in the COUNT project, it is simpler to repeat the search.

► **To find the next unreferenced symbol:**

- Execute the **Psearch** function (press F3) to repeat the regular-expression search. PWB finds the `Syllables` function.

► **To go to the definition of `Syllables` in the source:**

1. From the Browse menu, choose Goto Definition.
Because the cursor is on `Syllables` in the Browser Output window, PWB automatically selects the definition.
2. Choose OK.

PWB jumps to the definition of `Syllables` in `COUNT.C` where you can remove the unused function. Now you can remove the unused variable by following the same steps.

► **To find the unused variable:**

1. Return to the Browser Output window.
2. Press F3 to find the next unreferenced variable `Consonants`.
3. Choose Goto Definition, and then choose OK.

PWB jumps to the definition of `Consonants`.

You can delete the line to remove the definition of the extra variable. The only remaining cleanup is to remove the declarations for `Syllables` and `Consonants` from the `COUNT.H` file.

Advanced Browser Database Information

In the previous sections, you learned the basics of building a browser database and some useful applications of the Source Browser. In this section, you will find information on what goes into a browser database and how to estimate the disk requirements to build one. You will also learn how to build a database for non-PWB projects and how to build a single database for related projects.

Estimating .SBR and .BSC File Size When you build a browser database, you first create an `.SBR` file for each source file in the project. Each of these files contains the following information:

- The name of the source file and the files it includes.
- Every symbol defined in the source file and the files it includes.

These symbols are the names of all functions, types (including the names of all classes, structures, and enumerations and their members), macros (including

symbols in the expanded macro), and variables in the file. These symbols also include all parameters and local variables for the functions.

For C++, the names are the decorated names (names with encoded type information), which can take up about half of the .SBR file size. For more information on decorated names, see Appendix B.

- The location of all symbol definitions in the files.
- The location of all references to every symbol in the files.
- Linkage information.

This is a tremendous amount of information about your program and can therefore occupy a large quantity of disk space. The benefit is that the Source Browser provides fast, sophisticated access to this database of knowledge about your program.

For C source files, the .SBR file is typically half the size of the preprocessed source file (that is, the source file with comments removed, all files included, and all macros expanded).

For C++, the expansion of the .SBR file is from approximately 2 to 20 times the size of the source file. This dramatic expansion occurs because:

- More information is defined in C++ include files than in C include files.
- The database contains decorated symbol names.

Intuitively, you might assume that the resulting browser database (.BSC file) is approximately the sum of all the .SBR files. However, the browser database is the *union* of the information in the component .SBR files. This means that the .BSC file is not extremely large. Much of the information in the .SBR files is defined in include files, which are common to many modules in the project. The union of the .SBR files is relatively small because most of the include-file information is duplicated in each .SBR file.

A 400K .BSC file is common for a modestly sized program. At the time this book was written, the largest known browser database was about four megabytes.

Building Databases for Non-PWB Projects The simplest way to build a browser database for non-PWB projects is to build the browser database separately from the project. You can use a makefile or a batch file for this purpose. The process requires only two steps:

1. Create an .SBR file for each module. The simplest way to do this is to run the compiler with the options to produce an .SBR file and no other files. For example, the CL command line:

```
CL /Zs /W0 /Fr *.c
```

specifies that the compiler processes all .C files in the current directory, checks syntax only (/Zs) and issues no warnings (/W0). Therefore, no object files are produced. However, browser information (.SBR files) are generated (/Fr).

2. Combine the .SBR files into a browser database.

The syntax for this command is:

BSCMAKE options /oproject.BSC *.sbr

For complete information on BSCMAKE options and syntax, see Chapter 21.

The process of creating a browser database changes little between projects. Therefore, you could use a batch file for many projects similar to the following example:

```
ECHO OFF
REM Require at least one command-line option
IF %1.==. GOTO USAGE

REM Compile to generate only .SBR files
CL /Zs /W0 /Fr *.c

REM Build the browser database
BSCMAKE %2 %3 %4 %5 %6 %7 %8 /o%1.BSC *.sbr
GOTO END

:USAGE
REM Print instructions
ECHO -Usage: %0 project [option]...
ECHO -   project      Base name of browser database
ECHO -   [option]...  List of BSCMAKE options
:END
```

This batch file assumes that all the project sources are in the current directory. It requires that you specify the name of the browser database and allows BSCMAKE options. You may want to change this file to specify different BSCMAKE or compiler options.

If your project's sources are distributed across several directories, you must write a custom batch file or makefile to build the database. For more information on the BSCMAKE utility, see Chapter 21.

► To use a custom browser database in PWB:

1. From the Browse menu, choose Open Custom.
2. Choose the Use Custom Database button.
3. Select your custom browser database and choose OK.

If you want to save this database name permanently, choose Save Current Database.

4. Choose OK.

The PWB Source Browser opens your custom database.

You can now browse your non-PWB project.

If you are using a makefile to build your project, you can choose **Open Project** from the **Project** menu and open it as a non-PWB project makefile. If the project makefile has the same base name as the browser database and resides in the same directory, PWB automatically opens the database when you open the project. For more information on using a non-PWB makefile for a project in PWB, see “Using a Non-PWB Makefile” on page 61.

Building Combined Databases If you have two or more closely related projects, you can combine the browser databases for the projects. For example, if two large programs differ only in one or two modules so that most of the sources are shared between the two projects, it can be useful to browse both projects with a single browser database.

► **To build a combined browser database:**

1. Generate the .SBR files for both projects.
2. Pass all of the .SBR files to BSCMAKE to build the combined database.

The resulting database is the inclusive-OR of the information in the two projects.

5.2 Executing Functions and Macros

The menus and dialog boxes in PWB provide access to almost everything you need to do to develop your projects. You can edit, search, and browse your source files. You can build, run, and debug your project, and you can view Help for the entire system. However, the visible display provides access to only part of the capabilities available in PWB. Behind the menu commands lie functions with many more options than you can access from the menus. Many functions and macros are not assigned to keys by default.

The sophisticated PWB user learns how to use the functions and predefined macros to perform the precisely correct action. Each function has several forms that are invoked with the combinations of the **Arg** and **Meta** prefixes. These two functions are used to introduce arguments and modify the action of PWB functions.

Arg (ALT+A)

The fundamental function in PWB. You use **Arg** to begin selecting text, introduce text and numeric function arguments, or modify the action of functions by increasing the Arg count.

TIP Lasttext (CTRL+O) recovers the previous text argument and displays it in the Text Argument dialog box.

To pass a text argument to a function, for example, press ALT+A, and then type the text. The text you type doesn't go into your file. The Text Argument dialog box appears when you type the first letter of the text.

Arg[11]	Text Argument
[Ambidextrous.....]	
<Cancel> < Help >	

You can then edit the text. PWB displays the current argument count and Meta state in the dialog box.

TIP A selection is also a text argument.

Notice that there is no OK button in this dialog box. Instead of choosing OK, press the key for the function you want to execute with this argument. Choose the Cancel button if you do not want to execute a function.

Meta (F9)

Modifies the action of a function in different ways from the various argument types. It generally toggles an aspect of the function's action.

For example, the text-deletion functions usually move the deleted text to the clipboard. However, when modified with **Meta**, they clear the text without changing the clipboard.

The combination of **Arg** and **Meta** greatly increases the number of variations available to each function. For example, the **Psearch** function can perform different search operations depending on how it is executed. **Psearch** can:

- Repeat the previous search (**Psearch**).
- Search for text (**Arg text Psearch**).
- Perform a case-sensitive text search (**Arg Meta text Psearch**).
- Search for a regular expression (**Arg Arg text Psearch**).
- Search for a case-sensitive regular expression (**Arg Arg Meta text Psearch**).

Because you can reassign keys to your preference, the PWB documentation cannot assume that a specific key executes a given function or macro. Therefore, the PWB documentation gives a sequence of functions or macros by name, followed by the same sequence of actions by key name. In this book, the key is the default key. In PWB Help, the displayed key is the one currently assigned to that function. When no key is assigned, PWB displays `unassigned`.

For example, to insert the definition of a macro at the cursor, you pass the name of the macro to the **Tell** function and modify **Tell**'s action with the **Meta** prefix. This sequence of actions is expressed as follows:

- Execute the function sequence **Arg Meta macroname Tell**
(ALT+A F9 *macroname* CTRL+T).

If the **Tell** function is assigned to a different key, Help displays that key in place of CTRL+T.

Chapter 7, “Programmer’s WorkBench Reference,” contains complete descriptions of all forms of each function in PWB.

Executing Functions and Macros by Name

The most frequently used functions and macros are assigned to certain keys by default. For example, the **Paste** function is assigned to SHIFT+ENTER, **Linsert** is assigned to CTRL+N, and so on. Sometimes, however, you want to use a function or macro that is not assigned to a key. You can always assign a key by using the Key Assignments command or by using the **Assign** function. However, that is a lot of trouble for something you need only once. PWB allows you to execute a function or macro by name, rather than by pressing a key.

► To execute a function or macro by name:

- Perform the function sequence **Arg function Execute**
(ALT+A *function* F7).

In other words, press ALT+A (execute the **Arg** function), type the name of the function or macro, and then press F7 (invoke the **Execute** function).

The argument to **Execute** doesn’t have to be a single function or macro name. It can be a list of functions and macros. The argument is really a temporary, nameless macro. This means that you can do anything in an argument to **Execute** that you can do in a macro. PWB follows the rules for macro syntax and execution. You can define labels, test function results, and loop.

Warning When executed from a macro, PWB functions that display a yes-or-no prompt assume a “Yes” response. To restore the prompt, use the macro prompt directive (<). For more information, see “Macro Prompt Directives” in PWB Help.

5.3 Writing PWB Macros

The Programmer's WorkBench, like other editors designed for programmers, provides a macro language so that you can customize and extend the editor or automate common tasks. You can create macros in one of the following ways:

- By recording actions you perform. The recording mechanism allows you to perform a procedure once, while PWB is recording. After you've recorded it, you can execute the macro to repeat the recorded procedure.
- By manually writing macros. This technique is less automatic but does allow you to write more powerful macros.

These two techniques are not mutually exclusive. You can start by recording a macro that approaches the steps you want to perform, then edit it to expand its functionality or handle different situations.

When Is a Macro Useful?

Macros are useful for automating procedures you perform frequently. You may also write macros that automate tedious one-time tasks.

Of course, not every task is a good candidate for automation. It might take longer to write the macro than to do the task by hand. If you don't expect to perform a task often, don't automate it. Also, automated editing procedures introduce an element of risk. You might not foresee situations that your macro can encounter. Incorrect macros can sometimes be destructive.

A little experience with macros and some careful testing will enable you to create a good set of macros for your own use.

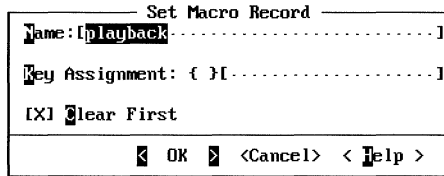
Recording Macros

Recording actions you perform with the mouse or at the keyboard can be a powerful way to write a macro. You turn on recording and perform the actions that you want the macro to execute. You can concentrate on the task that you want to automate, instead of concentrating on the syntax of the macro language.

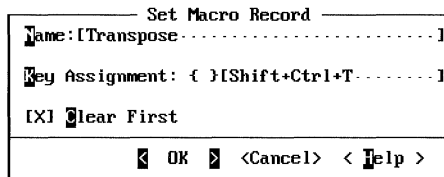
For example, if you occasionally reverse characters when you type quickly, a macro to switch them back is useful. Before recording a macro to transpose characters, you should think about what you are going to do while recording the macro. To transpose characters, you will select the character at the cursor, cut it onto the clipboard, move over one character, and then paste the character you cut.

► **To record a macro that transposes characters:**

1. From the Edit menu, choose Set Record.
PWB brings up the Set Macro Record dialog box.



2. In the Name text box, type Transpose.
3. Click the mouse in the key box (between the braces { }), or press TAB until the cursor is in the key box.
4. Press CTRL+SHIFT+T (for transpose).
PWB automatically fills in the name of the key you pressed.



5. Press TAB to leave the key box, and then choose OK.
PWB closes the Set Macro Record dialog box. When you turn on macro recording, PWB records a macro called Transpose and associates it with SHIFT+CTRL+T.

Important The Set Macro Record command does *not* start the macro recorder. It only specifies the name and key association for the macro you are going to record.

6. From the Edit menu, choose Record On.
When you choose Record On, the macro recorder starts. To indicate that the macro recorder is running, PWB displays the letter X on the status bar. Notice that the Project, Options, and Help menus are unavailable while PWB is recording a macro.
7. Select the character at the cursor by holding down the SHIFT key and pressing the RIGHT ARROW key.
8. Press SHIFT+DEL to cut the character onto the clipboard.

9. Press the RIGHT ARROW key to move the cursor to the new location for the character.
10. Press SHIFT+INS to paste the character from the clipboard back into the text.
11. From the Edit menu, choose Record On to stop the macro recorder.

Press SHIFT+CTRL+T to switch the character at the cursor with the character to the right. You can now use the new macro and key assignment for the rest of the PWB session.

► **To edit the macro:**

- From the Window menu, choose Record from the PWB Windows cascaded menu.

PWB opens the Record window.



The Record window shows the definition of the `Transpose` macro that you just recorded. You can edit the definition to change the way the macro works. For example, you decide that the macro should reverse the character at the cursor with the character to the left, instead of the character to the right.

► **To redefine the macro:**

1. Change the macro to read as follows:

```
Transpose:=select left delete left paste
```

2. Move the cursor to the macro definition.

TIP You can find a complete list of PWB functions in Chapter 7 and in PWB Help.

3. Press ALT+=, the default key for the **Assign** function.
Assigning the macro replaces the previous definition of `Transpose` with the new definition.
4. Return to the file you were originally viewing.

Up to this point, the macro exists only in memory. To use your recorded macro for subsequent PWB sessions, you must save the definition of the macro to disk.

► **To save the macro:**

1. If the Record window is not open, choose Record from the PWB Windows cascaded menu.
PWB opens the Record window.
2. From the File menu, choose Save.
PWB inserts the macro definition and the key assignment into your `TOOLS.INI` file for future sessions. When you leave PWB, you are prompted to save `TOOLS.INI`. Your changes are not permanent until you actually save `TOOLS.INI`.

Flow Control Statements

Recorded macros have the inherent limitation of playing back one fixed sequence of commands. Often you need a macro to execute repeatedly until some condition is satisfied. This requires that you use flow control statements to govern the actions your macro takes.

All editor functions return a true or false value. The macro flow control operators that use these values are:

Operator	Meaning
<code>+>label</code>	Branch to <i>label</i> if last function yields TRUE
<code>->label</code>	Branch to <i>label</i> if last function yields FALSE
<code>=>label</code>	Branch unconditionally to <i>label</i>
<code>:>label</code>	Define <i>label</i>

These rudimentary operators are not as sophisticated as a high-level language's IF statement or FOR loop. They are more like an assembly language's conditional jump instruction. However, they provide the essential capabilities needed for writing loops and other conditional constructs.

Flow Control Example

If you frequently perform multiple-window editing, a macro that restores the display to a single window can be helpful. Such a macro requires the following logic:

1. Switch to the next window.
2. If the switch is not successful (meaning that only one window is present), end the macro.
3. If the switch is successful (another window is present), close that window and go back to step one.

This macro will be called `CloseWindows` and assigned to `SHIFT+CTRL+W`.

► To create the `CloseWindows` macro:

1. From the File menu, choose All Files.

PWB displays the All Files dialog box.

Notice that your `TOOLS.INI` file is in the list of open files, even though you did not explicitly open it. PWB opens `TOOLS.INI` to load its configuration information (unless when you specify `/DT` on the PWB command line).

2. Select `TOOLS.INI` file in the list of open files.
3. Choose OK.

PWB opens a window and displays your `TOOLS.INI` file.

4. Find the section of `TOOLS.INI` that begins with `[pwb]`. This is the section where PWB keeps its startup configuration information.
5. In the PWB section, type the following two new lines:

```
CloseWindows:= :>Loop Openfile -> Meta Window Window =>Loop
CloseWindows: SHIFT+CTRL+W
```

If you want these definitions to take effect immediately, select both lines and press `ALT+=` to execute the Assign function. You can also assign the definitions one at a time.

6. Choose Save from the File menu to make this macro and key assignment part of your `TOOLS.INI` file.

The next time you start PWB, the `CloseWindows` macro is defined and assigned to the `SHIFT+CTRL+W` key.

The first line you typed uses the `:=` operator to associate the macro definition with the name “`CloseWindows`.” After the operator is the list of functions and macro operators that specify what the macro is to do. The second line is a separate statement that uses the `:` operator to assign the macro to the `SHIFT+CTRL+W` key.

The CloseWindows macro works as follows:

1. `:>Loop` defines a label called `Loop`. There cannot be a space between the `:>` operator and the label name.
2. `Openfile` switches to the window under the active window.
3. The `->` operator examines the return value from the **Openfile** function. If the function returns false because there is no other window, the `->` operator exits the macro.
4. The phrase `Meta Window` closes the active window.
5. `Window` returns to the window you started from.
6. `=>Loop` unconditionally transfers control back to the `Loop` label and starts the sequence again.

When this macro is defined, you can press `SHIFT+CTRL+W` whenever you want to close all windows except the active window.

User Input Statements

PWB macros can prompt for input. This helps you write more general macros. For example, you might keep a history of the changes you make to a file at the top in a format similar to the following:

```
/** Revision History **  
//15-Nov-1991:IAD:Add return value for DoPrint  
//31-Oct-1991:IAD:Implement printing primitives
```

To facilitate entering the revision history in reverse chronological order and to make it easy to keep track of where you were in the source file, you can write a macro to perform the following steps:

1. Set a mark at the cursor for future reference.
2. Insert a revision history header at the beginning of the file if one is not present.
3. Insert the current date.
4. Prompt for initials and insert them just below the header.
5. Prompt for comments and insert them after the initials.
6. Return to the saved position in the file.

Note that while this macro is executing, you can choose the Cancel button in the dialog boxes that prompt for initials and comments. The macro must handle these cases and gracefully back out of the changes to the file.

► **To enter this macro in TOOLS.INI:**

1. Open TOOLS.INI for editing.
2. Type the following macros and key assignment in the [pwb] section of TOOLS.INI:

```

LineComment:="// "
RevHead:= "*** Revision History ***"
RevComment:= \
    Arg Arg "Start" Mark \
    Begfile Arg RevHead Psearch +>Found \
    Linsert LineComment RevHead \
:>Found \
    Down Linsert Begline LineComment Curdate " (" \
    Arg "Initials" Prompt ->Quit Paste Endline ") " \
    Arg "Comment" Prompt ->Quit Paste =>End \
:>Quit Meta Ldelete \
:>End Arg "Start" Mark
RevComment:Ctrl+H

```

There are at least two spaces before the backslash at the end of each line. The backslashes are line-continuation characters. They allow you to write a macro that is more than one line long. In this case, line continuations format the macro in a readable way. To further assist in readability, you can indent the parts of the macro which define the actual keystrokes, as in the preceding example.

3. Choose Save from the File menu to save your changes.
4. To reinitialize PWB, execute the **Initialize** function by pressing SHIFT+F8.

PWB discards all of its current settings and rereads the PWB section of TOOLS.INI. The same effect can be achieved by quitting and restarting PWB.

The following discussion analyzes the workings of the definitions you added to TOOLS.INI. It repeats one or two lines from the text you typed and describes how each line works. You may want to refer to the full definition as you follow along.

The first two lines

```

LineComment:="//"
RevHead:= "*** Revision History ***"

```

define two utility macros that are used by the main RevComment macro. They define strings that are used several times in RevComment.

The third line

```

RevComment:= \

```

declares the name of the macro. The succeeding lines define the action of the RevComment macro.

The first line of the definition

```
Arg Arg "Start" Mark \
```

sets a mark named “Start” at the cursor so that the macro can restore the cursor position after inserting the comments at the beginning of the file.

The next line

```
Begfile Arg RevHead Psearch +>Found \
```

moves to the beginning of the file (**Begfile**), then searches forward for the revision-history header. If the header is found, PWB branches to the `Found` label; otherwise, it executes the next line.

```
Linsert LineComment RevHead \
```

If the macro is here, the header was not located in the file. The **Linsert** function creates a new line, and PWB types the revision-history header. The macro continues with the line:

```
:>Found \
```

This line defines the `Found` label. At this point in the macro, the cursor is on the line with the header. The next lines insert the new revision information, starting with the following line:

```
Down Linsert Begline LineComment Curdate " (" \
```

PWB moves the cursor down one line (**Down**), inserts a new line (**Linsert**), moves to the beginning of the line (**Begline**), and calls the `LineComment` macro to designate the line as a comment. PWB then types the current date (**Curdate**) and an open parenthesis.

The macro prompts for initials:

```
Arg "Initials" Prompt ->Quit Paste Endline ") " \
```

The macro uses the **Prompt** function to get your initials. If you choose the Cancel button, the function returns false, so the macro branches to the label `Quit`. If you choose the OK button, the text you typed in the dialog box is passed to the **Paste** function, which inserts the text. The macro moves the cursor to the end of the line (**Endline**) and types a closing parenthesis.

The code on this line explicitly handles the case when you cancel the prompt (the false condition). The phrase `->Quit` causes PWB to skip to the label `Quit` when **Prompt** returns false.

If you use the **Prompt** function and you do not handle the false condition, a null argument (a text string with zero length) is passed to the next function. Therefore, a phrase like `Arg "Que?" Prompt Paste` pastes either the input or nothing, depending on whether you choose the OK or Cancel button. Passing a null argument to **Paste** is harmless, but some functions require an argument. In these cases, you can use the `->` operator to terminate the macro.

The `RevComment` macro uses an explicit label so that it can end the macro without an error when you choose the Cancel button. The next line of the macro is almost the same as the previous line in the macro.

```
Arg "Comment" Prompt ->Quit Paste =>End \
```

On this line, if the paste is carried out, an unconditional branch is taken to the label `End` and passes over the `Quit` branch, which is defined on the next line.

```
:>Quit Meta Ldelete \
```

The `Quit` branch is taken when you cancel a prompt. The macro has to clean up the text already inserted by the macro. The **Meta Ldelete** function deletes the incomplete line that would have been the revision-history entry. The next line defines the last step of the macro.

```
:>End Arg "Start" Mark
```

The `End` label defines the entry point for the common cleanup code. This line restores the cursor to the initial position when you invoked the macro. Because this line does not end in a line-continuation character (`\`), it is the end of the `RevComment` macro.

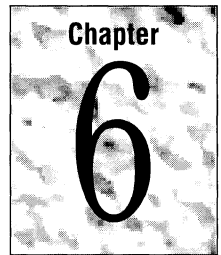
The last line that you typed is not part of the `RevComment` macro. It is a separate `TOOLS.INI` entry.

```
RevComment:Ctrl+H
```

This line assigns the `CTRL+H` key to the `RevComment` macro.

You can polish this macro by adding `Arg "Start" Meta Mark` to the end of the macro. This phrase deletes the mark. A better alternative is to use the **Savecur** and **Restcur** functions instead of named marks. However, this example uses named marks to illustrate how to use them in a macro.

Customizing PWB



PWB is a completely customizable development environment. You can modify PWB in the following ways:

- Changing mapping of keystrokes to actions.
- Changing default behavior of PWB (for example, how tabs are handled or if PWB automatically saves files).
- Changing the colors of parts of the PWB display.
- Adding new commands to the Run menu.
- Programming new editor actions (macros).

You can find instructions on how to write macros in “Writing PWB Macros” on page 109.

In addition to the customizations that you can make by using the commands in the Options menu, you can also customize PWB by editing the TOOLS.INI file.

Note Another category of customization that is not covered in this book is how to write PWB extensions. An extension is a dynamically loaded module that can access PWB’s internal functions. Extensions can do much more than macros. To learn more about writing PWB extensions, see the Microsoft Advisor Help system (choose “PWB Extensions” from the main Help table of contents).

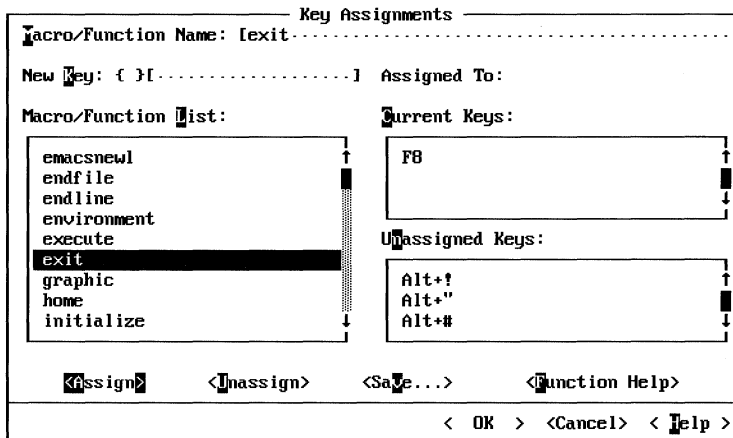
6.1 Changing Key Assignments

PWB maps actions (functions and macros) to keys. You can assign any of these actions to keys other than the default keys.

For example, **Exit** is a PWB function. Its default key assignment is F8. A BRIEF user may prefer to use ALT+X to leave the editor.

► **To make ALT+X execute the Exit function:**

1. From the Options menu, choose Key Assignments.
PWB displays the Key Assignments dialog box.



2. Select Exit in the Macro/Function List box, or type `exit` in the Macro/Function Name text box.
3. Move the cursor to the New Key box between the braces (`{}`) by clicking between the braces or by pressing ALT+K.
4. Press ALT+X.

PWB types `ALT+X` in the text box after the braces and displays the name of the macro or function that `ALT+X` is currently assigned to. With the default settings, you can see that `ALT+X` is assigned to the **Unassigned** function. Pressing a key in the key box is a quick way to find out the name of the function assigned to the key.

Note When the cursor is in the key box (between the braces), most keys lose their usual meaning, including ESC, F1, and the dialog box access keys. The key you press is interpreted as the key to be assigned. Only TAB, SHIFT+TAB, ENTER, and NUMENTER retain their usual meaning. To assign one of these keys, type the name of the key in the text box.

5. Press TAB to move the cursor out of the key box.
6. Choose Assign.

PWB assigns Exit to the `ALT+X` key. Note that Exit is still assigned to the F8 key. Functions can be assigned to many keys.

7. Choose OK.

Important To change a key, you *must* choose the Assign button. The OK button dismisses only the dialog box. It does not perform any other action. This design allows you to assign many keys in one session with the dialog box.

The change remains in effect for the duration of the session.

► **To make a permanent key assignment:**

1. From the Options menu, choose Key Assignments.
2. Choose Save.

PWB displays the Save Key Assignments dialog box, which lists all of the unsaved assignments that you have made during the PWB session by using the Key Assignments dialog box.

3. Delete any settings that you do not want to save.
4. Choose OK.

PWB writes your new settings into the [PWB] section of TOOLS.INI for subsequent sessions. When you exit PWB, you are prompted to save TOOLS.INI. Your changes are not permanent until you actually save the file to disk.

TIP Key assignments can be temporary.

If you already know the function name, you can make a quick assignment for the current session by using the **Assign** function instead of going through the Key Assignments dialog box.

► **To assign a key using the Assign function:**

- Execute the function sequence:

Arg *function:key* **Assign** (ALT+A *function:key* ALT+=).

For example, to assign **Exit** to ALT+X:

1. Press ALT+A to execute **Arg**.
2. Type `exit:ALT+X`
3. Press ALT+= to execute **Assign**.

The assignment is in effect for the rest of the PWB session.

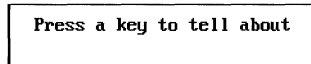
The key assignments you make by using the **Assign** function are not listed in the Save Key Assignments dialog box.

To discover the name of the function or macro that is currently assigned to a key, use the Key Assignments dialog box (as previously described) or use the **Tell** function.

► **To find a current key assignment using Tell:**

1. Press CTRL+T to execute **Tell**.

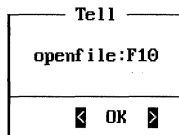
PWB displays the prompt:



Press a key to tell about

2. Press the key you want to find out about.

If you press F10, PWB displays the function assigned to the F10 key (Openfile).



Tell

open file:F10

◀ OK ▶

The **Tell** function has many other uses in addition to displaying key assignments. For more information on **Tell**, see page 216.

6.2 Changing Settings

When you first use PWB, you don't have to specify the tab stops, whether the editor starts in insert or overtype mode, and so on. These settings (called "switches") are all covered by defaults. PWB's default behavior can be extensively customized by changing the values of PWB switches.

TIP To see a list of PWB switches in Help, type ALT+A switches F1.

Switches fall into three categories:

- Boolean switches. True/false or on/off switches that can also be specified as yes/no or 0/1. An example of a Boolean switch is **Autosave**, which governs whether PWB saves a file when you switch to a different one.
- Numeric switches. An example of a numeric switch is **Undocount**, which determines the maximum number of editing actions you can undo.
- Text switches. Examples of a text switch are **Markfile**, the name of the file in which to store marks, **Tabstops**, a list of tab-stop intervals, and **Readonly**, the operating-system command for PWB to run when saving a read-only file.

► **To change the setting for Tabstops:**

1. From the Options menu, choose Editor Settings.

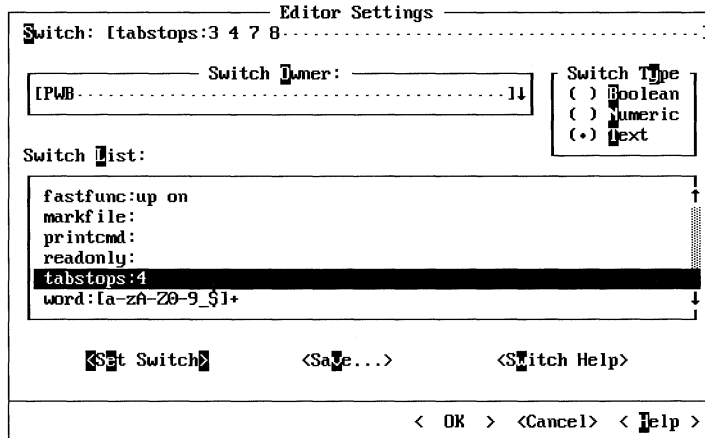
PWB displays the Editor Settings dialog box.

2. **Tabstops** is a text switch (not a numeric switch as you might expect), so select the Text option button.
3. Select **Tabstops** in the Switch List box.
PWB shows the current setting for **Tabstops** in the Switch text box at the top of the dialog box.
4. Move to the Switch text box by clicking in the box or by pressing ALT+S.
PWB selects only the switch value, instead of the entire text.
5. Type the new setting:

3 4 7 8

This setting defines a tab stop at columns 4, 8, 15, and every eight columns thereafter. At this point, the Editor Settings dialog box should look like:

TIP Choose Switch Help to determine what a switch does and the syntax to specify its value.



6. Choose the Set Switch button to change the setting of the **Tabstops** switch.
7. Choose OK.

Important To change a setting you *must* choose the Set Switch button. The OK button only dismisses the dialog box. It does not perform any other action. This design allows you to set many switches in one session with the dialog box.

The new tab stops you set are used for the current session. If you want to use this setting permanently, you must choose the Save button in the Editor Settings dialog box. This changes your TOOLS.INI file in the same way as for key assignments.

TIP You can set temporary switch settings.

You can make temporary switch assignments for the current session by using the **Assign** function. You do this in the same way as for a key assignment by typing **Arg switch:value Assign** (ALT+A *switch:value* ALT+=).

You may be curious about the Switch Owner box that you did not use in this example. The Switch Owner is either PWB or a PWB extension such as PWBHELP (the extension that provides the Microsoft Advisor in PWB). Type or select a switch owner to set switches for that extension. Each extension has its own section in TOOLS.INI.

Note When you choose Set Switch, most switch settings take effect immediately. However, changes to the **Height** switch do not take effect until you choose OK.

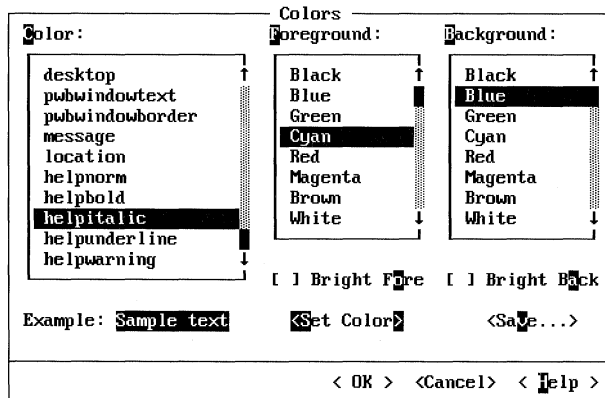
6.3 Customizing Colors

You can change the color of almost any item in the PWB interface. For a table showing the names and meanings of PWB's color settings, see the "Programmer's WorkBench Reference" on page 271.

Some displays show a brilliant green for the left and right triangular symbols surrounding buttons in Help.

► To change the light green to light cyan:

1. From the Options menu, choose Colors.
PWB displays the Colors dialog box.



2. Select **Helpitalic** in the Color list box.
3. Select Cyan in the Foreground list box.
4. Choose Set Color.

To verify your change, press F1. The green symbols in help are now light cyan. While you are viewing Help, you can find out what parts of PWB the rest of the color names determine. To leave Help, click the Cancel button or press ESC. PWB returns you to the Colors dialog box.

The Bright Fore and Bright Back check boxes determine if the given color is the usual version of the color or the bright version of the color. Bright black, for example, is usually a dark gray color.

If you want to save your new colors for subsequent sessions, choose the Save button. PWB displays the Save Colors dialog box where you can delete modifications that you don't want to save. When you choose OK in the Save Colors dialog box, PWB modifies TOOLS.INI to record your changes.

6.4 Adding Commands to the Run Menu

You can add up to six commands to the Run menu to integrate your own utilities into PWB. A command is the name of any executable (.EXE or .COM) file, batch (.BAT) file, or built-in operating-system command such as DIR or COPY.

Suppose you use an outline processor to keep project notes. You can start the outline processor from PWB's Run menu.

► To add a command to the Run menu:

1. From the Run menu, choose Customize Run Menu.
2. Choose the Add button.

PWB displays the Add Custom Run Menu Item dialog box for you to describe your custom menu item:

Add Custom Run Menu Item		
Menu Text:	[Project ~Notes.....]	
Path Name:	[.....]	
Arguments:	[.....]	
Output File Name:	[.....]	
Initial Directory:	[.....]	
Help Line:	[.....]	
<input type="checkbox"/>	Use Dialog Box for Arguments and Output File	
<input type="checkbox"/>	Prompt Before Returning	<input type="checkbox"/> Execute in Background
Shortcut Key:	(.) None	() Alt+ <input type="checkbox"/> [...]
<input type="button" value="OK"/> <input type="button" value="Cancel"/> <input type="button" value="Help >"/>		

3. Type `Project ~Notes...` in the Menu Text box.

The tilde (~) before the letter N indicates the highlighted access letter for the menu command. The ellipsis (...) uses the standard convention to indicate that the command will require more information before it is completed. An ellipsis is commonly associated with a dialog box command but can be used in this context as well.

4. Specify the full path to the outlining program, `OUTLINE.EXE`, in the Path Name text box. (The program name `OUTLINE.EXE` is for example purposes only. Substitute the name of your own outliner or other program in its place.)
5. Specify the arguments you want to pass to the outliner in the Arguments text box: `%|dpfF.log`.

This example illustrates a powerful feature of PWB: its ability to extract parts of the filename to form a new name for customized menu items. The specification `%|dpfF` extracts the drive (d), path (p), and base name (f) of the current file. Anything after `F` is added to the end of the name.

For example, if the current file is `C:\SOURCE\COUNT.C`, the argument that PWB passes to the program is `C:\SOURCE\COUNT.LOG`.

6. In the Help Line text box, type the explanatory message that appears on the status bar when you browse this menu item:

Run the OUTLINE program

7. Choose OK to confirm your entries.

PWB adds the command to your Run menu and modifies `TOOLS.INI` to save the new item. You can now access your outline processor directly from the Run menu.



Note You can add other text processing or word processing programs to the Run menu. If you change the current file using another program, PWB prompts you to update the file or to ignore the changes made by the other program.

6.5 How PWB Handles Tabs

The following functions and switches control how PWB handles tabs:

Name	Type	Description
Realtabs	Switch	Determines if PWB preserves tabs on modified lines
Entab	Switch	The white space translation method
Tabalign	Switch	The alignment of the cursor within a tab field
Filetab	Switch	The width of a tab field
Tabdisp	Switch	The fill-character for displaying tab fields
Tab	Function	Moves the cursor to the next tab stop
Backtab	Function	Moves the cursor to the previous tab stop
Tabstops	Switch	Tab positions for Tab and Backtab

For detailed information on each function and switch, see Help or the “Programmer’s WorkBench Reference.” For instructions on how to set a switch see “Changing Settings” on page 122. For instructions on how to assign a function to a key, see “Changing Key Assignments” on page 119.

To understand how PWB handles tabs, you need to know only a few facts:

- The **Tab** (TAB) and **Backtab** (SHIFT+TAB) cursor-movement functions and the **Tabstops** switch have nothing to do with tab characters. They affect cursor movement, rather than the handling of tab characters, and are not discussed further here. For more information on these items, see the “Programmer’s WorkBench Reference.”
- PWB never changes any line in your file unless you explicitly modify it (lines longer than PWB’s limit of 250 characters are the exception).

Some text editors translate white space (that is, entab or detab) when they read and write the file. PWB does not translate white space when it reads or writes a file. This is to be compatible with source-code control systems that would detect the translated lines as changed lines.

- PWB translates white space according to the **Entab** switch only when you modify a line.
- **Tabalign** has an effect only when **Realtabs** is set to yes.
- A “tab break” occurs every **Filetab** columns.
- When PWB displays a tab in the file, it fills from the tab character to the next tab break with the **Tabdisp** character.

Figure 6.1 illustrates how PWB displays tabs.

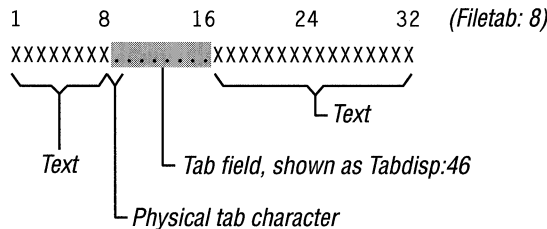


Figure 6.1 How PWB Displays Tabs

- When translating white space, PWB preserves the exact spacing of text as it is displayed on screen.

To set the width of displayed tabs, change the setting of the **Filetab** switch.

To tell PWB to translate white space on lines that you modify, set the **Realtabs** switch to `no` and the **Entab** switch to a nonzero value, according to the translation method that you want to use. The **Entab** switch takes one of the following values:

Entab	Translation Method
0	Translate white space to space characters
1	Translate white space outside of quotation-mark pairs to tabs
2	Translate white space to tabs

To preserve white space exactly as you type it, set the **Realtabs** switch to yes and the **Entab** switch to 0.

When **Realtabs** is yes, the **Tabalign** switch comes into effect. When **Tabalign** is set to yes, PWB automatically repositions the cursor onto the physical tab character in the file, similar to the way a word processor positions the cursor. When **Tabalign** is set to no, PWB allows the cursor to be anywhere in the tab field.

If you want the TAB key to type a tab character, assign the TAB key to the **Graphic** function. Note that when a dialog box is displayed, the TAB key always moves to the next option. You can always use the following method to type a tab character, whether you are in a dialog box or an editing window.

► **To type a literal tab character in your text or in a dialog box:**

1. Execute the **Quote** function (press CTRL+P).
2. Press TAB.

Examples

The following example sets up tabs so that they act the same as in other Microsoft editors, such as QuickC or Word:

```
realtabs:yes
tabalign:yes
graphic:tab
trailspace:yes
entab:0
```

The **Trailspace** switch is needed so that the TAB key will have an effect on otherwise blank lines.

To save your file so that it does not include any actual tab characters (ASCII 9), use the following settings:

```
realtabs:no
entab:0
tabstops:3
```

The **Tabstops** value determines the number of spaces inserted for each press of the tab key.

Another example of a common tab configuration is one in which the TAB key inserts a tab in insert mode but moves over text to the next tab stop when the editor is in overtype mode.

First, use the following tab settings:

```
realtabs:yes
tabalign:yes
```

Then insert the following macro into the PWB section of your TOOLS.INI:

```
;Insert mode and overtype mode tabbing
TabI0:= Insertmode +>over Insertmode "\t" => \
      :>over Insertmode Tab
TabI0:TAB
```

For more information on PWB macros see “Writing PWB Macros” on page 109.

6.6 PWB Configuration

PWB keeps track of three kinds of information between sessions in these three files:

File	Information Saved
TOOLS.INI	Configuration and customizations, such as key assignments, colors, and macro definitions
CURRENT.STS	The editing environment used most recently
<i>project</i> .STS	The editing and building environment for a project

TOOLS.INI is described in the next section: “The TOOLS.INI File” on page 131. For more information about CURRENT.STS, see “Current Status File CURRENT.STS” on page 138, and for more information about the *project*.STS files, see “Project Status Files” on page 138.

When you start PWB, it reads the TOOLS.INI file, loads PWB extensions, and reads the CURRENT.STS or project status file in the following order:

1. PWB reads the [PWB] section of TOOLS.INI (except when PWB is started using the /D or /DT command-line options). For more information on tagged sections, see “TOOLS.INI Section Tags” on page 132.

If the [PWB] section contains **Load** switches, PWB loads the specified extension when each switch is encountered. When PWB loads an extension, it also reads the extension’s tagged section of TOOLS.INI, if any. For example, when the Help extension is loaded, PWB reads the [PWB-PWBHELP] section of TOOLS.INI.

2. PWB autoloads extensions (except when the /D or /DA option is used to start PWB).
The automatic loading of PWB extensions is described in the next section, “Autoloading Extensions.”
3. PWB reads the TOOLS.INI operating-system tagged section (except when /D or /DT is used).
4. PWB reads the CURRENT.STS status file (except when /D or /DS is used to start PWB).
5. PWB reads the TOOLS.INI tagged section for the file extension of the current file (except when /D or /DT is used to start PWB).
6. PWB runs the **Autostart** macro if it is defined in TOOLS.INI (except when /D or /DT is used).

Autoloading Extensions

PWB automatically loads extensions if they follow a specific naming convention and reside in a certain directory. For extensions that follow the convention, it is not necessary to put load statements in TOOLS.INI.

PWB searches the directory where the PWB executable file is located for file-names with the following pattern:

```
PWB*.MXT
```

PWB loads as many extensions with names of this form as it finds. When PWB loads an extension, it also loads the extension’s tagged section of TOOLS.INI.

To suppress extension autoloading, use the /DA option on the PWB command line.

Important Do not rename editor extensions. PWB and some extensions may assume the predefined filename.

The TOOLS.INI File

PWB, like other Microsoft tools, stores information in a file called TOOLS.INI. This file retains information about how you want PWB to work under various circumstances. PWB expects to find this file in the directory specified by your INIT environment variable.

TOOLS.INI is a text file. You can edit it using PWB or any other text editor. PWB also can store information directly to TOOLS.INI when, for example, you choose the Save Colors button in the Colors dialog box. PWB modifies this file when you save a recorded macro, a changed switch, a new key assignment, a custom browser database, or a custom project template.

TOOLS.INI Section Tags

The TOOLS.INI file is divided into sections, separated by “tags.” These tags are specified in the form:

[*tagname*]

The *tagname* is the base name of an executable file, such as NMAKE, CVW, or PWB. The tag defines the start of a TOOLS.INI section that contains settings for the indicated tool.

PWB extends this simple syntax to enable you to take different action depending on the operating system or the current file’s extension. The extended syntax is:

[**PWB**-*modifier*]

The modifier can be the base name of a PWB extension, an operating system’s identifier, or a filename extension for files that you edit.

Operating-System Tags

The following table lists the operating-system tags for various operating environments. If you are running Windows, use the tag for the version of DOS that you are running.

Tag	Operating Environment
[PWB-4.0]	MS-DOS versions 4.0 and 4.01
[PWB-5.0]	MS-DOS version 5.0

Be sure to use the correct version number for your operating system.

Filename-Extension Tags

The operating-system tags are read only once at startup. PWB reads the filename-extension tagged sections each time you switch to a file with that extension. For example, suppose that you want the tab stops for C and C++ files to be every four columns, and every eight columns for text files.

► To set tab options based on filename extension:

1. Open your TOOLS.INI file in an editing window.
2. Create a C and C++ section by typing the tag:

```
[PWB-.C PWB-.H PWB-.CPP PWB-.HPP]
```

TIP To open TOOLS.INI, choose it in the All Files dialog box. TOOLS.INI is always open.

3. Create a text file section by typing the tag:

```
[PWB-.TXT]
```

4. Put the appropriate **Tabstops**, **Entab**, and **Realtabs** switches in each section. The lines that begin with a semicolon are comments.

```
[PWB-.C PWB-.H PWB-.CPP PWB-.HPP]
; Set the tab stops for C and C++ to 4
tabstops : 4
; Translate white space to tabs
entab    : 1
realtabs : no
```

```
[PWB-.TXT]
; Set the tab stops for text files to 8
tabstops : 8
; Translate white space to spaces
entab    : 0
realtabs : no
```

Depending on whether the current file is a C (.C or .H) file or a text (.TXT) file, the tab stops are set at 4 or 8 columns, respectively.

PWB reads multiple sections and applies the appropriate settings. You can use this to your advantage by storing all your general settings in the [PWB] section and storing differences in separate tagged sections.

TIP The default extension tag is [PWB-..].

Filename-extension tagged sections are useful for the kinds of files you edit most frequently. However, it's impossible to define settings for every conceivable extension. To handle this case, PWB provides a special extension (..) that means "all extensions not defined elsewhere in TOOLS.INI."

For example, to set tab stops to 8 for all files except C and C++ files, modify the preceding example to use the [PWB-..] tag in place of [PWB-.TXT].

Note When you choose the Save button in the Key Assignments, Editor Settings, and Colors dialog boxes, and when you save a recorded macro or custom Run menu command, PWB saves the setting in the main section. If the setting is for a PWB extension, it is saved in that extension's tagged section. PWB never modifies or writes settings in a filename-extension or operating-system section.

Named Tags

You can define tagged sections of TOOLS.INI that you load manually. Use manually loaded sections to make special key assignments, to load complex or rarely used macros, or to use a special PWB configuration under a particular circumstance.

The syntax for a manually-loaded section tag is:

[**PWB-name**]

Where *name* is the name of the tagged section. A single section of TOOLS.INI can be given several tag names. These tags have the form:

[**PWB-name1 PWB-name2...**]

When you want to use the settings defined in one of these named sections, pass the name of the section to the **Initialize** function (SHIFT+F8).

► **To read a tagged section of TOOLS.INI:**

- Execute **Arg name Initialize** (ALT+A *name* SHIFT+F8)

You can use this method to read any tagged section, including the automatically loaded sections.

Note When you execute Initialize with no arguments, PWB clears all the current settings before reading the [PWB] section, including settings that you have made for specific PWB extensions. PWB does not reread the operating-system or other additional sections of TOOLS.INI. To reread the main section without clearing other settings that you want to remain in effect, label the main PWB section with the tag [PWB PWB-main]. You can then use **Arg main Initialize** to recover your startup settings, instead of using **Initialize** with no arguments.

TOOLS.INI Statement Syntax

Within each TOOLS.INI section you place a series of comments or statements. Each statement is a macro definition, key assignment, or switch setting, and must be stated on a single logical line. Statements can be continued across lines by using line-continuations.

General Macro Syntax

The general syntax for a macro definition is:

name := *definition*

PWB does not reserve any names. Therefore, be careful not to redefine a PWB function. For more information about how to write macros, see “Writing PWB Macros” on page 109.

General Key Syntax

The general syntax for a key assignment is:

name : *key*

The *name* is the name of a function or macro, and the *key* is the name of a key. To see how to write a given key, use the **Tell** function as described in “Changing Key Assignments” on page 119.

Note that certain keys have fixed meanings when the cursor is in a dialog box or in the Help window. You can assign one of these keys to a function or macro, but the fixed meaning is used in a dialog box or the Help window.

The following keys have fixed meanings:

Key	Dialog Box	Help Window
ESC	Choose Cancel	Close the Help window
F1	See Help on the dialog box (choose Help)	See Help on the current item
TAB	Move to the next option	Move to the next hyperlink
SHIFT+TAB	Move to the previous option	Move to the previous hyperlink
SPACEBAR	Toggle the setting of the current option	Activate the current hyperlink
ENTER, SHIFT+ENTER, NUMENTER, SHIFT+NUMENTER	Choose the default action	Activate the current hyperlink

Note The Windows operating environment or a terminate-and-stay-resident (TSR) program may override PWB’s use of specific keys. PWB has no knowledge of keys that are reserved by these external processes. PWB lists these keys as available keys in the Key Assignments dialog box and allows you to assign functions to these keys, but you may not be able to use them. See the documentation for your operating environment to see what keys are reserved by the system.

General Switch Syntax

The general syntax for a switch setting is:

switch : *value*

The exact syntax for the switch value depends on the switch. See Chapter 7, “PWB Reference,” for more information about each switch.

Line Continuation

All statements in TOOLS.INI must be stated on a single logical line. A logical line can be written on several physical lines by using the TOOLS.INI line-continuation character, the backslash (\).

The backslash must be preceded by a space to be treated as a line-continuation character. Precede the backslash by two spaces if you want the concatenated statement to contain a space at that location. If the backslash is preceded by a tab, PWB treats the tab as if it were two spaces. The backslash should be the last character on the line except for spaces or tabs.

The backslash in the following statement is *not* a line continuation.

```
Qreplace:CTRL+\
```

However, the backslash at the end of the first line below *is* a line continuation.

```
findtag:=Arg Arg "^\\[[^\\]]+\\\" Psearch ->nf \
  Arg Setwindow => :>nf Arg "no tag" Message
```

In this example, the backslash is preceded by two spaces. The first space is included to separate `->nf` from `Arg` in the concatenated macro definition. The second space identifies the backslash that follows it as the line-continuation character.

Comments

In the TOOLS.INI file, PWB treats the text that follows a semicolon (;) up to the end of the line as a comment. To specify the beginning of a comment, you must place the semicolon at the beginning of a line or following white space.

For example, the first semicolon in the following statement is part of a command, and the second semicolon begins a comment.

```
Printcmd:lister -t4 %s -c; ;Print using lister program
```

In the following example, the first semicolon is a key name, and the second semicolon begins a comment.

```
Sinsert:CTRL+; ;Stream insertion: CTRL plus semicolon
```

Semicolons inside a quoted string do not begin a comment.

Environment Variables

The INIT environment variable tells PWB where to find the TOOLS.INI file and where to store the CURRENT.STS file. The proper setting of these variables—INIT, TMP, LIB, INCLUDE, HELPFILES, and PATH—governs whether your development environment works smoothly.

► **To set the INIT environment variable from the command line:**

- Type `SET INIT=C:\INIT`

The operating-system **SET** command sets the environment variable to contain the string `C:\INIT`. This example presumes that you want to store your initialization files in `C:\INIT`. You could use any other directory. Make sure that the INIT environment variable lists a single directory. Multiple directories in INIT can cause inconsistent behavior.

The following list outlines how the environment works:

- The environment is always inherited from the parent process. The parent is the process that starts the current process. In DOS, the parent is often `COMMAND.COM` or Windows.
- Inheritance of environment variables is a one-way process. A child inherits from its parent. You can make changes to the environment in a child (when you use the Environment Variables command in PWB, for example), but they are not passed back to the parent. This means that any changes to environment variables that you make while shelled out are lost when you return to PWB.
- Each DOS session under Windows inherits its environment from Windows. Changes made to the environment in one session do not affect any other session.

The best way to make sure your environment is set properly is to explicitly set it in one of your startup files. These are:

- `CONFIG.SYS`
- `AUTOEXEC.BAT`

PWB can save the complete table of environment variables for each project. You can then use the Environment Variables command from the Options menu to change environment variables for individual projects.

If you prefer that PWB save the environment variables for all PWB sessions or use the current operating-system environment when it starts up, change the **Envcursave** and **Envprojsave** switches. For more information on these switches, see the “Programmer’s WorkBench Reference” on pages 279 and 280.

Current Status File CURRENT.STS

The first time you run PWB or CodeView, it creates a CURRENT.STS (current status) file in your INIT directory. If there is no INIT directory, PWB and CodeView create the file in the current directory.

CURRENT.STS keeps track of the following items for PWB:

- Open windows, including their size and position and the list of open files in each window
- Screen height
- Window style
- Find string
- Replace string
- The options used in a find or find-and-replace operation, such as the use of regular expressions
- Optionally, all environment variables

PWB and CodeView share the current location and filename for the active window. When you leave CodeView after a debugging session and return to PWB, PWB positions the cursor at the place where you stopped debugging. For more information on the items that CodeView saves in CURRENT.STS, see “The CURRENT.STS State File” on page 343.

The next time you run PWB, it reads CURRENT.STS and restores the editing environment to what it was when you left PWB. For more information on how PWB uses environment variables, see “Environment Variables” on page 137.

The status files are plain text files. You can load one into an editor and read it. However, you might corrupt the file if you try to modify it. There is no need to modify it because PWB keeps it updated for you. No harm occurs if you delete CURRENT.STS. However, you will have to manually reopen the files you were working on.

Project Status Files

For each project, PWB creates a project status file. PWB stores this file in the project directory and gives it the name *project.STS*, where *project* is the base name of the project.

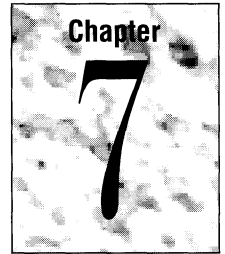
Project status files contain the same kind of information that CURRENT.STS contains, except on a per-project basis. This scheme allows PWB to keep track of your screen layout, file history, and environment variables for each project. The project status files also contain the current project template, language and utility options, build directory, and the program’s run-time arguments.

The main difference between the two status files is that the `CURRENT.STS` file supplies default status information—settings that PWB uses when you have not set a project. PWB uses the project's status file when you open that project.

By default, PWB saves a project's environment variables in the project status file. For more information on how PWB uses environment variables, see “Environment Variables” on page 137.

Important While it is harmless to delete `CURRENT.STS`, you should *not* delete project status files. They contain important information for building and updating your project. If you delete a project status file, you may need to delete the project makefile and start over.

Programmer's WorkBench Reference



7.1 PWB Command Line

Syntax **PWB** *[[options]]* *[[/t]] files*

Options Use the following case-insensitive options when starting PWB:

/D*[[S|T|A]]...*

Disables PWB loading the initialization files or PWB extensions as indicated by the following letters:

Letter	Meaning
---------------	----------------

S	Disable reading the status file CURRENT.STS
---	---

T	Disable reading TOOLS.INI
---	---------------------------

A	Disable PWB extension autoload
---	--------------------------------

The /D option alone disables loading all the PWB extension and initialization files. See: **Autoload**.

Note If you start PWB with the /DT option, this means that PWB options you change during the session cannot be saved.

/PP *makefile*

Opens the specified PWB project.

/PF *makefile*

Opens the specified non-PWB project (foreign makefile).

/PL

Resets the last project. Use this option to start PWB in the same state you last left it. You can set this option as the default by setting the **Lastproject** switch to yes.

/E *command*

Executes the given command or sequence of commands as a macro upon startup.

If *command* contains a space, *command* should be enclosed in double quotation marks (""). A single command need not be quoted. If *command* uses literal quotation marks, place a backslash (\) before each mark. To use a backslash, precede it with another backslash.

/R

PWB starts in no-edit mode. You cannot modify files in this mode. See: **Noedit**.

/M {*mark* | *line*}

PWB starts at the specified location. See: **Mark**.

[[/T] ,file]...

Tells PWB to load the given files on startup. If you specify a single file, PWB loads it. If you specify multiple files, PWB loads the first file; then when you use File Next or the **Exit** function, PWB loads the next file in the list.

If a /T precedes a filename or wildcard, PWB loads each file as a temporary file. PWB does not include temporary files in the list of files saved between sessions.

Note No other options can follow /T on the PWB command line. You must specify /T for each file you want to be temporary.

7.2 PWB Menus and Keys

Many PWB menu commands activate PWB functions or predefined macros. The menu commands that are attached to functions and macros are listed in the tables that follow. To assign a shortcut key for one of these menu commands, use the Key Assignments command on the Options menu and assign a key to the corresponding function or macro. For details on using the Key Assignments dialog box, see “Changing Key Assignments” on page 119.

Names beginning with an underscore (pwb...) are macros. Names without an underscore are functions.

Table 7.1 File Menu and Keys

Menu Command	Macro or Function	Default Keys
New	_pwbnewfile	Unassigned
Close	_pwbclosefile	Unassigned
Next	_pwbnextfile	Unassigned
Save	_pwbsavefile	SHIFT+F2
Save All	_pwbsaveall	Unassigned
DOS Shell	_pwbshell	Unassigned
<i>n file</i>	_pwbfile <i>n</i>	Unassigned
Exit	_pwbquit	ALT+F4

Table 7.2 Edit Menu and Keys

Menu Command	Macro or Function	Default Keys
Undo	_pwbundo	Unassigned
Redo	_pwbredo	Unassigned
Repeat	_pwbrepeat	Unassigned
Cut	Delete	SHIFT+DEL, SHIFT+NUM-
Copy	Copy	CTRL+INS, SHIFT+NUM*
Paste	Paste	SHIFT+INS, SHIFT+NUM+
Delete	_pwbclear	DEL
Set Anchor	Savecur	Unassigned
Select To Anchor	Selcur	Unassigned
Stream Mode	_pwbstreammode	Unassigned
Box Mode	_pwbboxmode	Unassigned
Line Mode	_pwblinemode	Unassigned
Record On	_pwbrecord	Unassigned

Table 7.3 Search Menu and Keys

Menu Command	Macro or Function	Default Keys
Log	_pwblogsearch	Unassigned
Next Match (Logging on)	_pwbnextlogmatch	SHIFT+CTRL+F3
Next Match (Logging off)	_pwbnextmatch	Unassigned
Previous Match (Logging on)	_pwbpreviouslogmatch	SHIFT+CTRL+F4
Previous Match (Logging off)	_pwbpreviousmatch	Unassigned
Goto Match	_pwbgotomatch	Unassigned

Table 7.4 Project Menu and Keys

Menu Command	Macro or Function	Default Keys
Compile File	_pwbcompile	Unassigned
Build	_pwbbuild	Unassigned
Rebuild All	_pwbrebuild	Unassigned
Close	_pwbcloseproject	Unassigned
Next Error	_pwbnextmsg	SHIFT+F3
Previous Error	_pwbprevmsg	SHIFT+F4
Goto Error	_pwbsetmsg	Unassigned

Table 7.5 Run Menu and Keys

Menu Command	Macro or Function	Default Keys
<i>command1</i>	_pwbuser1	[[ALT+Fn]]
<i>command2</i>	_pwbuser2	[[ALT+Fn]]
<i>command3</i>	_pwbuser3	[[ALT+Fn]]
<i>command4</i>	_pwbuser4	[[ALT+Fn]]
<i>command5</i>	_pwbuser5	[[ALT+Fn]]
<i>command6</i>	_pwbuser6	[[ALT+Fn]]
<i>command7</i>	_pwbuser7	[[ALT+Fn]]
<i>command8</i>	_pwbuser8	[[ALT+Fn]]
<i>command9</i>	_pwbuser9	[[ALT+Fn]]

Table 7.6 Browse Menu and Keys

Menu Command	Macro or Function	Default Keys
Goto Definition	Pwbrowsegotodef	Unassigned
Goto Reference	Pwbrowsegotoref	Unassigned
View Relationship	Pwbrowseviewrel	Unassigned
List References	Pwbrowselistref	Unassigned
Call Tree (Fwd/Rev)	Pwbrowsecalltree	Unassigned
Function Hierarchy	Pwbrowsefuhier	Unassigned
Module Outline	Pwbrowseoutline	Unassigned
Which Reference	Pwbrowsewhref	Unassigned
Class Tree (Fwd/Rev)	Pwbrowsecltree	Unassigned
Class Hierarchy	Pwbrowseclhier	Unassigned
Next	Pwbrowsenext	CTRL+NUM+
Previous	Pwbrowseprev	CTRL+NUM-

Table 7.7 Window Menu and Keys

Menu Command	Macro or Function	Default Keys
New	_pwbnewwindow	Unassigned
Close	_pwbclose	CTRL+F4
Close All	_pwbcloseall	Unassigned
Move	_pwbmove	CTRL+F7
Size	_pwbresize	CTRL+F8
Restore	_pwbrestore	CTRL+F5
Minimize	_pwbminimize	CTRL+F9
Maximize	_pwbmaximize	CTRL+F10
Cascade	_pwbcascade	F5
Tile	_pwbtile	SHIFT+F5
Arrange	_pwbarrange	ALT+F5
<i>n file</i>	_pwbwindow <i>n</i>	ALT+ <i>n</i>

Table 7.8 Help Menu and Keys

Menu Command	Macro or Function	Default Keys
Index	_pwbhelp_index	Unassigned
Contents	_pwbhelp_contents	SHIFT+F1
Topic	_pwbhelp_context	F1
Help on Help	_pwbhelp_general	Unassigned
Next	_pwbhelp_again	Unassigned
Search Results	_pwbhelp_searchres	Unassigned

7.3 PWB Default Key Assignments

PWB's default keys assignments are shown in table 7.9. In each position having the text `Unassigned`, you can assign a function or macro to that key without taking away a default keystroke. You cannot assign keys for positions that are empty. These can usually be expressed in a different way. For example, `CTRL+{` is expressed as `SHIFT+CTRL+{`.

Table 7.9 PWB Default Key Assignments

Key	Plain	SHIFT	ALT	CTRL	CTRL+SHIFT
!	Graphic	—	—	—	—
#	Graphic	—	—	—	—
\$	Graphic	—	—	—	—
%	Graphic	—	—	—	—
&	Graphic	—	—	—	—
(Graphic	—	—	—	—
*	Graphic	—	—	—	—
+	Graphic	—	—	—	—
,	Graphic	—	Unassigned	—	—
-	Graphic	—	Unassigned	Unassigned	—
.	Graphic	—	Unassigned	Unassigned	—
/	Graphic	—	Unassigned	Unassigned	—
0	Graphic	—	Unassigned	Unassigned	—
1	Graphic	—	_pwbwindow1	Unassigned	—
2	Graphic	—	_pwbwindow2	Unassigned	—
3	Graphic	—	_pwbwindow3	Unassigned	—
4	Graphic	—	_pwbwindow4	Unassigned	—
5	Graphic	—	_pwbwindow5	Unassigned	—

Table 7.9 (continued)

Key	Plain	SHIFT	ALT	CTRL	CTRL+SHIFT
6	Graphic	—	_pwbwindow6	Unassigned	—
7	Graphic	—	_pwbwindow7	Unassigned	—
8	Graphic	—	_pwbwindow8	Unassigned	—
9	Graphic	—	_pwbwindow9	Unassigned	—
:	Graphic	—	Unassigned	—	Unassigned
;	Graphic	—	Unassigned	Unassigned	—
<	Graphic	—	Unassigned	—	Unassigned
=	Graphic	—	Assign	Unassigned	—
>	Graphic	—	Unassigned	—	Unassigned
?	Graphic	—	Unassigned	—	Unassigned
@	Graphic	—	—	—	Unassigned
A	Graphic	Graphic	Arg	Mword	Unassigned
B	Graphic	Graphic	(Browse menu)	Unassigned	Unassigned
C	Graphic	Graphic	Unassigned	Ppage	Unassigned
D	Graphic	Graphic	Unassigned	Right	Unassigned
E	Graphic	Graphic	(Edit menu)	Up	Unassigned
F	Graphic	Graphic	(File menu)	Pword	Unassigned
G	Graphic	Graphic	Unassigned	Cdelete	Unassigned
H	Graphic	Graphic	(Help menu)	Unassigned	Unassigned
I	Graphic	Graphic	Unassigned	Unassigned	Unassigned
J	Graphic	Graphic	Unassigned	Sinsert	Unassigned
K	Graphic	Graphic	Unassigned	Unassigned	Unassigned
L	Graphic	Graphic	Unassigned	Replace	Unassigned
M	Graphic	Graphic	Unassigned	Mark	Unassigned
N	Graphic	Graphic	Unassigned	Linsert	Unassigned
O	Graphic	Graphic	(Options menu)	Lasttext	Unassigned
P	Graphic	Graphic	(Project menu)	Quote	Unassigned
Q	Graphic	Graphic	Unassigned	Unassigned	Unassigned
R	Graphic	Graphic	(Run menu)	Mpage	Record
S	Graphic	Graphic	(Search menu)	Left	Sethelp
T	Graphic	Graphic	Unassigned	Tell	Unassigned
U	Graphic	Graphic	Unassigned	Lastselect	Unassigned
V	Graphic	Graphic	Unassigned	Insertmode	Unassigned
W	Graphic	Graphic	(Window menu)	Mlines	Unassigned
X	Graphic	Graphic	Unassigned	Down	Unassigned
Y	Graphic	Graphic	Unassigned	Ldelete	Unassigned
Z	Graphic	Graphic	Unassigned	Plines	Unassigned

Table 7.9 (continued)

Key	Plain	SHIFT	ALT	CTRL	CTRL+SHIFT
[Graphic	—	Unassigned	Pbal	Unassigned
\	Graphic	—	Unassigned	Qreplace	Unassigned
]	Graphic	—	Unassigned	Setwindow	Unassigned
^	Graphic	—	—	—	Unassigned
_	Graphic	—	—	—	Unassigned
{	Graphic	—	Unassigned	—	—
	Graphic	—	Unassigned	—	—
}	Graphic	—	Unassigned	—	—
~	Graphic	—	Unassigned	—	Unassigned
F1	_pwbhelp- _context	_pwbhelp- _contents	_pwbhelp_back	Pwbhelpnext	Unassigned
F2	Setfile	_pwbsavefile	Unassigned	Unassigned	Unassigned
F3	Psearch	_pwbnextmsg	Unassigned	Compile	_pwbnext- logmatch
F4	Msearch	_pwbprevmsg	_pwbquit	_pwbclose	_pwbprevious- logmatch
F5	_pwbcascade	_pwbtile	_pwbarrange	_pwbrestore	Unassigned
F6	Selwindow	_pwb- prevwindow	Unassigned	Winstyle	Unassigned
F7	Execute	Refresh	Unassigned	_pwbmove	Unassigned
F8	Exit	Initialize	Unassigned	_pwbresize	Unassigned
F9	Meta	Shell	Unassigned	_pwbminimize	Unassigned
F10	Openfile	Unassigned	Unassigned	_pwbmaximize	Unassigned
F11	Unassigned	Unassigned	Unassigned	Unassigned	Unassigned
F12	Unassigned	Unassigned	Unassigned	Unassigned	Unassigned
F13	Unassigned	Unassigned	Unassigned	Unassigned	Unassigned
F14	Unassigned	Unassigned	Unassigned	Unassigned	Unassigned
F15	Unassigned	Unassigned	Unassigned	Unassigned	Unassigned
F16	Unassigned	Unassigned	Unassigned	Unassigned	Unassigned
LEFT	Left	Select	Unassigned	Mword	Select
RIGHT	Right	Select	Unassigned	Pword	Select
UP	Up	Select	Unassigned	Mlines	Unassigned
DOWN	Down	Select	Unassigned	Plines	Unassigned
INS	Insertmode	Paste	Unassigned	Copy	Unassigned
DEL	_pwbclear	Delete	Unassigned	Unassigned	Unassigned
HOME	Begline	Select	Unassigned	Begfile	Select
END	Endline	Select	Unassigned	Endfile	Select

Table 7.9 (continued)

Key	Plain	SHIFT	ALT	CTRL	CTRL+SHIFT
ENTER	Emacsnewl	Newline	Unassigned	Unassigned	Unassigned
BKSP	Emacscdel	Emacscdel	Undo	Unassigned	Undo
ESC	Cancel	Unassigned	Unassigned	Unassigned	Unassigned
GOTO	Home	Unassigned	Unassigned	Unassigned	Unassigned
NUM*	Graphic	Copy	Unassigned	Unassigned	Unassigned
NUM+	Graphic	Paste	Unassigned	Pwbrowseext	Unassigned
NUM-	Graphic	Delete	Unassigned	Pwbrowseprev	Unassigned
NUM/	Graphic	—	Unassigned	Unassigned	Unassigned
NUM- ENTER	Emacsnewl	Newline	Unassigned	Unassigned	Unassigned
PGUP	Mpage	Select	Unassigned	Unassigned	Select
PGDN	Ppage	Select	Unassigned	Unassigned	Select
TAB	Tab	Backtab	Unassigned	Unassigned	Unassigned

Note on Available Keys

PWB allows you to assign functions and macros to almost any key combination. However, some keys have a fixed meaning in certain circumstances or operating environments. PWB lists these key as available keys in the Key Assignments dialog box, and PWB allows you to assign a command to the key. However, when the circumstance holds, or you are running PWB in a specific environment, certain keys have a fixed meaning that overrides any assignment that you make.

Help Window

In the Help window, the following keys have a fixed meaning:

Key	Meaning
ESC	Close the Help window
TAB	Move to next hyperlink
SHIFT+TAB	Move to previous hyperlink
ENTER	Activate current hyperlink
NUMENTER	Activate current hyperlink
SHIFT+ENTER	Activate current hyperlink
SHIFT+NUMENTER	Activate current hyperlink
SPACE	Activate current hyperlink

Dialog Boxes

In dialog boxes, all keys have predetermined meanings. Your assignments have no effect when a dialog box is displayed. In particular, note the following keys:

Key	Meaning
ESC	Choose Cancel
ENTER	Choose the active command button
F1	Choose Help
TAB	Move to the next option or command
SHIFT+TAB	Move to the previous option or command
SPACE	Toggle active option
CTRL+P	When used in a text box, inserts the next key as a literal value. Use this key to type a literal tab character.

The Text Argument dialog box is an exception. All keys except ESC (Cancel) and F1 (Help) have their assigned meaning.

Microsoft Windows

When running PWB under Windows, some keys are reserved for use by Windows. You can override Windows' use of these keys by setting options in a PIF file.

Key	Default Meaning in Windows
ALT+ESC	Switch to the next window in Windows
CTRL+ESC	Switch to the Windows Task Manager
ALT+TAB	Switch to the next application
ALT+SPACE	Activate the current window's system menu
ALT+ENTER	Shift application between full screen and window

7.4 PWB Functions

PWB provides a rich variety of editing, searching, and project-management capabilities in the form of functions. Most of PWB's menus and dialogs call these functions (or macros that use these functions) to perform their actions. You can write your own macros that use these capabilities in ways that precisely suit your needs. You can also execute every function directly, either by pressing a key or by using the **Execute** function.

Table 7.10 summarizes PWB functions. Most functions can be executed in different ways to perform related actions. Complete details are given in the A-to-Z reference that follows the table.

Table 7.10 PWB Functions

Function	Description	Keys
Arg	Begin a function argument	ALT+A
Arrangewindow	Arrange windows or icons	Unassigned
Assign	Define a macro or assign a key	ALT+=
Backtab	Move to previous tab stop	SHIFT+TAB
Begfile	Move to beginning of file	CTRL+HOME
Begline	Move to beginning of line	HOME
Cancel	Cancel arguments or current operation	ESC
Cancelexport	Cancel background search	Unassigned
Cdelete	Delete character	CTRL+G
Clearmsg	Clear Build Results	Unassigned
Clearsearch	Clear Search Results	Unassigned
Closefile	Close current file	Unassigned
Compile	Compile and build	CTRL+F3
Copy	Copy selection to the clipboard	CTRL+INS, SHIFT+NUM*
Curdate	Today's date (<i>dd-Mmm-yyyy</i>)	Unassigned
Curday	Day of week (Tue)	Unassigned
Curtime	Current time (<i>hour:minute:second</i>)	Unassigned
Delete	Delete selection	SHIFT+DEL, SHIFT+NUM-
Down	Move down one line	CTRL+X, DOWN
Emascdel	Delete character	BKSP, SHIFT+BKSP
Emasnewl	Start a new line	ENTER, NUMENTER
Endfile	Move to end of file	CTRL+END
Endline	Move to end of line	END
Environment	Set or insert environment variable	Unassigned
Execute	Execute macros and functions by name	F7
Exit	Advance to next file or leave PWB	F8
Graphic	Type character	(many)
Home	Move to window corner	GOTO
Information	(Obsolete)	—
Initialize	Reinitialize	SHIFT+F8
Insert	Insert spaces or lines	Unassigned
Insertmode	Toggle insert/overtyping mode	CTRL+V, INS
Lastselect	Recover last selection	CTRL+U
Lasttext	Recover last text argument	CTRL+O

Table 7.10 (continued)

Function	Description	Keys
Ldelete	Delete lines	CTRL+Y
Left	Move left	CTRL+S, LEFT
Linsert	Insert lines or indent line	CTRL+N
Logsearch	Toggle search logging	Unassigned
Mark	Set, clear, or go to a mark or line number	CTRL+M
Maximize	Enlarge window to full size	Unassigned
Menukey	Activate menu	ALT
Message	Display a message or refresh the screen	Unassigned
Meta	Modify the action of a function	F9
Mgrep	Search across files for text or pattern	Unassigned
Minimize	Shrink window to an icon	Unassigned
Mlines	Scroll down by lines	CTRL+UP, CTRL+W
Movewindow	Move window	Unassigned
Mpage	Move up one page	CTRL+R, PGUP
Mpara	Move up one paragraph	Unassigned
Mreplace	Multifile replace with confirmation	Unassigned
Mreplaceall	Multifile replace	Unassigned
Msearch	Search backward for pattern or text	F4
Mword	Move back one word	CTRL+A, CTRL+LEFT
Newfile	Create a new pseudofile	Unassigned
Newline	Move to the next line	SHIFT+ENTER, SHIFT+NUMENTER
Nextmsg	Go to build message location	Unassigned
Nextsearch	Go to search match location	Unassigned
Noedit	Toggle the no-edit restriction	Unassigned
Openfile	Open a new file	F10
Paste	Insert file or text from clipboard	SHIFT+INS, SHIFT+NUM+
Pbal	Balance paired characters	CTRL+[
Plines	Scroll up by lines	CTRL+DOWN, CTRL+Z
Ppage	Move down one page	CTRL+C, PGDN
Ppara	Move down one paragraph	Unassigned
Print	Print file or selection	Unassigned
Project	Set or clear project	Unassigned
Prompt	Request text argument	Unassigned

Table 7.10 (continued)

Function	Description	Keys
Psearch	Search forward for pattern or text	F3
Pwbhelp	Help topic lookup	Unassigned
Pwbhelpnext	Relative help topic lookup	CTRL+F1
Pwbhelpsearch	Global full-text help search	Unassigned
Pwbrowse1stdef	Go to first definition	Unassigned
Pwbrowse1stref	Go to first reference	Unassigned
Pwbrowsecalltree	Browse Call Tree (Fwd/Rev)	Unassigned
Pwbrowseclhier	Browse Class Hierarchy	Unassigned
Pwbrowsecltree	Browse Class Tree (Fwd/Rev)	Unassigned
Pwbrowsefuhier	Browse Function Hierarchy	Unassigned
Pwbrowsegotodef	Browse Goto Definition	Unassigned
Pwbrowsegotoref	Browse Goto Reference	Unassigned
Pwbrowselistref	Browse List References	Unassigned
Pwbrowsenext	Browse Next	CTRL+NUM+
Pwbrowseoutline	Browse Module Outline	Unassigned
Pwbrowsepop	Go to previously browsed location	Unassigned
Pwbrowseprev	Browse Previous	CTRL+NUM-
Pwbrowseviewrel	Browse View Relationship	Unassigned
Pwbrowsewhref	Browse Which Reference?	Unassigned
Pwbwindow	Open a PWB window	Unassigned
Pword	Move forward one word	CTRL+F, CTRL+RIGHT
Qreplace	Replace with confirmation	CTRL+\
Quote	Insert literal key	CTRL+P
Record	Toggle macro recording	SHIFT+CTRL+R
Refresh	Reread or discard file	SHIFT+F7
Repeat	Repeat the last editing operation	Unassigned
Replace	Replace pattern or text	CTRL+L
Resize	Resize window	Unassigned
Restcur	Restore saved position	Unassigned
Right	Move right	CTRL+D, RIGHT
Saveall	Save all modified files	Unassigned
Savecur	Save cursor position	Unassigned
Sdelete	Delete streams	Unassigned
Searchall	Highlight occurrences of pattern or text	Unassigned
Selcur	Select to saved position	Unassigned

Table 7.10 (continued)

Function	Description	Keys
Select	Select text	SHIFT+PGUP, SHIFT+CTRL+PGUP, SHIFT+PGDN, SHIFT+CTRL+PGDN, SHIFT+END, SHIFT+CTRL+END, SHIFT+HOME, SHIFT+CTRL+HOME, SHIFT+LEFT, SHIFT+CTRL+LEFT, SHIFT+UP, SHIFT+RIGHT, SHIFT+CTRL+RIGHT, SHIFT+DOWN
Selmode	Change selection mode: box	Unassigned
Selwindow	Move to window	F6
Setfile	Open or change files	F2
Sethelp	Opens, closes, and lists help files	SHIFT+CTRL+S
Setwindow	Adjust file in window	CTRL+]]
Shell	Start a shell or run a system command	SHIFT+F9
Sinsert	Insert a stream of blanks or break line	CTRL+J
Tab	Move to the next tab stop	TAB
Tell	Show key assignment or macro definition	CTRL+T
Unassigned	Remove a function assignment from a key	(All unassigned keys)
Undo	Undo and redo editing operations	ALT+BKSP, SHIFT+CTRL+BKSP
Up	Move up	CTRL+E, UP
Usercmd	Execute a custom Run menu command	Unassigned
Window	Move to next or previous window	Unassigned
Winstyle	Add or remove scroll bars	CTRL+F6

Cursor-Movement Commands

PWB provides the following commands to navigate through text. In addition to the commands in the PWB editor, the Source Browser provides powerful commands to navigate through the source of your programs.

Table 7.11 Cursor-Movement Commands

Cursor Movement	Command	Keys
Up one line	Up	UP
Down one line	Down	DOWN
Left one column	Left	LEFT
Right one column	Right	RIGHT
Upper-left corner of window	Home	GOTO
Top of window	Meta Up	F9 UP
Bottom of window	Meta Down	F9 DOWN
Leftmost column in window	Meta Left	F9 LEFT
Rightmost column in window	Meta Right	F9 RIGHT
Lower-right corner of window	Meta Home	F9 GOTO
Up one window	Mpage	PGUP
Down one window	Ppage	PGDN
Column one	Meta Begline	F9 HOME
One column past window width	Meta Endline	F9 END
Back one word	Mword	CTRL+LEFT
Forward one word	Pword	CTRL+RIGHT
Beginning of line	Begline	HOME
End of line	Endline	END
Next paragraph	Ppara	Unassigned
Previous paragraph	Mpara	Unassigned
End of paragraph	Meta Ppara	F9 Unassigned
End of previous paragraph	Meta Mpara	F9 Unassigned
Beginning of file	Begfile	CTRL+HOME
End of file	Endfile	CTRL+END
To specific line number	Arg <i>number</i> Mark	ALT+A <i>number</i> CTRL+M
Position before last scroll	Arg Mark	ALT+A CTRL+M
Saved position	Restcur	Unassigned
Named mark	Arg <i>name</i> Mark	ALT+A <i>name</i> CTRL+M
Scroll window down one line	Mlines	CTRL+UP
Scroll window up one line	Plines	CTRL+DOWN
Scroll window so cursor at top	Arg Plines	ALT+A CTRL+DOWN
Scroll window so cursor at bottom	Arg Mlines	ALT+A CTRL+UP
Scroll window so cursor at home	Arg Setwindow	ALT+A CTRL+]

Arg

Key ALT+A

Arg

Begin an argument to a function or begin a selection.

After you execute **Arg**, PWB displays Arg[1] on the status bar. Each time you execute **Arg**, PWB increments the Arg count.

PWB functions perform variations of their action depending on the Arg count and the “Meta state.” You can use the **Meta** and **Arg** function prefixes in any order. See: **Meta**.

Examples

► To select text or create a function argument:

1. Execute **Arg** (ALT+A).
2. Execute a cursor-movement function.

Or hold down the SHIFT key and click the left mouse button.

PWB creates a stream, box, or line selection based on the current selection mode. A selection in each of these modes creates a function argument called “streamarg,” “boxarg,” or “linearg,” respectively.

► To create a text argument:

1. Execute **Arg** (ALT+A).
2. Type the text of the argument.

When you type the first character of the argument, PWB displays the Text Argument dialog box where you can enter the textarg without modifying your file. The Text Argument dialog box does not have an OK button; instead, you execute the function to which you are passing the text argument. Choose Cancel to save the text and do nothing.

► To “pick up” text from a window:

1. Select the text that you want to use in the Text Argument dialog box.
2. Execute **Lasttext** (CTRL+O).

PWB copies the selected text into the text argument dialog box.

► To cancel an argument or selection:

- Execute **Cancel** (ESC).

Returns	The return value of Arg cannot be tested.
See	Cancel, Lastselect, Lasttext, Meta, Prompt

Arrangewindow

Key Unassigned

Arrangewindow

Cascades all unminimized windows on the desktop. Does not affect minimized windows. See: **_pwbcascade**.

Arg Arrangewindow (ALT+A Unassigned)

Arranges all unminimized windows on the desktop. Does not affect minimized windows. See: **_pwbarrange**.

Meta Arrangewindow (F9 Unassigned)

Tiles up to 16 unminimized windows. Does not affect minimized windows. See: **_pwbtile**.

Meta Arg Arrangewindow (F9 ALT+A Unassigned)

Arranges all icons (minimized windows) on the desktop.

Returns	True	Windows or icons arranged.
	False	Nothing to arrange, or more than 16 windows open.

Assign

Key ALT+=

The **Assign** function assigns a function to a keystroke, defines a macro, or sets a PWB switch. You can also assign keys and set switches by using the commands in the Options menu. To see the current assignment for a key or the definition of a macro, use Options Keys Assignments or the **Tell** function (CTRL+T). See: **Tell**.

Assign

Performs the assignment using the text on the current line. If the line ends with a line continuation, PWB uses the next line, and so on for all continued lines.

Arg Assign (ALT+A ALT+=)

Same as **Assign**, except uses text starting from the cursor.

Arg *textarg* Assign (ALT+A *textarg* ALT+=)
Performs the assignment using the specified *textarg*.

Arg *mark* Assign (ALT+A *mark* ALT+=)
Performs the assignment using the text from the line at the cursor to the specified mark. The *mark* argument can be either a line number or a previously defined mark name. See: **Mark**.

Arg *boxarg* | *linearg* | *streamarg* Assign (ALT+A *boxarg* | *linearg* | *streamarg* ALT+=)
Performs the assignment using the selected text. Ignores blank and comment lines.

Returns

True	Assignment successful.
False	Assignment invalid.

Example ► **To set the Tabstops switch to 8:**

1. Execute **Arg** (ALT+A).
2. Type the following switch assignment:

```
tabstops:8
```
3. Execute **Assign** (ALT+=).

Update

Assign

Arg Assign

With PWB 1.x, **Assign** and **Arg Assign** do not recognize line continuations. With PWB 2.00, they use all continued lines for the assignment.

Arg *streamarg* Assign

With PWB 1.x, a *streamarg* is not allowed. With PWB 2.00, **Assign** accepts a *streamarg*.

Arg ? Assign

With PWB 1.x, this form of the **Assign** function displays the current assignments for all functions, switches, and macros in the “<ASSIGN>Current Assignments and Switch Settings” pseudofile.

With PWB 2.00, the <ASSIGN> pseudofile does not exist; therefore, this form of the **Assign** function is obsolete. If you use this command or execute a macro that executes this command, PWB issues the error:

```
Missing ':' in '?'
```

PWB is expecting an assignment or definition using the name ?, which is a legal macro name.

Backtab

Key SHIFT+TAB

Backtab

Moves the cursor to the previous tab stop on the line.

Returns True Cursor moved.
 False Cursor is at left margin.

Update PWB 2.0 supports variable tab stops. PWB 1.x supports only fixed-width tab stops.

See **Tab, Tabstops**

Begfile

Key CTRL+HOME

Begfile

Moves the cursor to the beginning of the file.

Returns True Cursor moved.
 False Cursor not moved; the cursor is already at the beginning of the file.

See **Endfile**

Begline

Key HOME

Begline

Places the cursor on the first nonblank character in the line.

Meta Begline (F9 HOME)

Places the cursor in the first character position of the line (column one).

Returns True Cursor moved.
 False Cursor not moved; the cursor is already at the destination.

Example The following macro moves the cursor to column one, then toggles between column one and the first nonblank character of the line.

```
toggle_begline := Left ->x Meta :>x Begline
```

The result of the **Left** function is tested to determine if the cursor is already in column one. If the cursor is in column one, PWB skips the **Meta** and executes **Begline** to move to the first nonblank character. If the cursor is not in column one, PWB executes **Meta Begline** to move there.

Example This macro mimics the behavior of the BRIEF HOME key:

```
bhome:= Meta Begline +> Home +> Begfile
```

The result of **Meta Begline** (go to column 1 on the line) is tested to determine if the cursor moved. If the cursor moved, the test (+>) succeeds and the macro exits. If the cursor did not move, the cursor is already in column 1, so the macro advances to the home position with **Home**. If the cursor did not move going to the home position, the macro advances to the beginning of the file with **Begfile**.

See **Left, Meta**

Cancel

Key ESC

Cancel

Cancels the current selection, argument, or operation. If a message appears on the status bar, the **Cancel** function restores the original contents of the status bar.

If a dialog box or menu is open, **Cancel** closes the dialog box or menu and takes no further action. If Help on a dialog box, menu, or message box is being displayed, **Cancel** closes the Help dialog box.

Returns **Cancel** always returns true.

See **Arg**

Cancelsearch

Key Unassigned

Cancelsearch

Cancels a background search.

The Search Results window contains the partial results of the aborted search and is not flushed. You can browse matches listed in the Search Results by using the Next Match, Previous Match, and Goto Match commands from the Search menu and by using the **Nextsearch** function (Unassigned).

Cancelsearch applies only to multithreaded environments.

Returns True Background search was canceled.

False No background search in progress.

See **Nextsearch**, **_pwbnextlogmatch**, **_pwbpreviouslogmatch**, **_pwbgotomatch**

Cdelete

Key CTRL+G

Cdelete

Deletes the previous character, excluding line breaks. If the cursor is in column 1, **Cdelete** moves the cursor to the end of the previous line.

In insert mode, **Cdelete** deletes the previous character, reducing the line length by 1.

In overtype mode, **Cdelete** deletes the previous character and replaces it with a space character. If the cursor is beyond the end of the line, the cursor moves to the immediate right of the last character on the line.

Emacsdel is similar to **Cdelete**. However, in insert mode, **Emacsdel** deletes line breaks; in overtype mode beyond the end of the line, it does not automatically move to the end of the line.

Returns True Cursor moved.
 False Cursor not moved.

See **Delete, Emacscdel, Ldelete, Sdelete**

Clearmsg

Key Unassigned

Clearmsg
Clears the contents of the Build Results window.

Arg Clearmsg (ALT+A Unassigned)
Clears the current set of messages in the Build Results window.

Returns True Cleared a message set or the contents of Build Results.
 False The Build Results window is empty.

See **Nextmsg, _pwbnextmsg, _pwbprevmsg, _pwbsetmsg**

Clearsearch

Key Unassigned

Clearsearch
Clears the contents of the Search Results window.

Arg Clearsearch (ALT+A Unassigned)
Clears the current set of matches in the Search Results window.

Returns True Cleared a match set or the contents of Search Results.
 False The Search Results window is empty.

See **Clearmsg, Logsearch, _pwbnextlogmatch, _pwbpreviouslogmatch, _pwbgotomatch**

Closefile

Key Unassigned

Closefile

Closes the file in the active window. If no files remain in the window's file history, the window is also closed.

Arg Closefile (ALT+A Unassigned)

Closes the file named by the text at the cursor.

Arg *linearg* | *boxarg* | *streamarg* Closefile

(ALT+A *linearg* | *boxarg* | *streamarg* Unassigned)

Closes the file named by the selected text.

Arg *textarg* Closefile (ALT+A *textarg* Unassigned)

Closes the specified file.

Returns True The file was closed.

False No file was closed.

See Refresh, `_pwbclosefile`

Compile

Key CTRL+F3

The **Compile** function compiles and builds targets in the project or runs external commands, capturing the result of the operation in the Build Results window. Under multithreaded environments the commands run in the background.

Arg Compile (ALT+A CTRL+F3)

Compiles the current file. This is equivalent to Project Compile File. **Arg Compile** fails if no project is open. See: `_pwbcompile`.

Arg *textarg* Compile (ALT+A *textarg* CTRL+F3)

Builds the target specified by *textarg*. This is equivalent to Build Target command on the Project menu. **Arg *textarg* Compile** fails if no project is open.

To build the current project, execute **Arg a11 Compile**.

Arg Meta *textarg* Compile (ALT+A *textarg* F9 CTRL+F3)

Rebuilds the specified target and its dependents. See: `_pwbrebuild`.

This command is equivalent to specifying the `NMAKE /a` option. Note that you can also include `NMAKE` command-line macro definitions in the text you pass to the **Compile** function.

Arg Meta Compile (ALT+A F9 CTRL+F3)

Aborts the background compile after prompting for confirmation. Also clears the queue of pending background operations (if any).

Arg Arg *textarg* Compile (ALT+A ALT+A *textarg* CTRL+F3)

Runs the program or operating-system command specified by *textarg*. The output is displayed in the Compile Results window.

Under multithreaded environments, the program runs in the background, and the Compile Results window is updated as the program executes. Several programs can be queued for background execution.

Do not use this command to execute an interactive program. The program is able to change the display but may not receive input. To run an interactive program, use the **Shell** function (SHIFT+F9).

Returns	True	Operation successfully initiated.
	False	Operation not initiated.

Copy

Keys CTRL+INS, SHIFT+NUM*

Menu Edit menu, Copy command

Copy

Copies the current line to the clipboard.

Arg Copy (ALT+A CTRL+INS)

Copies text from the cursor to the end of the line. The text is copied to the clipboard, but the line break is not included.

Arg *boxarg* | *linearg* | *streamarg* Copy

(ALT+A *boxarg* | *linearg* | *streamarg* CTRL+INS)

Copies the selected text to the clipboard.

Arg *textarg* Copy (ALT+A *textarg* CTRL+INS)

Copies the specified *textarg* to the clipboard.

Arg *mark* Copy (ALT+A *mark* CTRL+INS)

Copies the text from the cursor to the mark. The text is copied to the clipboard. The *mark* argument can be either a line number or a previously defined mark.

See: **Mark**.

The text is copied as a boxarg or linearg depending on the relative positions of the cursor and the mark. If the cursor and the mark are in the same column, the text is copied as a linearg. If the cursor and the mark are in different columns, the text is copied as a boxarg.

Arg number Copy (ALT+A *number* CTRL+INS)

Copies the specified number of lines to the clipboard, starting with the current line. For example, **Arg 5 Copy** copies five lines to the clipboard.

Returns Copy always returns true.

See **Delete, Ldelete, Sdelete, Paste**

Curdate

Key Unassigned

Curdate

Types the current date at the cursor in the format *day-month-year*, for example: 17-Apr-1999.

Returns True Date typed.
False Typing the date would make the line too long.

See **Curday, Curfile, Curfilenam, Curfileext, Curtime**

Curday

Key Unassigned

Curday

Types the three-letter abbreviation for the current day of the week, as follows:
Mon Tue Wed Thu Fri Sat Sun.

Returns True Day typed.
False Typing the day would make the line too long.

See **Curdate, Curfile, Curfilenam, Curfileext, Curtime**

Curtime

Key Unassigned

Curtime

Types the current time in the format *hours:minutes:seconds*, for example, 17:08:32.

Returns True Time typed.
False Typing the time would make the line too long.

See **Curdate, Curday, Curfile, Curfilenam, Curfileext**

Delete

Keys SHIFT+DEL, SHIFT+NUM-

Menu Edit menu, Cut command

Delete

Deletes the single character at the cursor, excluding line breaks. It does not copy the deleted character onto the clipboard. Note that the **Delete** function can delete more than one character, depending on the current selection mode.

Arg Delete (ALT+A SHIFT+DEL)

Deletes from the cursor to the end of the line. The deleted text is copied onto the clipboard. In stream selection mode, the deletion includes the line break and joins the current line to the next line.

Arg boxarg | linearg | streamarg Delete

(ALT+A *boxarg | linearg | streamarg* SHIFT+DEL)

Deletes the selected text. The text is copied on to the clipboard.

Meta ... Delete (F9 ... SHIFT+DEL)

As above but discards the deleted text. The contents of the clipboard are not changed.

Returns Delete always returns true.

Down

Keys DOWN, CTRL+X

Down

Moves the cursor down one line. If a selection has been started, it is extended by one line. If this movement results in the cursor moving out of the window, the window is adjusted downward as specified by the **Vscroll** switch.

Meta Down (F9 DOWN)

Moves the cursor to the bottom of the window without changing the column position.

Returns True Cursor moved.
False Cursor did not move; the cursor is at the destination.

See **Up**

Emacscdel

Keys BKSP, SHIFT+BKSP

Emacscdel

Deletes the previous character. If the cursor is in column 1, **Emacscdel** moves the cursor to the end of the previous line.

In insert mode, **Emacscdel** deletes the previous character, reducing the length of the line by 1. If the cursor is in column one, **Emacscdel** deletes the line break, joining the current line to the previous line.

In overtype mode, **Emacscdel** deletes the previous character and replaces it with a space character. If the cursor is in column 1, **Emacscdel** moves the cursor to the end of the previous line and does not delete the line break.

Emacscdel is similar to **Cdelete**, but **Cdelete** never deletes line breaks; in overtype mode beyond the end of the line, **Cdelete** automatically moves to the end of the line.

Returns True Cursor moved.
 False Cursor not moved.

See **Cdelete, Delete, Ldelete, Sdelete**

Emacsnewl

Keys ENTER, NUMENTER

Emacsnewl

In insert mode, starts a new line. In overtype mode, moves the cursor to the beginning of the next line. PWB automatically positions the cursor on the new line, depending on the setting of the **Softcr** switch.

Returns **Emacsnewl** always returns True.

Update In PWB 1.x, PWB performs special automatic indentation for C files. In PWB 2.00, language-specific automatic indentation is handled by language extensions if the feature is enabled. Otherwise, PWB uses its default indentation rules.

See **Newline, Softcr, C_Softcr**

Endfile

Key CTRL+END

Endfile

Places the cursor at the end of the file.

Returns	True	Cursor moved.
	False	Cursor did not move; the cursor is at the end of the file.

See	Begfile
------------	----------------

Endline

Key	END
------------	-----

Endline

Moves the cursor to the immediate right of the last character on the line.

Meta Endline (F9 END)

Moves the cursor to the column that is one column past the active window width.

Returns	True	Cursor moved.
	False	Cursor did not move; the cursor is at the destination.

See	Begline, Traildisp, Trailspace
------------	---------------------------------------

Environment

Key	Unassigned
------------	------------

Environment

Executes the current line as an environment-variable setting.

For example, if the current line contains the following text when you execute **Environment**:

```
PATH=C:\UTIL;C:\DOS
```

PWB adds this setting to the current environment table. The effect is the same as the operating-system SET command. PWB uses the new environment variable for the rest of the session (including shells).

Depending on the settings of the **Envcursave** and **Envprojsave** switches, PWB saves the environment table for PWB sessions and/or projects.

See: **Envcursave, Envprojsave.**

Arg *textarg* Environment (ALT+A *textarg* Unassigned)
Executes the argument as an environment-variable setting.

Arg *linearg* | *boxarg* Environment (ALT+A *linearg* | *boxarg* Unassigned)
Executes each selected line or line fragment as an environment-variable setting.

Meta Environment (F9 Unassigned)
Performs environment-variable substitutions for all variables on the current line, replacing each variable with its value.

The syntax for an environment variable is

$\$(ENV)$ | $\$ENV$:

where *ENV* is the uppercase name of the environment variable.

Arg Meta Environment (ALT+A F9 Unassigned)
Performs environment-variable substitutions (described above) for the text from the cursor to the end of the line.

Arg *boxarg* | *linearg* | *streamarg* Meta Environment
(ALT+A *boxarg* | *linearg* | *streamarg* F9 Unassigned)
Performs environment-variable substitutions for the selected text.

Returns

True	Environment variable successfully set or substituted.
False	Syntax error or line too long.

Update Because the <ENVIRONMENT> pseudofile no longer exists, this form of the **Environment** function is obsolete; it is replaced by the Environment command on the Options menu.

Execute

Key F7

The **Execute** function executes PWB functions and macros by name. It allows you to execute commands that are not assigned to a key or execute a sequence of commands in one step.

The **Execute** function executes the commands by the same rules as macros. Function prompts are suppressed, and you can use the macro flow-control and macro prompt directives. You do not need to define a macro to use these features.

Arg Execute (ALT+A F7)
Executes the text from the cursor to the end of the line as a PWB macro.

Arg *linearg* | *textarg* **Execute** (ALT+A *linearg* | *textarg* F7)
Executes the specified text as a PWB macro.

Returns

True	Last executed function returned true.
False	Last executed function returned false.

Exit

Key F8

Exit

If you specified multiple files on the PWB command line, PWB advances to the next file. Otherwise, PWB quits and returns control to the operating system.

If the **Autosave** switch is set to yes, the file is saved if it has been modified. If **Autosave** is no and the file is modified, PWB prompts for confirmation to save the file.

Meta Exit (F9 F8)

Performs like **Exit** with the **Autosave** switch set to no, independent of the current setting of **Autosave**. If you have changed any files, PWB asks for confirmation to save before exiting.

Arg Exit (ALT+A F8)

Like **Exit**, except PWB quits immediately without advancing to the next file (if any).

Arg Meta Exit (ALT+A F9 F8)

Like **Meta Exit**, except PWB quits immediately without advancing to the next file.

Returns No return value.

See `_pwbquit`

Graphic

Keys Assigned to most alphanumeric and punctuation keys.

Graphic

Types the character corresponding to the key that you pressed.

Returns True The character is typed.
False Typing the character would make the line too long.

See **Assign, Quote**

Home

Key GOTO (Numeric-keypad 5)

Home

Places the cursor in the upper-left corner of the window.

Meta Home (F9 GOTO)

Places the cursor in the lower-right corner of the window.

Returns True Cursor moved.
False Cursor not moved; it is already at the destination.

See **Begline, Endline, Left, Right**

Initialize

Key SHIFT+F8

Initialize

Discards all current settings, including extension settings, then reads the statements from the [PWB] section of TOOLS.INI.

Arg Initialize (ALT+A SHIFT+F8)

Reads the statements from a tagged section of TOOLS.INI. The tag name is specified by the continuous string of nonblank characters starting at the cursor.

Arg *textarg* Initialize (ALT+A *textarg* SHIFT+F8)

Reads the statements from the TOOLS.INI tagged section specified by *textarg*.

Example

The section tagged with

```
[PWB-name]
```

is initialized by the command

```
Arg name Initialize
```

Example

To reload the main section of TOOLS.INI without clearing other settings that you want to remain in effect, label the main section of TOOLS.INI with the tag:

```
[PWB PWB-main]
```

then use **Arg main Initialize** to recover your main settings instead of using **Initialize** with no arguments.

Returns

True Initialized tagged section in TOOLS.INI.

False Did not find tagged section in TOOLS.INI.

Information

Update

(obsolete)

The PWB 1.x **Information** function and its associated pseudofile <INFORMATION-FILE> are obsolete; they do not exist in PWB 2.00.

Insert

Key

Unassigned

Insert

Inserts a single-space character at the cursor, independent of the insert/overtyping mode.

Arg Insert (ALT+A Unassigned)

Breaks the line at the cursor.

Arg *boxarg* | *linearg* | *streamarg* **Insert**
(ALT+A *boxarg* | *linearg* | *streamarg* Unassigned)
Inserts space characters into the selected area.

Returns True Spaces or line break inserted.
False Insertion would make a line too long.

Example If paragraphs in your file consist of a sequence of lines beginning in the same column and are separated from other paragraphs by at least one blank line, the following macro indents a paragraph to the next tab stop:

```
para_indent:=_pwbboxmode Meta Mpara Down Begline Arg \
Meta Ppara Up Begline Tab Insert
```

This macro starts with the predefined PWB macro `_pwbboxmode` to set box selection mode, then creates a box selection from the beginning of the paragraph to the end, one tab stop wide. The **Insert** function inserts spaces in the selection.

See **Sinsert**, **Linsert**

Insertmode

Keys INS, CTRL+V

Insertmode

Toggles between insert mode and overwrite mode. If overwrite mode is on, the letter O appears on the status bar. The cursor can also change shape, depending on the **Cursormode** switch. See: **Cursormode**.

In insert mode, each character you type is inserted at the cursor. This insertion shifts the remainder of the line one position to the right.

In overwrite mode, the character you type replaces the character at the cursor.

Returns True PWB is in insert mode.
False PWB is in overwrite mode.

Lastselect

Key CTRL+U

Lastselect

Duplicates the last selection.

The Arg count and Meta state that were previously in effect are not duplicated—only the selection. The new Arg count is one, and the Meta state is the current Meta state. To use a higher Arg count, execute **Arg** (ALT+A). To toggle the Meta state, execute **Meta** (F9).

The re-created selection uses the same pair of *line:column* coordinates as the previous selection. Thus, different text can be selected if you have made additions or deletions to the file since the last selection.

See Arg, Lasttext, Meta

Lasttext

Key CTRL+O

Lasttext

Displays the last text argument in the Text Argument dialog box. You can edit the text and then execute any PWB function that accepts a text argument, or you can cancel the dialog box.

If you edit the text and then cancel the dialog box, PWB retains the modified text. Thus, when you execute **Lasttext** again, the new text appears in the dialog box.

Arg [[Arg]]... [[Meta]] **Lasttext** (ALT+A [[ALT+A]]... [[F9]] CTRL+O)

Displays the last text argument in the Text Argument dialog box with the specified Arg count and Meta state.

Arg [[Arg]]... *linearg* | *boxarg* | *streamarg* [[Meta]] **Lasttext**
(ALT+A [[ALT+A]]... *linearg* | *boxarg* | *streamarg* [[F9]] CTRL+O)

Displays the first line of the selection in the Text Argument dialog box with the specified Arg count and Meta state.

Returns The return value of **Lasttext** cannot be tested.

Example

The `OpenInclude` macro that follows opens an include file named in the next `#include` directive. The macro demonstrates a technique using the `Lasttext` function to pick up text from the file and modify it without modifying the file or the clipboard.

```
OpenInclude := \
  Up Meta Begline Arg Arg "[ \t]*#[ \t]*include" Psearch -> \
  Arg Arg "[<>\\]" Psearch -> Right Savecur Psearch -> \
  Selcur Lasttext Begline "$INCLUDE:" Openfile <n +> \
  Lastselect Openfile <
```

In the fourth line, `Lasttext` pulls the selected filename into the Text Argument dialog box. The text argument is modified to prepend `$INCLUDE:` before passing it to the `Openfile` function.

Example

In some macro-programming situations, you don't want to use the text immediately. Instead, you need to pick up some text, do some other processing, then use the text. In this situation, use the phrase:

(make selection) **Lasttext Cancel ...**

This picks up the text, then cancels the Text Argument dialog box. The selected text remains in the `Lasttext` buffer for later use. To reuse the text, call `Lasttext` again.

See

Arg, Lastselect, Meta, Prompt

Ldelete

Key

CTRL+Y

Ldelete

Deletes the current line and copies it to the clipboard.

Arg Ldelete (ALT+A CTRL+Y)

Deletes text from the cursor to the end of the line and copies it to the clipboard.

Arg mark Ldelete (ALT+A *mark* CTRL+Y)

Deletes the text from the line at the cursor to the line specified by *mark* and copies it to the clipboard. The mark cannot be a line number.

Arg number Ldelete (ALT+A *number* CTRL+Y)

Deletes the specified number of lines starting from the line at the cursor and copies them to the clipboard.

Arg *boxarg* | *linearg* **Ldelete** (ALT+A *boxarg* | *linearg* CTRL+Y)

Deletes the specified text and copies it to the clipboard. The argument is a *linearg* or *boxarg* regardless of the current selection mode. The argument is a *linearg* if the starting and ending points are in the same column.

Meta ... Ldelete (F9 ... CTRL+Y)

As above but discards the deleted text. The clipboard is not changed.

Returns **Ldelete** always returns true.

See **Cdelete**, **Delete**, **Emacsdel**, **Sdelete**

Left

Keys LEFT, CTRL+S

Left

Moves the cursor one character to the left. If this movement results in the cursor moving out of the window, the window is adjusted to the left as specified by the **Hscroll** switch.

Meta Left (F9 LEFT)

Moves the cursor to the first column in the window.

Returns True Cursor moved.
False Cursor not moved; the cursor is in column one.

See **Begline**, **Down**, **Endline**, **Home**, **Right**, **Up**

Linsert

Key CTRL+N

Linsert

Inserts one blank line above the current line.

Arg Linsert (ALT+A CTRL+N)

Inserts or deletes blanks at the beginning of a line to move the first nonblank character to the cursor.

Arg *boxarg* | *linearg* **Linsert** (ALT+A *boxarg* | *linearg* CTRL+N)

Inserts blanks within the specified area.

The argument is a *linearg* or *boxarg* regardless of the current selection mode. The argument is a *linearg* if the starting and ending points are in the same column.

Arg *mark* **Linsert** (ALT+A *mark* CTRL+N)

Like *boxarg* | *linearg* except the specified area is given by the cursor position and the position of the specified mark. The *mark* argument must be a named mark: it cannot be a line number. See: **Mark**.

Returns **Linsert** always returns true.

See **Insert**, **Sinsert**

Logsearch

Key Unassigned

Logsearch

Toggles the search-logging state.

The default search-logging mode when PWB starts up is determined by the **Enterlogmode** switch.

Returns True Search logging turned on.
False Search logging turned off.

Mark

Key CTRL+M

The **Mark** function moves the cursor to a mark or specific location, defines marks, and deletes marks. Note that you cannot set a mark at specific text in a PWB window such as Help; PWB marks only the window position.

If you want to save marks between sessions, assign a filename to the **Markfile** switch or use the Set Mark File command on the Search menu.

Mark (CTRL+M)

Moves the cursor to the beginning of the file.

Arg Mark (ALT+A CTRL+M)

Restores the cursor to its location prior to the last window scroll. Use **Arg Mark** to return to your previous location after a search or other large jump.

Arg number Mark (ALT+A *number* CTRL+M)

Moves the cursor to the beginning of the line specified by *number* in the current file. Line numbering starts at 1.

Arg textarg Mark (ALT+A *textarg* CTRL+M)

Moves the cursor to the specified mark.

Arg Arg textarg Mark (ALT+A ALT+A *textarg* CTRL+M)

Defines a mark at the cursor position. The name of the mark is specified by *textarg*.

Arg Arg textarg Meta Mark (ALT+A ALT+A *textarg* F9 CTRL+M)

Deletes the specified mark. This form of the **Mark** function always returns true.

Returns

True Move, definition, or deletion successful.

False Invalid argument or mark not found.

See

Markfile, Restcur, Savecur, Selcur

Maximize

Key

Unassigned

Maximize

Expands the window to its maximum size. If the window is already maximized, the window is restored.

When the window is maximized and scroll bars are turned off by using the **Winstyle** function, PWB turns off the window borders. This is the "clean screen" look.

Meta Maximize (F9 Unassigned)

Restores the window to its original size.

Returns True Window is maximized.
 False Window is restored.

See **Minimize, Winstyle**

Menukey

Key ALT

Menukey

Activates the menu bar. Unlike other PWB functions, **Menukey** can be assigned to only one key. It cannot be assigned to a combination of keys.

Returns You cannot test the return value of **Menukey**.

Message

Key Unassigned

Message

Clears the status bar.

Arg Message (ALT+A Unassigned)

Displays the text from the cursor to the end of the line on the status bar.

Arg *textarg* Message (ALT+A *textarg* Unassigned)

Displays *textarg* on the status bar.

Meta ... Message (F9 ... Unassigned)

As above and also repaints the screen.

Returns **Message** always returns true.

Example The following macro is useful when writing new macros (the ! is the macro name):

```
! := Meta Message
```

With this definition you can place an exclamation point in your macros wherever you want a screen update. If you also want to display a status-bar message at the time of the update, use the phrase:

... **Arg** "*text of message*" ! ...

See **Prompt**

Meta

Key F9

Meta

Modifies the action of the function it prefixes.

When the Meta state is turned on, the letter A (for “Alternate”) appears in the status bar. You can use the **Meta** and **Arg** function prefixes in any order.

Returns True Meta state turned on.
False Meta state turned off.

See **Arg, Lasttext, Lastselect**

Mgrep

Key Unassigned

The **Mgrep** function searches all the files listed in the **Mgreplist** macro. PWB places all matches in the Search Results window. Under multithreaded environments, PWB performs the search in the background.

To browse the list of matches, use **_pwbnextlogmatch** (CTRL+SHIFT+F3), **_pwbpreviouslogmatch** (CTRL+SHIFT+F4), and the **Nextsearch** function (Unassigned).

Mgrep (Unassigned)

Searches for the previously searched string or pattern.

Arg Mgrep (ALT+A Unassigned)

Searches for the string specified by the characters from the cursor to the first blank character.

Arg *textarg* Mgrep (ALT+A *textarg* Unassigned)

Searches for *textarg*.

Arg Arg Mgrep (ALT+A ALT+A Unassigned)

Searches for the regular expression specified by the characters from the cursor to the first blank character.

Arg Arg *textarg* Mgrep (ALT+A ALT+A *textarg* Unassigned)

Searches for the regular expression specified by *textarg*.

Meta ... Mgrep (F9 ... Unassigned)

As above except that the value of the **Case** switch is reversed for the search.

Returns

True

With MS-DOS, indicates that a match was found. With multithreaded environments, indicates that a background search was successfully initiated.

False

No matches, no search pattern specified, search pattern invalid, or search terminated by CTRL+BREAK.

Update

In PWB 2.00, search and build results and their browsing functions are separate. A background build operation and a background search can be performed simultaneously.

In PWB 1.x, search and build results appear in the same window, and are browsed with the same commands. A background build operation and a multifile search cannot be performed at the same time in PWB 1.x.

Minimize

Key

Unassigned

Minimize

Shrinks the active window to an icon (a minimized window). If the window is already minimized, restores the window.

Arg Minimize (ALT+A Unassigned)

Minimizes all open windows.

Meta Minimize (F9 Unassigned)

Restores the window to its unminimized state.

Returns True Window minimized: the window is an icon.
 False Window restored: the window is not an icon.

See **Maximize**

Mlines

Keys CTRL+UP, CTRL+W

Mlines

Scrolls the window down as specified by the **Vscroll** switch.

Arg Mlines (ALT+A CTRL+UP)

Scrolls the window so the line at the cursor moves to the bottom of the window.

Arg number Mlines (ALT+A *number* CTRL+UP)

Scrolls the window down by *number* lines.

Returns True Window scrolled.
 False Invalid argument.

See **Plines**

Movewindow

Key Unassigned

Movewindow

Enters window-moving mode. In window-moving mode, only the following actions are available:

Action	Key
Move up one row	UP
Move down one row	DOWN
Move left one column	LEFT
Move right one column	RIGHT
Accept the new position	ENTER
Cancel the move	ESC

Arg number Movewindow (ALT+A *number* Unassigned)

Moves the upper-left corner of the window to the screen row specified by *number*.

Meta Arg number Movewindow (F9 ALT+A *number* Unassigned)

Moves the upper-left corner of the window to the screen column specified by *number*.

Returns	True	Window moved.
	False	Window not moved.

Mpage

Keys PGUP, CTRL+R

Mpage

Moves the cursor backward in the file by one window.

Returns	True	Cursor moved.
	False	Cursor not moved.

See Ppage

Mpara

Key Unassigned

Mpara

Moves the cursor to the beginning of the first line of the current paragraph. If the cursor is already on the first line of the paragraph, it is moved to the beginning of the first line of the preceding paragraph.

Meta Mpara (F9 Unassigned)

Moves the cursor to the first blank line preceding the current paragraph.

Returns	True	Cursor moved.
	False	Cursor not moved; no more paragraphs in the file.
See	Ppara	

Mreplace

Key Unassigned

Mreplace

Performs a find-and-replace operation across multiple files, prompting for the find-and-replacement strings and for confirmation at each occurrence.

Mreplace searches all the files listed in the special macro **Mgreplist**.

Arg Arg Mreplace (ALT+A ALT+A Unassigned)

Performs the same action as **Mreplace** but uses regular expressions.

Meta ... Mreplace (F9 ... Unassigned)

As above except reverses the sense of the **Case** switch for the operation.

Returns	True	At least one replacement made.
	False	No replacements made or operation aborted.

See **Mgrep, Mreplaceall, Qreplace, Replace**

Mreplaceall

Key Unassigned

Mreplaceall

Performs a find-and-replace operation across multiple files, prompting for the find-and-replacement strings. **Mreplaceall** searches all the files listed in the special macro **Mgreplist**.

Arg Arg Mreplaceall (ALT+A ALT+A Unassigned)

Performs the same action as **Mreplaceall** but uses regular expressions.

Meta ... Mreplaceall (F9 ... Unassigned)

As above except reverses the sense of the **Case** switch for the operation.

Returns True At least one replacement made.
 False No replacements made or operation aborted.

See **Mgrep, Mreplace, Qreplace, Replace**

Msearch

Key F4

Msearch

Searches backward for the previously searched string or pattern.

Arg Msearch (ALT+A F4)

Searches backward for the string specified by the text from the cursor to the first blank character.

Arg *textarg* Msearch (ALT+A *textarg* F4)

Searches backward for the specified text.

Arg Arg Msearch (ALT+A ALT+A F4)

Searches backward for the regular expression specified by the text from the cursor to the first blank character.

Arg Arg *textarg* Msearch (ALT+A ALT+A *textarg* F4)

Searches backward for the regular expression defined by *textarg*.

Meta ... Msearch (F9 ... F4)

As above except reverses the sense of the **Case** switch for the search.

Returns True String found.
 False Invalid argument, or string not found.

See **Mgrep, Psearch**

Mword

Keys CTRL+LEFT, CTRL+A

Mword

Moves the cursor to the beginning of the current word, or if the cursor is not in a word or at the beginning of the word, moves the cursor to the beginning of the previous word. A word is defined by the **Word** switch.

Meta Pword (F9 CTRL+RIGHT)

Moves the cursor to the immediate right of the previous word.

Returns True Cursor moved.
False Cursor not moved; there are no more words in the file.

See **Pword**

Newfile

Key Unassigned

The **Newfile** function creates a new pseudofile. If the **Newwindow** switch is set to yes, it opens a new window for the file.

Newfile (Unassigned)

Creates a new untitled pseudofile. The new pseudofile is given a unique name of the form:

`<Untitled.nnn>Untitled.nnn`

where *nnn* is a three-digit number starting with 001 at the beginning of each PWB session. The window title shows `Untitled.001`. Use the pseudofile name `<Untitled.001>` to refer to the file in a text argument or dialog box.

Arg Newfile (ALT+A Unassigned)

Creates a new pseudofile with the name specified by the text from the cursor to the end of the line. The resulting full pseudofile name is:

`"<Text on the line>Text on the line"`

Arg textarg Newfile (ALT+A *textarg* Unassigned)

Creates a new pseudofile with the name specified by *textarg*. The resulting full pseudofile name is:

`"<textarg>textarg"`

If you want to use a different short name and window title, use the full name as an argument to the **Setfile** or **Openfile** functions. For example, **Arg** "<temp>Temporary File" **Openfile** opens a pseudofile in a new window that has the title Temporary File.

Returns True Successfully created the pseudofile.
 False Unable to create the pseudofile.

Newline

Keys SHIFT+ENTER, SHIFT+NUMENTER

Newline

Moves the cursor to a new line.

If the **Softcr** switch is set to yes, PWB automatically indents to an appropriate position based on the type of file you are editing.

Meta Newline (F9 SHIFT+ENTER)

Moves the cursor to column 1 of the next line.

Returns Newline always returns true.

Update In PWB 1.x, PWB performs special automatic indentation for C files. In PWB 2.00, language-specific automatic indentation is handled by language extensions if the feature is enabled. Otherwise, PWB uses its default indentation rules.

See **Emacsnewl**

Nextmsg

Key Unassigned

Nextmsg

Advances to next message in the Build Results window.

Arg number Nextmsg (ALT+A *number* Unassigned)

Moves to the *n*th message in the current set of messages, where *n* is specified by *number*.

To move relative to the current message, use a signed number. For example, when *number* is +1, PWB moves to the next message, and when it is -1, PWB moves to the previous message.

Arg Nextmsg (ALT+A Unassigned)

Moves to the next message in the current set of messages that does not refer to the current file.

Meta Nextmsg (F9 Unassigned)

Advances to the next set of messages.

Arg Arg Nextmsg (ALT+A ALT+A Unassigned)

Sets the message at the cursor as the current message. This works only when the cursor is on a message in the Build Results window.

Returns

True Message found.
False No more messages found.

Update

In PWB 1.x, **Nextmsg** also browses the results of searches. In PWB 2.00, search results are browsed with the **Nextsearch** function.

Meta Nextmsg

In PWB 1.x, deletes the current set of messages and advances to the next set. In PWB 2.00, **Meta Nextmsg** does not delete the set. To delete sets of messages in PWB 2.00, use the **Clearmsg** function.

Meta Arg Arg Nextmsg

In PWB 1.x, closes the Compile Results window. In PWB 2.00, it behaves like **Arg Arg Nextmsg**.

See

Clearmsg

Nextsearch

Key

Unassigned

Nextsearch

Advances to the next match in the Search Results window.

Arg number Nextsearch (ALT+A *number* Unassigned)

Moves to the *n*th match in the current set of matches, where *n* is specified by *number*.

To move relative to the current match, use a signed number. For example, when *number* is +1, PWB moves to the next match, and when it is -1, PWB moves to the previous match.

Arg Nextsearch (ALT+A Unassigned)

Moves to the next match in the current set of matches that does not refer to the current file.

Meta Nextsearch (F9 Unassigned)

Advances to the next set of matches.

Arg Arg Nextsearch (ALT+A ALT+A Unassigned)

Sets the match at the cursor as the current match. This works only when the cursor is on a match in the Search Results window.

Update

In PWB 1.x, the results of searches are browsed using the **Nextmsg** function.

See

Clearsearch

Noedit

Key

Unassigned

The **Noedit** function toggles the no-edit state of PWB or the current file. When the no-edit state is turned on, PWB displays the letter R on the status bar and disallows modification of the file.

Noedit

Toggles the no-edit state. If you started PWB with the /R (read-only) option, **Noedit** removes the no-edit limitation.

Meta Noedit (F9 Unassigned)

Toggles the no-edit state for the current file. This form of the **Noedit** command works only for disk files and has no effect on pseudofiles.

If you have the **Editreadonly** switch set to no, PWB turns on the no-edit state for files that are marked read-only on disk. This function toggles the no-edit state for the file so that you can modify it.

Returns

True	File or PWB in no-edit state; modification disallowed.
False	File or PWB not in no-edit state; modification allowed.

Openfile

Key F10

The **Openfile** function opens a file in a new window, ignoring the **Newwindow** switch.

Arg Openfile (ALT+A F10)

Opens the file at the cursor in a new window. The name of the file is specified by the text from the cursor to the first blank character.

Arg *textarg* Openfile (ALT+A *textarg* F10)

Opens the specified file in a new window.

If the argument is a wildcard, PWB creates a pseudofile containing a list of files that match the pattern. To open a file from this list, position the cursor at the beginning of the name and use **Arg Openfile** or **Arg Setfile**.

Returns True File and window successfully opened.
False No argument specified, or file did not exist and you did not create it.

See **Newfile**, **Setfile**

Paste

Keys SHIFT+INS, SHIFT+NUM+

Menu Edit menu, Paste command

Paste (SHIFT+INS)

Copies the contents of the clipboard to the file at the cursor. The text is always inserted independent of the insert/overtyping mode.

If the clipboard contents were copied to the clipboard as a *linearg*, PWB inserts the contents of the clipboard above the current line. Otherwise, the contents of the clipboard are inserted at the cursor.

Arg *boxarg* | *linearg* | *streamarg* Paste

(ALT+A *boxarg* | *linearg* | *streamarg* SHIFT+INS)

Replaces the selected text with the contents of the clipboard.

Arg Paste (ALT+A SHIFT+INS)

Copies the text from the cursor to the end of the line. The text is copied to the clipboard and inserted at the cursor.

Arg *textarg* Paste (ALT+A *textarg* SHIFT+INS)

Copies *textarg* to the clipboard and inserts it at the cursor.

Arg Arg *filename* Paste (ALT+A ALT+A *filename* SHIFT+INS)

Copies the contents of the file specified by *textarg* to the current file above the current line.

Arg Arg !*textarg* Paste (ALT+A ALT+A !*filename* SHIFT+INS)

Runs *textarg* as an operating-system command, capturing the command's output to standard output. The output is copied to the clipboard and inserted above the current line.

You must enter the exclamation mark as shown.

Returns

True **Paste** always returns true except for the following cases.

False Tried **Arg Arg *filename* Paste** and file did not exist, or the pasted text could make a line too long.

Example

The following command copies a sorted copy of the file SAMPLE.TXT to the current file: **Arg Arg !SORT <SAMPLE.TXT Paste** (ALT+A ALT+A !SORT <SAMPLE.TXT SHIFT+INS).

Pbal

Key

CTRL+[

Pbal

Scans backward through the file, balancing parentheses (()) and brackets ([]). The first unmatched parenthesis or bracket is highlighted when found.

If an unbalanced parenthesis or bracket is found, it is highlighted and the corresponding character is inserted at the cursor. If no unbalanced characters are found, PWB displays a message box.

The search does not include the cursor position and looks for more opening brackets or parentheses than closing ones.

Arg Pbal (ALT+A CTRL+[)

Like **Pbal** except that it scans forward through the file and searches for right brackets or parentheses lacking opening partners.

Meta Pbal (F9 CTRL+[)

Like **Pbal** but does not insert the unbalanced character. If no unbalanced characters are found, moves to the matching character.

Arg Meta Pbal (ALT+A F9 CTRL+[])

Like **Arg Pbal** but does not insert the character. If no unbalanced characters are found, moves to the matching character.

Update	In PWB 1.x, the messages appear on the status bar. In PWB 2.00, they appear in a message box.	
Returns	True	Balance successful.
	False	Invalid argument, or no unbalanced characters found.
See	Infodialog	

Plines

Keys CTRL+DOWN, CTRL+Z

Plines

Scrolls the text up as specified by the **Vscroll** switch.

Arg Plines (ALT+A CTRL+DOWN)

Scrolls the text such that the line at the cursor is moved to the top of the window.

Arg number Plines (ALT+A *number* CTRL+DOWN)

Scrolls the text up by *number* lines.

Returns	True	Text scrolled.
	False	Invalid argument.

See **Mlines**

Ppage

Keys PGDN, CTRL+C

Ppage

Moves the cursor forward in the file by one window.

Returns True Cursor moved.
False Cursor not moved.

See Mpage

Ppara

Key Unassigned

Ppara

Moves the cursor to the beginning of the first line of the next paragraph.

Meta Ppara (F9 Unassigned)

Moves cursor to the beginning of the first blank line after the current paragraph.
If the cursor is not on a paragraph, moves the cursor to the first blank line after the next paragraph.

Returns True Cursor moved.
False Cursor not moved; no more paragraphs in the file.

See Mpara

Print

Key Unassigned

The **Print** function prints files or selections. If the **Printcmd** switch is set, PWB uses the command line given in the switch. Otherwise, PWB copies the file or selection to PRN. Under multithreaded environments, PWB runs the print command in the background.

Print (Unassigned)

Prints the current file.

Arg *textarg* Print (ALT+A *textarg* Unassigned)

Prints all the files listed in *textarg*. Use a space to separate each name from the preceding name. You can use environment variables to specify paths for the files.

Arg *boxarg* | *linearg* | *streamarg* Print

(ALT+A *boxarg* | *linearg* | *streamarg* Unassigned)

Prints the selected text.

Arg Meta Print (ALT+A F9 Unassigned)

Cancels the current background print.

Returns

True Print successfully submitted.

False Could not start print job.

Update

In PWB 1.x there is no way to cancel a background print.

Project

Key

Unassigned

Project

Open the last project.

Arg Project (ALT+A Unassigned)

Open the project makefile at the cursor as a PWB project. The name of the project is specified by the text from the cursor to the first blank character.

Arg *textarg* Project (ALT+A *textarg* Unassigned)

Open the project makefile specified by *textarg* as a PWB project.

Arg Arg Project (ALT+A ALT+A Unassigned)

Close the current project.

Arg Meta Project (ALT+A F9 Unassigned)

Open the project makefile at the cursor as a non-PWB project (foreign makefile).

Arg *textarg* Meta Project (ALT+A *textarg* F9 Unassigned)

Open the project makefile specified by *textarg* as a non-PWB project.

Returns	True	A project is open.
	False	A project is not open.
See	Lastproject	

Prompt

Key Unassigned

The **Prompt** function displays the Text Argument dialog box where you can enter a text argument. You can use this function interactively, but because it is mainly useful in macros, it is not assigned to a key by default. You usually use **Lasttext** or **Arg** to directly enter a text argument.

Prompt

Displays the Text Argument dialog box without a title. See: **Lasttext**

Arg Prompt (ALT+A Unassigned)

Uses the text of the current line from the cursor to the end of the line as the title.

Arg textarg Prompt (ALT+A *textarg* Unassigned)

Uses *textarg* as the title.

Arg boxarg | linearg | streamarg Prompt

(ALT+A *boxarg* | *linearg* | *streamarg* Unassigned)

Uses the selected text as the title. If the selection spans more than one line, the title is the first line of the selected text.

Returns	True	Textarg entered; the user chose the OK button.
	False	The dialog box was canceled.

Example With the following macro, PWB prompts for a Help topic:

```
QueryHelp := Arg "Help Topic to Find:" Prompt -> Pwbhelp
QueryHelp : Ctrl+Q
```

When you press CTRL+Q, PWB displays a dialog box with the string `Help Topic to Find:` as the title and waits for a response. PWB passes your response to the **Pwbhelp** function as if the command **Arg *textarg* Pwbhelp** had been executed. If you cancel the dialog box, **Prompt** returns false and the macro conditional `->` terminates the macro without executing **Pwbhelp**.

See **Assign**

Psearch

Key F3

Psearch

Searches forward for the previously searched string or pattern.

Arg Psearch (ALT+A F3)

Searches forward in the file for the string specified by the text from the cursor to the first blank character.

Arg *textarg* Psearch (ALT+A *textarg* F3)

Searches forward for the specified text.

Arg Arg Psearch (ALT+A ALT+A F3)

Searches forward in the file for the regular expression specified by the text from the cursor to the first blank character.

Arg Arg *textarg* Psearch (ALT+A ALT+A *textarg* F3)

Searches forward for the regular expression defined by *textarg*.

Meta ... Psearch (F9 ... F3)

As above but reverses the value of the **Case** switch for one search.

Returns

True	String found.
False	Invalid argument, or string not found.

Pwbhelp

Key Unassigned

Pwbhelp

Displays the default Help topic.

Arg Pwbhelp (ALT+A Unassigned)

Displays Help on the topic at the cursor. Equivalent to the macro `_pwbhelp_context` (F1).

Arg *textarg* Pwbhelp (ALT+A *textarg* Unassigned)

Displays Help on the specified text argument.

Arg *streamarg* Pwbhelp (ALT+A *streamarg* Unassigned)

Displays Help on the selected text. The selection cannot include more than one line.

Meta Pwbhelp (F9 Unassigned)

Prompts for a key, then displays Help on the function or macro assigned to the key you press.

If you press a key that is not assigned to a function or macro, PWB displays help on the **Unassigned** function. If you press a key that PWB does not recognize, the prompt remains displayed until you press a key that PWB recognizes.

Returns True Help topic found.
False Help topic not found.

Pwbhelpnext

Key CTRL+F1

Pwbhelpnext

Displays the next physical topic in the current Help database.

Meta Pwbhelpnext (F9 CTRL+F1)

Displays the previous Help topic on the backtrace list. This is the Help topic that you previously viewed. Up to 20 Help topics are retained in the backtrace list.

Equivalent to the Back button on the Help screens and the macro `_pwbhelp_back` (ALT+F1).

Arg Pwbhelpnext (ALT+A CTRL+F1)

Displays the next occurrence of the current Help topic within the Help system.

Equivalent to the macro **_pwbhelp_again** (Unassigned).

Use this command when the Help topic appears several times in the set of open Help databases.

Returns	True	Help topic found.
	False	Help topic not found.

Pwbhelpsearch

Key Unassigned

The **Pwbhelpsearch** function performs a global search of the Help system. The search is case insensitive unless you use the **Meta** form of **Pwbhelpsearch**, which uses the setting of the **Case** switch to determine case sensitivity.

Pwbhelpsearch (Unassigned)

Displays the results of the last global Help search.

Equivalent to the predefined macro **_pwbhelp_searchres** (Unassigned).

Arg Pwbhelpsearch (ALT+A Unassigned)

Searches Help for the word at the cursor.

Arg *textarg* Pwbhelpsearch (ALT+A *textarg* Unassigned)

Searches Help for the selected text.

Arg Arg Pwbhelpsearch (ALT+A ALT+A Unassigned)

Searches Help using the regular expression at the cursor.

Arg Arg *textarg* Pwbhelpsearch (ALT+A ALT+A *textarg* Unassigned)

Searches Help for the selected regular expression.

Meta ... Pwbhelpsearch (F9 ... Unassigned)

As above except the search is case sensitive if the **Case** switch is set to yes.

Returns	True	At least one match found.
	False	No matches found, or search canceled.

Pwbrowse Functions

Most of the **Pwbrowse...** functions provided by the PWBROWSE Source Browser extension display one of the Source Browser's dialog boxes. The Source Browser functions attached to Browse menu commands are listed in the following table.

Function	Browse Menu Command	Key
Pwbrowsecalltree	Call Tree (Fwd/Rev)	Unassigned
Pwbrowseclhier	Class Hierarchy	Unassigned
Pwbrowsecltree	Class Tree (Fwd/Rev)	Unassigned
Pwbrowsefuhier	Function Hierarchy	Unassigned
Pwbrowsegotodef	Goto Definition	Unassigned
Pwbrowsegotoref	Goto Reference	Unassigned
Pwbrowselistref	List References	Unassigned
Pwbrowsenext	Next	CTRL+NUM+
Pwbrowseoutline	Module Outline	Unassigned
Pwbrowseprev	Previous	CTRL+NUM-
Pwbrowseviewrel	View Relationship	Unassigned
Pwbrosewhref	Which Reference	Unassigned

The browser functions in the following table do not correspond to a Browse menu command.

Function	Description	Key
Pwbrowse1stdef	Go to 1st definition	Unassigned
Pwbrowse1stref	Go to 1st reference	Unassigned
Pwbrosepop	Go to previously browsed location	Unassigned

Pwbwindow

Key Unassigned

The **Pwbwindow** function opens PWB windows. If the specified window is already open, PWB switches to that window.

Arg Pwbwindow (ALT+A Unassigned)

Opens the PWB window with the name at the cursor. The name is specified by the text from the cursor to the first blank character.

Arg *textarg* Pwbwindow (ALT+A *textarg* Unassigned)

Opens the specified PWB window.

Arg Meta Pwbwindow (ALT+A F9 Unassigned)

Closes the PWB window specified by the name at the cursor.

Arg *textarg* Meta Pwbwindow (ALT+A *textarg* F9 Unassigned)

Closes the specified PWB window.

Returns

True	The specified window was opened.
False	The window could not be opened.

Pword

Keys CTRL+RIGHT, CTRL+F

Pword

Moves the cursor to the beginning of the next word. A word is defined by the **Word** switch.

Meta Pword (F9 CTRL+RIGHT)

Moves the cursor to the immediate right of the current word, or if the cursor is not in a word, moves it to the right of the next word.

Returns

True	Cursor moved.
False	Cursor not moved; there are no more words in the file.

See **Mword**

Qreplace

Key CTRL+A

The **Qreplace** function performs a find-and-replace operation on the current file, prompting for find-and-replacement strings and confirmation at each occurrence.

Qreplace (CTRL+A)

Performs the replacement from the cursor to the end of the file, wrapping around the end of the file if the **Searchwrap** switch is set to yes.

Arg boxarg | linearg | streamarg Qreplace

(ALT+A *boxarg* | *linearg* | *streamarg* CTRL+A)

Performs the replacement over the selected area.

Note that PWB does not adjust the selection at each replacement for changes in the length of the text. For *boxarg* and *streamarg*, PWB may replace text that was not included in the original selection or miss text included in the original selection.

Arg mark Qreplace (ALT+A *mark* CTRL+A)

Performs the replacement on text from the cursor to the specified mark. Replaces over text as if it were selected, according to the current selection mode. The *mark* argument cannot be a line number. See: **Mark**.

Arg number Qreplace (ALT+A *number* CTRL+A)

Performs the replacement for the specified number of lines, starting with the line at the cursor.

Arg Arg ... Qreplace (ALT+A ALT+A ... CTRL+A)

As above except using regular expressions.

Meta ... Qreplace (F9 ... CTRL+A)

As above except the sense of the **Case** switch is reversed for the operation.

Returns True At least one replacement was performed.

False String not found, or invalid pattern.

See **Mreplace**, **Replace**, **Searchwrap**

Quote

Key CTRL+P

Quote

Reads one key from the keyboard and types it into the file or dialog box. In a dialog box, the key is always CTRL+P, no matter what function or macro you may have assigned to CTRL+P for the editor.

This is useful for typing a character (such as TAB or CTRL+L) whose keystroke is assigned to a PWB function.

Returns True **Quote** always returns true except in the following case.
False Character would make line too long.

Record

Key SHIFT+CTRL+R

The **Record** function toggles macro recording. While a macro is being recorded, PWB displays the letter X on the status bar, and a bullet appears next to the Record On command from the Edit menu. If a menu command cannot be recorded, it is disabled while recording.

When macro recording is stopped, PWB assigns the recorded commands to the default macro name **Playback**. During the recording, PWB writes the name of each command to the definition of **Playback** in the Record window, which can be viewed as it is updated.

Macro recording in PWB does not record changes in cursor position accomplished by clicking the mouse. Use the keyboard if you want to include cursor movements in a macro.

Record (SHIFT+CTRL+R)

Toggles macro recording on and off.

Arg *textarg* **Record** (ALT+A *textarg* SHIFT+CTRL+R)

Turns on recording if it is off and assigns the name specified in the text argument to the recorded macro. Turns off recording if it is turned on.

Meta Record (F9 SHIFT+CTRL+R)

Toggles macro recording. While recording, no editing commands are executed until recording is turned off. Use this form of the function to record a macro without modifying your file.

Arg Record (ALT+A SHIFT+CTRL+R)

Arg Arg *textarg* Record (ALT+A ALT+A *textarg* SHIFT+CTRL+R)

Arg Arg Meta Record (ALT+A ALT+A F9 SHIFT+CTRL+R)

As above but if the target macro already exists, the commands are appended to the end of the macro.

Returns

True	Recording turned on.
False	Recording turned off.

Update In PWB 2.00, more menu commands can be recorded than with PWB 1.x.

Refresh

Key SHIFT+F7

Refresh

Prompts for confirmation and then rereads the file from disk, discarding its Undo history and all modifications to the file since the file was last saved.

Returns	Condition
True	File reread.
False	Prompt canceled

Arg Refresh (ALT+A SHIFT+F7)

Prompts for confirmation and then removes the file from the active window and the window's file history. If the active window is the last window that has the file in its history, the file is discarded from memory without saving changes, and the file is closed.

Returns	Condition
True	File removed from the window.
False	Prompt canceled, or bad argument. The file is not removed from the window.

Repeat

Key Unassigned

Repeat

Repeats the last editing action relative to the current cursor position. The **Repeat** function considers the following types of operations to be editing actions:

- Typing a contiguous stream of characters without entering a command or moving the cursor
- Deleting text
- Pasting from the clipboard

Repeat does not repeat macros or cursor movements.

Arg *number* **Repeat** (ALT+A *number* Unassigned)

Performs the last action the number of times specified by *number*.

Returns True Action repeated and returned true.

False Action repeated and returned false, or no action to repeat.

Replace

Key CTRL+L

The **Replace** function performs a find-and-replace operation on the current file, prompting for find and replacement strings. **Replace** substitutes all matches of the search pattern without prompting for confirmation.

Replace (CTRL+L)

Performs the replacement from the cursor to the end of the file, wrapping around the end of the file if the **Searchwrap** switch is on.

Arg *boxarg* | *linearg* | *streamarg* **Replace**
(ALT+A *boxarg* | *linearg* | *streamarg* CTRL+L)

Performs the replacement over the selected area.

Note that PWB does not adjust the selection at each replacement for changes in the length of the text. For *boxarg* and *streamarg*, PWB may replace text that was not included in the original selection or miss text included in the original selection.

Arg mark Replace (ALT+A *mark* CTRL+L)

Performs the replacement on text from the cursor to the specified mark. It searches the range of text as if it were selected, according to the current selection mode. The *mark* argument cannot be a line number.

Arg number Replace (ALT+A *number* CTRL+L)

Performs the replacement over the specified number of lines, starting with the current line.

Arg Arg ... Replace (ALT+A ALT+A ... CTRL+L)

As above except using regular expressions.

Meta ... Replace (F9 ... CTRL+L)

As above except the sense of the **Case** switch is reversed for the operation.

Returns

True At least one replacement was performed.

False String not found, or invalid pattern.

See

Qreplace, Searchwrap

Example

To use the replace function in a macro, use the phrase:

```
...Replace "pattern" Newline "replacement" Newline +>found...
```

Enter the replies to the prompts as you would when executing **Replace** interactively. This example also shows where to place the conditional to test the result of **Replace**.

You can specify special characters in the find-and-replacement strings by using escape sequences similar to those in the C language. Note that backslashes in the macro string must be doubled.

To restore the usual prompts, use the phrase:

```
...Replace <
```

To use an empty replacement text (replace with nothing), use the following phrase:

```
...Replace "pattern" Newline " " Cdelete Newline...
```

If you find that you write many macros with empty replacements, the common phrase can be placed in a macro, as follows:

```
nothing := " " Cdelete Newline
```

In addition, macro definitions can be more readable with the following definition:

with := Newline

With these definitions, you can write:

... Replace "pattern" with nothing ...

Resize

Key Unassigned

Resize

Enters window-resizing mode. When in window-resizing mode, only the following actions are available:

Action	Key
Shrink one row	UP
Expand one row	DOWN
Shrink one column	LEFT
Expand one column	RIGHT
Accept the new size	ENTER
Cancel the resize	ESC

Arg number Resize (ALT+A *number* Unassigned)

Resizes the window to *number* rows high.

Arg number Meta Resize (ALT+A *number* F9 Unassigned)

Resizes the window to *number* columns wide.

See **Movewindow**

Restcur

Key Unassigned

Restcur

Moves the cursor to the last position saved with the **Savecur** function (Unassigned, Set To Anchor command, Edit menu). **Restcur** always clears the saved position.

Returns True Position restored.
False No saved position to restore.

See Selcur

Right

Keys RIGHT, CTRL+D

Right

Moves the cursor one character to the right. If this action causes the cursor to move out of the window, PWB adjusts the window to the right according to the **Hscroll** switch.

Meta Right (F9 RIGHT)

Moves the cursor to the rightmost position in the window.

Returns True Cursor on text in the line.
False Cursor past text on the line.

Example In a macro, the return value of the **Right** function can be used to test if the cursor is on text in the line or past the end of the line.

The following macro tests the return value to simulate the **Endline** function:

```
MyEndline := Begline :>loop Right +>loop
```

See Begline, Endfile, Endline, Home, Left

Saveall

Key	Unassigned
	Saveall Saves all modified disk files. Pseudofiles are not saved.
Returns	Saveall always returns true.

Savecur

Key	Unassigned
Menu	Edit menu, Set Anchor command
	Savecur Saves the cursor position (sets an anchor). To restore the cursor to the saved position, use the Restcur function (Unassigned). To select text from the current position to the saved position, use the Select To Anchor command from the Edit menu or the Selcur function (Unassigned).
Returns	Savecur always returns true.

Sdelete

Key	Unassigned
	Sdelete Deletes the character at the cursor. Does not copy the character to the clipboard.
	Arg Sdelete (ALT+A Unassigned) Deletes text from the cursor to the end of the line, including the line break. The deleted text is copied to the clipboard.

Arg *streamarg* | *boxarg* | *linearg* **Sdelete**
(ALT+A *streamarg* | *boxarg* | *linearg* Unassigned)

Deletes the selected stream of text from the starting point of the selection to the cursor and copies it to the clipboard. Always deletes a stream, regardless of the current selection mode.

Meta ... Sdelete (F9 ... Unassigned)

As above but discards the deleted text. The contents of the clipboard are unchanged.

Returns **Sdelete** always returns true.

Searchall

Key Unassigned

Searchall

Highlights all occurrences of the previously searched string or pattern. Moves the cursor to the first occurrence in the file.

Arg Searchall (ALT+A Unassigned)

Highlights all occurrences of the string specified by the text from the cursor to the first blank character.

Arg *textarg* **Searchall** (ALT+A *textarg* Unassigned)

Highlights all occurrences of *textarg*.

Arg Arg Searchall (ALT+A ALT+A Unassigned)

Highlights all occurrences of the regular expression defined by the characters from the cursor to the first blank character.

Arg *streamarg* **Searchall** (ALT+A *streamarg* Unassigned)

Highlights all occurrences of *streamarg*.

Arg Arg *textarg* **Searchall** (ALT+A ALT+A *textarg* Unassigned)

Highlights all occurrences of a regular expression defined by *textarg*.

Meta ... Searchall (F9 ... Unassigned)

As above but reverses the value of the **Case** switch for one search.

Returns True String or pattern found.
False No matches found.

Selcur

Key	Unassigned
Menu	Edit menu, Select To Anchor command

Selcur

Selects text from the cursor to the position saved using the Set Anchor command from the Edit menu or the **Savecur** function (Unassigned). If no position has been saved, **Selcur** selects text from the cursor to the beginning of the file.

Returns	Selcur always returns true.
----------------	------------------------------------

Select

Keys	SHIFT+PGUP, SHIFT+CTRL+PGUP, SHIFT+PGDN, SHIFT+CTRL+PGDN, SHIFT+END, SHIFT+CTRL+END, SHIFT+HOME, SHIFT+CTRL+HOME, SHIFT+LEFT, SHIFT+CTRL+LEFT, SHIFT+UP, SHIFT+RIGHT, SHIFT+CTRL+RIGHT, SHIFT+DOWN
-------------	--

Select

Causes a shifted key to take on the cursor-movement function associated with the unshifted key and begins or extends a selection.

To see the key combinations currently assigned to this function, use the Key Assignments command from the Options menu.

Selmode

Key	Unassigned
------------	------------

Selmode

Advances the selection mode between stream, line, and box modes, starting with the current mode.

Returns	True	New mode is stream mode.
	False	New mode is box mode or line mode.
See	_pwbstreammode, _pwbboxmode, _pwblinemode	

Selwindow

Key	F6	
	Selwindow	Moves the focus to the next window.
	Arg Selwindow (ALT+A F6)	Moves the focus to the next unminimized window. Minimized windows (icons) are skipped.
	Arg number Selwindow (ALT+A number F6)	Moves the focus to the specified window.
	Meta Selwindow (F9 F6)	Moves the focus to the previous window.
	Arg Meta Selwindow (ALT+A F9 F6)	Moves the focus to the previous unminimized window.
Returns	True	Focus moved to another window.
	False	No other windows are open.

Setfile

Key	F2	
	Setfile	Switches to the first file in the active window's file history. If there are no files in the file history, PWB displays the message <code>No alternate file</code> . When the Autosave switch is set to yes, PWB saves the current file if it has been modified.
		Setfile does not honor the Newwindow switch. To open a new window when you open a file, use Openfile .

Arg Setfile (ALT+A F2)

Switches to the filename that begins at the cursor and ends with the first blank character.

Arg *textarg* Setfile (ALT+A *textarg* F2)

Switches to the file specified by *textarg*. If the file is not already open, PWB opens it. You can use environment-variable specifiers in the argument.

If the argument is a drive or directory name, PWB changes the current drive or directory to the specified one and displays a message to confirm the change.

See: **Infodialog**.

Arg *!number* Setfile (ALT+A *!number* F2)

If the argument has the form *!number*, PWB switches to the file with that number in the file history. The number can be from 1 to 9, inclusive. See: **_pwbfile**.

Arg *wildcard* Setfile (ALT+A *wildcard* F2)

If the argument is a wildcard, PWB creates a pseudofile containing a list of files that match the pattern. To open a file from this list, position the cursor at the beginning of the name and execute **Arg Openfile** (ALT+A F10) or **Arg Setfile** (ALT+A F2).

Meta ... Setfile (F9 ... F2)

As above but does not save the changes to the current file.

Arg Arg Setfile (ALT+A ALT+A F2)

Saves the current file.

Arg Arg *textarg* Setfile (ALT+A ALT+A *textarg* F2)

Saves the current file under the name specified by *textarg*.

Returns

True File opened successfully.

False No alternate file, the specified file does not exist, and you did not wish to create it; or the current file needs to be saved and cannot be saved.

See

Newfile

Sethelp

Key

SHIFT+CTRL+S

The **Sethelp** function opens and closes single Help files. The **Sethelp** function can also display the current list of open Help files. **Sethelp** affects only the current PWB session.

Arg Sethelp (ALT+A SHIFT+CTRL+S)

Opens the Help file specified by the filename at the cursor.

Arg *streamarg* | *textarg* Sethelp (ALT+A *streamarg* | *textarg* SHIFT+CTRL+S)

Opens the Help file specified by the selected filename.

Meta ... Sethelp (F9 ALT+A SHIFT+CTRL+S)

As above except the specified Help file is closed.

Arg ? Sethelp (ALT+A ? SHIFT+CTRL+S)

Lists all currently open Help files.

Returns

True	Help file opened or closed, or list of Help files displayed.
False	The specified file could not be opened or closed, or the list of files could not be displayed.

See

Helpfiles

Setwindow

Key

CTRL+]

Setwindow

Redisplays the contents of the active window.

Meta Setwindow (F9 CTRL+])

Redisplays the current line.

Arg Setwindow (ALT+A CTRL+])

Adjusts the window so that the cursor position becomes the home position (upper-left corner).

Returns

Setwindow always returns true.

Shell

Key

SHIFT+F9

Shell

Runs an operating-system command shell. To return to PWB, type `exit` at the operating-system prompt.

Warning Do not start terminate-and-stay-resident (TSR) programs in a shell. This causes unpredictable results.

Arg Shell (ALT+A SHIFT+F9)

Runs the text from the cursor to the end of the line as a command to the shell, and returns to PWB.

Arg *boxarg* | *linearg* Shell (ALT+A *boxarg* | *linearg* SHIFT+F9)

Runs each selected line as a separate command to the shell, and returns to PWB.

Arg *textarg* Shell (ALT+A *textarg* SHIFT+F9)

Runs *textarg* as a command to the shell, and returns to PWB.

Meta ... Shell (F9 ... SHIFT+F9)

Runs a shell, ignoring the **Autosave** switch. Modified files are not saved to disk, but they are retained in PWB's virtual memory.

Returns

True	Shell ran successfully.
False	Invalid argument, or error starting the operating-system command processor.

See

Askrtm, Restart, Savescreen

Sinsert

Key

CTRL+J

Sinsert

Inserts a space at the cursor.

Arg Sinsert (ALT+A CTRL+J)

Inserts a line break at the cursor, splitting the line.

Arg *streamarg* | *linearg* | *boxarg* Sinsert

(ALT+A *streamarg* | *linearg* | *boxarg* CTRL+J)

Inserts a stream of blanks between the starting point of the selection and the cursor. The insertion is always a stream, regardless of the current selection mode.

Returns

True	Spaces or line break inserted.
False	Insertion would make a line too long.

Example The following macro inserts a stream of spaces up to the next tab stop, regardless of the current selection mode:

```
InsertTab := Arg Tab Sinsert
```

See **Insert, Linsert**

Tab

Key TAB

Tab

Moves the cursor to the next tab stop. If there are no tab stops to the right of the cursor, the cursor does not move. Tab stops are defined by the **Tabstops** switch.

Returns True Cursor moved.
False Cursor not moved.

Update In PWB 1.x, tab stops appear at fixed intervals. In PWB 2.00, tab stops can be at variable or fixed intervals.

See **Backtab**

Tell

Key CTRL+T

Tell

Displays the message `Press a key to tell` about and waits for a keystroke. After you press a key or combination of keys, **Tell** brings up the Tell dialog box showing the name of the key and its assigned function in TOOLS.INI key-assignment format.

The key-assignment format is:

function:key

If the key is not assigned a function, **Tell** displays `unassigned` for the function name. See: **Unassigned**.

If you press a combination of keys, but **Tell** still shows the `Press a key` prompt (when you press `SCROLL LOCK`, for example), PWB is unable to recognize that combination of keys and you cannot use it as a key assignment.

Arg Tell (ALT+A CTRL+T)

Prompts for a key, then displays the name of the function or macro assigned to the key in one of these formats:

function:key
macroname:=definition

Arg textarg Tell (ALT+A textarg CTRL+T)

Displays the definition of the macro named by *textarg*. If you specify a PWB function, **Tell** displays:

function:function

Meta ... Tell (F9 ... CTRL+T)

As above except **Tell** types the result into the current file rather than displaying it in a dialog box. This is how to discover the definition of any macro, including PWB macros.

Returns

True Assignment displayed or typed.
False No assignment for the key or the specified name.

Update

In PWB 1.x, the prompt and results appear on the status bar; in PWB 2.00, the prompt and results appear in dialog boxes.

Remarks

Meta Tell is a convenient and reliable way of writing a key assignment when you are configuring PWB.

For example, if you want to execute the **Curdate** function (type today's date) when you press the `CTRL`, `SHIFT`, and `D` keys simultaneously, perform the following steps:

1. Go to an empty line in the [PWB] section of `TOOLS.INI`.
2. Execute **Meta Tell** (F9 CTRL+T).

Tell displays the message: `Press a key to tell about.`

3. Press the `D`, `SHIFT`, and `CTRL` keys simultaneously.

If you have not already assigned a function to this combination, **Tell** types:

`unassigned:Shift+Ctrl+D`

4. Select the word `unassigned` and type `curdate`.
5. If you want the assignment to take effect immediately, move the cursor to the line you've just entered and execute the **Assign** function (ALT+=).

You can use **Meta Arg** *textarg* **Tell** to recover the definition of a predefined PWB macro or a macro that you have not saved or entered into a file.

See **Assign, Record**

Unassigned

Keys Assigned to all available keys.

Unassigned

Displays a message for keys that do not have a function assignment.

All unassigned keys are actually assigned the **Unassigned** function. Thus, to remove a function assignment for a key, assign the **Unassigned** function to the key. The **Unassigned** function is not useful in macros.

Returns The **Unassigned** function always returns false.

See **Assign, Tell**

Undo

Keys ALT+BKSP, SHIFT+CTRL+BKSP

Undo

Reverses the last editing operation. The maximum number of times this can be performed for each file is set by the **Undocount** switch.

Meta Undo (F9 ALT+BKSP)

Performs the operation previously reversed with **Undo**. This action is often called “redo.”

Returns True Operation undone or redone.
False Nothing to undo or redo.

See **_pwbundo, _pwbredo, Repeat**

Up

Keys UP, CTRL+E

Up

Moves the cursor up one line. If a selection has been started, it is extended by one line. If this movement results in the cursor moving out of the window, the window is adjusted upward as specified by the **Vscroll** switch.

Meta Up (F9 UP)

Moves the cursor to the top of the window without changing the column position.

Returns True Cursor moved.
False Cursor not moved; the cursor is already at the destination.

See **Down**

Usercmd

Key Unassigned

The **Usercmd** function executes a custom command added to the Run menu by using Customize command from the Run menu or setting the **User** switch.

Arg *number* Usercmd (ALT+A *number* Unassigned)

Executes the given custom Run menu command. The *number* can be in the range 1–9.

Returns True Command exists.
False Command does not exist, or invalid argument.

See `_pwbusern`

See **Assign, Record**

Window

Key Unassigned

Window

Switch to the next window.

Returns	Condition
True	Switched to next window.
False	No next window to switch to: zero or one window open.

Arg [[Arg]] Window (ALT+A [[ALT+A]] Unassigned)

Open a new window.

Returns	Condition
True	Opened a new window.
False	Window not opened.

Meta Window (F9 Unassigned)

Close the active window.

Returns	Condition
True	Window closed.
False	No open window to close.

Meta Arg Window (ALT+A F9 Unassigned)

Switch to the previous window.

Returns	Condition
True	Switched to previous window.
False	No previous window to switch to: zero or one window open.

Update In PWB 1.x, **Arg Window** and **Arg Arg Window** split the window at the cursor. In PWB 2.00, these forms of **Window** open a new window.

See Selwindow, Setwindow

Winstyle

Key CTRL+F6

Winstyle

Advances through the following series of window styles, starting from the current style:

Horizontal Scroll Bar	Vertical Scroll Bar
No	No
No	Yes
Yes	No
Yes	Yes

When the horizontal scroll bar is not shown, a maximized window does not show its bottom border. Similarly, when the vertical scroll bar is not shown, a maximized window does not show its left and right borders. PWB always displays the title bar.

To get the "clean-screen" look, maximize the window and advance the window style until the borders disappear.

Default Set the default window style with the **Defwinstyle** switch.

Returns True Changed window style.
False No windows open.

Update The no-border state in PWB 1.x is not available in PWB 2.00. In PWB 2.00, when a window is maximized and no scroll bars are present, PWB displays the window without borders.

See **Maximize**

7.5 Predefined PWB Macros

PWB predefines a number of macros, most of which correspond to a command in the PWB menus. You can define a shortcut key for a menu command by assigning the key to the corresponding macro. Note that some menu commands such as the Open command from the File menu do not correspond to a macro, and some macros do not correspond to a menu command.

Table 7.12 PWB Macros

Macro	Description	Key
Curfile	Current file's full path	Unassigned
Curfileext	Current file's extension	Unassigned
Curfilenam	Current file's name	Unassigned
_pwbarrange	Arrange command, Window menu	ALT+F5
_pwbboxmode	Box Mode command, Edit menu	Unassigned
_pwbbuild	Build command, Project menu	Unassigned
_pwbcancelbuild	Cancel Build command, Project menu	Unassigned
_pwbcancelprint	Cancel Print command, File menu	Unassigned
_pwbcancelsearch	Cancel Search command, Search menu	Unassigned
_pwbcascade	Cascade command, Window menu	F5
_pwbclear	Delete command, Edit menu	DEL
_pwbclose	Close command, Window menu	CTRL+F4
_pwbcloseall	Close All command, Window menu	Unassigned
_pwbclosefile	Close command, File menu	Unassigned
_pwbcloseproject	Close command, Project menu	Unassigned
_pwbcompile	Compile command, Project menu	Unassigned
_pwbfile <i>n</i>	<i>n file</i> , File menu	Unassigned
_pwbgotomatch	Goto Match command, Search menu	Unassigned
_pwbhelp_again	Next command, Help menu	Unassigned
_pwbhelp_back	Previous Help topic	ALT+F1
_pwbhelp_contents	Contents command, Help menu	SHIFT+F1
_pwbhelp_context	Topic command, Help menu	F1
_pwbhelp_general	Help on Help command, Help menu	Unassigned
_pwbhelp_index	Index command, Help menu	Unassigned
_pwbhelpnl	Display the message: Online Help Not Loaded	F1 when Help extension not loaded
_pwbhelp_searchres	Search Results command, Help menu	Unassigned
_pwblinemode	Line Mode command, Edit menu	Unassigned
_pwblogsearch	Log command, Search menu	Unassigned

Table 7.12 (continued)

Macro	Description	Key
_pwbmaximize	Maximize command, Window menu	CTRL+F10
_pwbminimize	Minimize command, Window menu	CTRL+F9
_pwbmove	Move command, Window menu	CTRL+F7
_pwbnewfile	New command, File menu	Unassigned
_pwbnewwindow	New command, Window menu	Unassigned
_pwbnextfile	Next command, File menu	Unassigned
_pwbnextlogmatch	Next Match command, Search menu	SHIFT+CTRL+F3
_pwbnextmatch	Next Match command, Search menu	Unassigned
_pwbnextmsg	Next Error command, Project menu	SHIFT+F3
_pwbpreviouslogmatch	Previous Match command, Search menu	SHIFT+CTRL+F4
_pwbpreviousmatch	Previous Match command, Search menu	Unassigned
_pwbprevmsg	Previous Error command, Project menu	SHIFT+F4
_pwbprevwindow	Move to previous window	SHIFT+F6
_pwbquit	Exit command, File menu	ALT+F4
_pwbrebuild	Rebuild All command, Project menu	Unassigned
_pwbrecord	Record command, Edit menu	Unassigned
_pwbredo	Redo command, Edit menu	Unassigned
_pwbrepeat	Repeat command, Edit menu	Unassigned
_pwbresize	Resize command, Window menu	CTRL+F8
_pwbrestore	Restore command, Window menu	CTRL+F5
_pwbsaveall	Save All command, File menu	Unassigned
_pwbsavefile	Save command, File menu	SHIFT+F2
_pwbsetmsg	Goto Error command, Project menu	Unassigned
_pwbshell	DOS Shell command, File menu	Unassigned
_pwbstreammode	Stream Mode command, Edit menu	Unassigned
_pwbtile	Tile command, Window menu	SHIFT+F5
_pwbundo	Undo command, Edit menu	Unassigned
_pwbuser <i>n</i>	<i>command n</i> , Run menu	ALT+F <i>n</i>
_pwbviewbuildresults	View build results button	Unassigned
_pwbviewsearchresults	View search results button	Unassigned
_pwbwindow <i>n</i>	<i>n file</i> , Window menu	ALT+ <i>n</i>

PWB continually redefines the following macros to reflect the current file's name:

Macro	Description
Curfile	Full path
Curfileext	File extension
Curfilenam	File base name

PWB uses the following special-purpose macros:

Macro	Description
Autostart	Executed on startup while reading TOOLS.INI
Mgreplist	List of files for logged searches, multifile replace, Mgrep , and Mreplace
Playback	Default name of recorded macros
Restart	(Obsolete)

By default, these macros are undefined.

Autostart

Key Unassigned

The special PWB macro **Autostart** is executed after PWB finishes all initialization at startup. If used, it must be defined in the [PWB] section of TOOLS.INI.

Definition By default, **Autostart** is not defined.

Curfile

Key Unassigned

The **Curfile** macro types the full path of the current file. This macro is redefined each time you switch to a new file.

Definition `curfile := "pathname"`

Example The following macro copies the full path of the current file to the clipboard for later use:

```
Path2clip := Arg Curfile Copy
```

See **Arg, Copy, Curdate, Curday, Curfilenam, Curfileext, Curtime**

Curfileext

Key Unassigned

The **Curfileext** macro types the filename extension of the current file. This macro is redefined each time you switch to a new file.

Definition `curfileext := "extension"`

Example The following macro copies the base name plus the extension of the current file to the clipboard for later use:

```
Filename2clip := Arg Curfilenam Curfileext Copy
```

See **Arg, Copy, Curdate, Curday, Curfile, Curfilenam, Curtime**

Curfilenam

Key Unassigned

The **Curfilenam** macro types the base name of the current file. This macro is redefined each time you switch to a new file.

Definition `curfilenam := "basename"`

Example The following macro copies the base name of the current file to the clipboard for later use:

```
Name2clip := Arg Curfilenam Copy
```

See **Arg, Copy, Curdate, Curday, Curfile, Curfileext, Curtime**

Mgreplist

Key Unassigned

The special PWB macro **Mgreplist** is used by the Find and Replace commands on the Search menu, **Mgrep**, **Mreplace**, and **Mreplaceall** to specify the list of files to search.

When you create a list of files to search using the Files button in either the Find or Replace dialog box, PWB redefines the **Mgreplist** macro with the specified list of files.

To see the current list of files, choose the Files button in the Replace dialog box. You can change the list in this dialog box, and either choose OK to perform the find-and-replace operation, or choose Cancel to cancel the replace and accept the changes to **Mgreplist**.

You can also insert the definition of **Mgreplist** into the current file by using the phrase: **Arg Meta** Mgreplist **Tell** (ALT+A F9 Mgreplist CTRL+T).

You can edit the macro, then redefine it by using the **Assign** function (ALT+=).

Definition Mgreplist:= "*list*"

list Space-separated list of filenames

The filenames can use the operating-system wildcards (* and ?), and can use environment-variable specifiers. Note that backslashes (\) must be doubled in the macro string.

See **Assign, Tell, Mgrep, Mreplace, Mreplaceall**

Restart

Key Unassigned

Update In PWB 1.x, the special PWB macro Restart is executed whenever PWB returns from a shell, build, or other external operation.

In PWB 2.00, the Restart macro is never executed automatically and has no special meaning; it is an ordinary macro.

`_pwbarrange`

Key ALT+F5

Menu Window menu, Arrange command

The `_pwbarrange` macro arranges all unminimized windows on the desktop. The following illustration shows a typical desktop after execution of `_pwbarrange`:

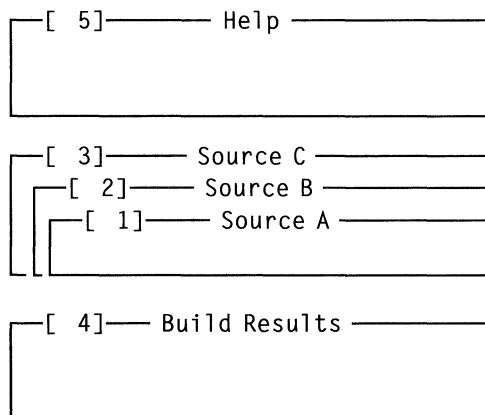


Figure 7.1 Arranged Windows

Definition `_pwbarrange:=cancel arg arrangewindow <`

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Arg Arrangewindow

Arranges all unminimized windows on the desktop.

<

Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See Arrangewindow

_pwbboxmode

Key Unassigned

Menu Edit menu, Box Mode command

The **_pwbboxmode** macro sets the selection mode to box selection mode.

Definition `_pwbboxmode := :>more selmode ->more selmode`

`:>more`

Defines the label `more`.

Selmode

Advances to the next selection mode.

`->more`

Branches to the label `more` if `Selmode` returns false.

The **Selmode** function advances the selection mode from box, to stream, to line.

Selmode returns true when the mode is stream mode. The macro executes the **Selmode** function until it returns true (sets stream mode), then advances the selection mode once to set box selection mode.

See Enterselmode, Selmode

_pwbuild

Key Unassigned

Menu Project menu, Build command

The **_pwbuild** macro builds the “all” target of the current PWB project. The “all” pseudotarget in a PWB project lists all the targets in the project.

For non-PWB projects, **_pwbuild** builds the targets that were last specified by using the Build Target command from the Project menu. PWB redefines **_pwbuild** each time you use Build Target. If no target has been specified, NMAKE builds the first target listed in the project makefile.

Definition `_pwbuild := cancel arg "all" compile <`

Cancel

Establishes a uniform “ground state” by cancelling any selection or argument.

Arg "all" Compile

Builds the `all` pseudotarget in the current project.

`<`

Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Arg, Cancel, Compile**

_pwbcancelbuild

Key Unassigned

Menu Project menu, Cancel Build command

The **_pwbcancelbuild** macro terminates the current background build or compile and flushes any queued build operations.

Definition `_pwbcancelbuild := cancel arg meta compile`

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Arg Meta Compile

Terminates the background build process.

See Arg, Cancel, Compile, Meta

_pwbcancelprint

Key Unassigned

The **_pwbcancelprint** macro terminates all background print operations.

Definition `_pwbcancelprint := cancel arg meta print`

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Arg Meta Print

Terminate background print operations.

See Arg, Cancel, Meta, Print

_pwbcancelsearch

Key Unassigned

Menu Search menu, Cancel Search command

The **_pwbcancelsearch** macro cancels the current background search. PWB performs logged searches in the background under multithreaded environments.

Definition `_pwbcancelsearch := cancel cancelsearch <`

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Cancelsearch

Cancels the current background search.

<
Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

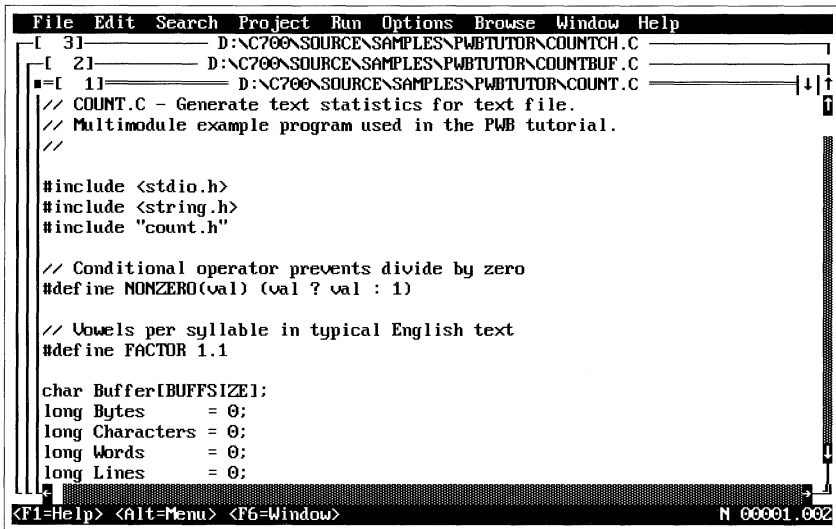
See **Cancel, Cancelsearch, Logsearch**

_pwbcascade

Key F5

Menu Window menu, Cascade command

The **_pwbcascade** macro arranges all unminimized windows in cascaded fashion so that all of their titles are visible. Up to 16 unminimized windows can be cascaded.



```

File Edit Search Project Run Options Browse Window Help
[ 3] D:\C700\SOURCE\SAMPLES\PWBTUTOR\COUNTCH.C
[ 2] D:\C700\SOURCE\SAMPLES\PWBTUTOR\COUNTBUF.C
[ 1] D:\C700\SOURCE\SAMPLES\PWBTUTOR\COUNT.C
// COUNT.C - Generate text statistics for text file.
// Multimodule example program used in the PWB tutorial.
//
#include <stdio.h>
#include <string.h>
#include "count.h"

// Conditional operator prevents divide by zero
#define NONZERO(val) (val ? val : 1)

// Vowels per syllable in typical English text
#define FACTOR 1.1

char Buffer[BUFSIZE];
long Bytes = 0;
long Characters = 0;
long Words = 0;
long Lines = 0;
<F1=Help> <Alt=Menu> <F6=Window> N 00001.002

```

Figure 7.2 Cascaded Windows

Definition **_pwbcascade** := cancel arrangewindow <

Cancel

Establishes a uniform "ground state" by canceling any selection or argument.

Arrangewindow

Cascades all unminimized windows.

<

Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See Arrangewindow, Cancel

_pwbclear

Key DEL

Menu Edit menu, Delete command

The **_pwbclear** macro removes the selected text from the file. If there is no selection, PWB removes the current line.

The selection or line is not copied to the clipboard. It can be recovered only by using the Undo command from the Edit menu or **Undo** (ALT+BKSP).

Definition `_pwbclear := meta delete`

Meta Delete

Removes the selection or the current line from the file without modifying the clipboard.

See Delete, Meta

_pwbcloseall

Key Unassigned

Menu Window menu, Close All command

The **_pwbcloseall** macro closes all open windows.

Definition `_pwbcloseall := cancel arg arg meta window <`

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Arg Arg Meta Window

Closes all windows.

<

Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Arg, Cancel, Meta, Window**

_pwbclosefile

Key Unassigned

Menu File menu, Close command

The **_pwbclosefile** macro closes the current file. If no files remain in the window's file history, the window is closed.

Definition `_pwbclosefile := cancel closefile <`

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Closefile

Closes the current file.

<

Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Cancel, Closefile**

_pwbcloseproject

Key Unassigned

Menu Project menu, Close Project command

The **_pwbcloseproject** macro closes the current project.

Definition `_pwbcloseproject := cancel arg arg project <`

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Arg Arg Project

Closes the current project.

<

Restores the function’s prompt (if any). By default, function prompts are suppressed within a macro.

See **Arg, Cancel, Project**

_pwbcompile

Key Unassigned

Menu Project menu, Compile File command

The **_pwbcompile** macro compiles the current file.

Definition `_pwbcompile := cancel arg compile <`

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Arg Compile

Compiles the current file.

<

Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Arg, Cancel, Compile**

_pwbgotomatch

Key Unassigned

Menu Search menu, Goto Match command

The **_pwbgotomatch** macro sets the match listed at the current location in the Search Results pseudofile as the current match and moves the cursor to the location specified by that match.

Definition `_pwbgotomatch := cancel arg nextsearch <`

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Arg Arg Nextsearch

Goes to the current match.

<

Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Arg, Cancel, Nextsearch**

_pwbhelpnl

The **_pwbhelpnl** macro displays a message indicating the Help extension is not loaded. The Help keys are assigned this macro until the Help extension is loaded.

Definition `_pwbhelpnl := cancel arg "OnLine Help Not Loaded" message`

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Arg "OnLine Help Not Loaded" Message

Displays the message on the status bar.

See **Arg, Cancel, Load, Message**

_pwbhelp_again

Key Unassigned

Menu Help menu, Next command

The **_pwbhelp_again** displays the next occurrence of the last topic for which you requested Help. If no other occurrences of the topic are defined in the open files, PWB redisplay the current topic.

The topic that PWB looks up when you use this command is displayed after the Next command on the Help menu, as follows:

Next: *topic key*

topic Topic string used for the command.

key Current key assignment for **_pwbhelp_again** (if any).

Definition `_pwbhelp_again:=cancel arg pwbhelp.pwbhelpnext <`

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Arg

Sets the **Arg** prefix for the **Pwbhelpnext** function.

Pwbhelp.

Specifies that the function is the PWBHELP extension function.

Pwbhelpnext

Displays the next occurrence of the previously requested topic.

<

Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Pwbhelpnext**

_pwbhelp_back

Key	ALT+F1
	The _pwbhelp_back macro displays the previously viewed Help topic. Up to 20 topics are kept in the Help backtrace list.
Definition	<code>_pwbhelp_back:=cancel meta pwbhelp.pwbhelpnext <</code>
	Cancel Establishes a uniform “ground state” by canceling any selection or argument.
	Meta Sets the meta prefix for the function.
	Pwbhelp. Specifies that the function is the PWBHELP extension function.
	Pwbhelpnext Displays the previously viewed Help topic.
	< Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.
See	Pwbhelpnext

_pwbhelp_contents

Key	SHIFT+F1
Menu	Help menu, Contents command
	The _pwbhelp_contents macro opens the Help window and displays the top-level contents of the Help system.
	Within the Help system, most Help topics have a Contents button at the top of the window. This button also takes you to the top-level contents.
Definition	<code>_pwbhelp_contents:=cancel arg "advisor.hlp!h.contents" pwbhelp.pwbhelp <</code>
	Cancel Establishes a uniform “ground state” by canceling any selection or argument.

Arg "advisor.hlp!h.contents"

Defines a text argument with the topic name for the general table of contents.

Pwbhelp.

Specifies that the function is the PWBHELP extension function.

Pwbhelp

Looks up the topic `h.contents` in the ADVISOR.HLP Help file.

<

Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Pwbhelp**

`_pwbhelp_context`

Key F1

Menu Help menu, Topic command

The **`_pwbhelp_context`** macro looks up Help on the topic at the cursor, the current selection, or the specified text argument.

Definition `_pwbhelp_context:=arg pwbhelp.pwbhelp <`

Arg

Sets the Arg prefix for the **Pwbhelp** function.

Pwbhelp.

Specifies that the function is the PWBHELP extension function.

Pwbhelp

Displays Help on the topic at the cursor. When text is selected, displays Help on the selected text. When you have entered an argument in the Text Argument dialog box, displays Help on the topic specified by the text argument.

<

Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Pwbhelp**

_pwbhelp_general

Key Unassigned

Menu Help menu, Help on Help command

The **_pwbhelp_general** macro opens the Help window and displays information about using the Help system.

Definition `_pwbhelp_general:=cancel arg "advisor.hlp!h.default" pwbhelp.pwbhelp <`

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Arg "advisor.hlp!h.default"

Defines a text argument with the topic name for default Help.

Pwbhelp.

Specifies that the function is the PWBHELP extension function.

Pwbhelp

Looks up the topic “h.default” in the ADVISOR.HLP Help file.

<

Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Pwbhelp**

_pwbhelp_index

Key Unassigned

Menu Help menu, Index command

The **_pwbhelp_index** macro opens the Help window and displays the top-level table of indexes in the Help system.

Within the Help system, most Help topics have an Index button at the top of the window. This button also takes you to the top-level table of indexes.

Definition `_pwbhelp_index:=cancel arg "advisor.hlp!h.index" pwbhelp.pwbhelp <`

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Arg "advisor.hlp!h.index"

Defines a text argument with the topic name for the Help index.

Pwbhelp.

Specifies that the function is the PWBHELP extension function.

Pwbhelp

Looks up the topic “h.index” in the ADVISOR.HLP Help file.

<

Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Pwbhelp**

_pwbhelp_searchres

Key Unassigned

Menu Help menu, Search Results command

The **_pwbhelp_searchres** macro opens the Help window and displays the list of matches found during the last global Help search.

Definition `_pwbhelp_searchres:=cancel pwbhelp.pwbhelpsearch <`

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Pwbhelp.

Specifies that the function is the PWBHELP extension function.

Pwbhelpsearch

Opens the Help window and displays the results of the last global Help search.

<

Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Pwbhelpsearch**

_pwblinemode

Key Unassigned

Menu Edit menu, Line Mode command

The **_pwblinemode** macro sets the selection mode to line selection mode.

Definition `_pwblinemode := :>more selmode ->more selmode selmode`

`:>more`

Defines the label `more`.

Selmode

Advances to the next selection mode.

`->more`

Branches to the label `more` if `Selmode` returns false.

The **Selmode** function advances the selection mode from box, to stream, to line.

Selmode returns true when the mode is stream mode. The macro executes the

Selmode function until it returns true (sets stream mode), then advances the selection mode twice to set line selection mode.

See **Enterselmode**, **Selmode**

_pwblogsearch

Key Unassigned

Menu Search menu, Log command

The **_pwblogsearch** macro toggles search logging on and off.

When search logging is turned on, PWB displays a bullet next to the Log

command on the Search menu. The Next Match command executes the

_pwbnextlogmatch macro, and the Previous Match command executes

the **_pwbpreviouslogmatch** macro. When search logging is turned off, no

bullet appears and the Next Match and Previous Match commands execute

_pwbnextmatch and **_pwbpreviousmatch**.

Definition `_pwblogsearch := cancel logsearch <`

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Logsearch

Toggles the logging of search results on and off.

<

Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Cancel, Logsearch**

_pwbmaximize

Key CTRL+F10

Menu Window menu, Maximize command

The **_pwbmaximize** macro enlarges the active window to its largest possible size, showing only the window, the menu bar, and the status bar on the PWB screen.

Definition **_pwbmaximize := cancel maximize <**

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Maximize

Enlarges the active window to full size.

<

Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Cancel, Minimize**

_pwbminimize

Key CTRL+F9

Menu Window menu, Minimize command

The **_pwbminimize** macro minimizes the active window, reducing the window to an icon. The following illustration shows an open Source window and two icons:

The screenshot shows a window titled "D:\C700\SOURCE\SAMPLES\PWB_TUTOR\COUNT.C". The window contains the following C code:

```

// COUNT.C - Generate text statistics for text file.
// Multimodule example program used in the PWB tutorial.
//
#include <stdio.h>
#include <string.h>
#include "count.h"

// Conditional operator prevents divide by zero
#define NONZERO(val) (val ? val : 1)

// Vowels per syllable in typical English text
#define FACTOR 1.1
  
```

At the bottom of the window, there are two minimized icons labeled "2" and "3", with "COUNTBUF.C" and "COUNTCH.C" written below them respectively. The status bar at the bottom of the window shows "<F1=Help> <Alt=Menu> <F6=Window>" on the left and "N 00001.003" on the right.

To restore a window to its original size, double-click in the box or use the Restore command (CTRL+F5) on the Window menu.

Definition `_pwbminimize := cancel minimize <`

Cancel

Establishes a uniform "ground state" by canceling any selection or argument.

Minimize

Shrinks the window to an icon.

<
Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Cancel, Maximize, Minimize**

_pwbmove

Key CTRL+F7

Menu Window menu, Move command

The **_pwbmove** macro starts window-moving mode for the active window. In window-moving mode, you can only do the following:

Action	Key
Move up one row	UP
Move down one row	DOWN
Move left one column	LEFT
Move right one column	RIGHT
Accept the new position	ENTER
Cancel the move	ESC

To move the window in larger increments, you can use a numeric argument with the **Movewindow** function.

Definition `_pwbmove := cancel movewindow <`

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Movewindow

Starts window-moving mode.

<
Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Arrangewindow, Cancel, Maximize, Minimize, Resize**

_pwbnewfile

Key Unassigned

Menu File menu, New command

The **_pwbnewfile** macro creates a new pseudofile.

New pseudofiles are given a unique name of the form:

<Untitled.*nnn*>Untitled.*nnn*

where <*nnn*> is a three-digit number starting with 001 at the beginning of each PWB session. The window title shows Untitled.*nnn*. The file may be referred to by the name <Untitled.*nnn*>.

When the **Newwindow** switch is set to yes, or the active window is a PWB window, a new window is opened for the file. Otherwise, the file is opened in the active window, and the previous file is placed in the window's file history.

Definition `_pwbnewfile := cancel newfile <`

Cancel

Establishes a uniform "ground state" by canceling any selection or argument.

Newfile

Creates a new untitled pseudofile.

<

Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Cancel, Setfile**

_pwbnewwindow

Key Unassigned

Menu Window menu, New command

The **_pwbnewwindow** macro opens a new window, which shows the current file. The new window has the complete file history as the original window.

Definition `_pwbnewwindow := cancel arg window`

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Arg Window

Opens a new window on the current file

See **Arg, Cancel, Window**

`_pwbnextfile`

Key Unassigned

Menu File menu, Next command

The **`_pwbnextfile`** macro moves to the next file in the list of files specified on the PWB command line. If no more files remain in the list, this macro ends the PWB session.

When the **Newwindow** switch is set to yes, or the active window is a PWB window, a new window is opened for the file. Otherwise, the file is opened in the active window, and the previous file is placed in the window’s file history.

Definition `_pwbnextfile := cancel exit <`

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Exit

Moves to the next file specified on the command line.

<

Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Exit, Askexit, Cancel, PWB Command Line**

_pwbnextlogmatch

Key SHIFT+CTRL+F3

Menu Search menu, Next Match command

The **_pwbnextlogmatch** macro advances the cursor to the next match listed in the Search Results pseudofile.

The Next Match command on the Search menu executes this macro when search logging is turned on. When search logging is turned off, Next Match executes the **_pwbnextmatch** macro.

Definition _pwbnextlogmatch := cancel nextsearch <

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Nextsearch

Advances to the next match in Search Results.

<

Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Cancel, Nextsearch**

_pwbnextmatch

Key Unassigned

Menu Search menu, Next Match command

The **_pwbnextmatch** macro searches forward in the file using the last search pattern and options. The search options are Match Case, Wrap Around, and Regular Expression.

The Next Match command on the Search menu executes this macro when search logging is turned off. When search logging is turned on, Next Match executes the **_pwbnextlogmatch** macro.

Definition _pwbnextmatch := cancel psearch <

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Psearch

Searches forward in the file for the next occurrence of the last search string or pattern.

<

Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Cancel, Psearch**

_pwbnextmsg

Key SHIFT+F3

Menu Project menu, Next Error command

The **_pwbnextmsg** macro moves the cursor to the next message in Build Results.

Definition `_pwbnextmsg := cancel nextmsg <`

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Nextmsg

Goes to the next message in Build Results.

<

Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Cancel, Nextmsg**

_pwbpreviouslogmatch

Key SHIFT+CTRL+F4

Menu Search menu, Previous Match command

The **_pwbpreviouslogmatch** macro moves the cursor to the previous match listed in the Search Results pseudofile.

The Previous Match command on the Search menu executes this macro when search logging is turned on. When search logging is turned off, Previous Match executes the **_pwbpreviousmatch** macro.

Definition `_pwbpreviouslogmatch := cancel arg "-1" nextsearch <`

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Arg "-1" Nextsearch

Moves to the previous match listed in Search Results.

<

Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See Arg, Cancel, Nextsearch

_pwbpreviousmatch

Key Unassigned

Menu Search menu, Previous Match command

The **_pwbpreviousmatch** macro searches backward in the file, using the last search pattern and options. The search options are Match Case, Wrap Around, and Regular Expression.

The Previous Match command on the Search menu executes this macro when search logging is turned off. When search logging is turned on, Previous Match executes the **_pwbpreviouslogmatch** macro.

Definition `_pwbpreviousmatch := cancel msearch <`

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Msearch

Searches backward in the file for the last search string or pattern.

<
Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Cancel, Msearch**

_pwbprevmsg

Key SHIFT+F4

Menu Project menu, Previous Error command

The **_pwbprevmsg** macro moves the cursor to the previous message in the Build Results pseudofile.

Definition `_pwbprevmsg := cancel arg "-1" nextmsg <`

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Arg "-1" Nextmsg

Goes to the previous message in Build Results.

<
Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Arg, Cancel, Nextmsg**

_pwbprevwindow

Key SHIFT+F6

The **_pwbprevwindow** macro moves the focus to the previous window. That is, PWB sets the previously active window as the active window. This action moves among the open windows in the reverse order of **Selwindow** (F6).

Definition `_pwbprevwindow:=cancel meta selwindow <`

Cancel

Establishes a uniform "ground state" by canceling any selection or argument.

Meta Selwindow

Moves the focus to the previous window.

<

Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Cancel, Meta, Selwindow**

_pwbquit

Key ALT+F4

Menu File menu, Exit command

The **_pwbquit** macro leaves PWB immediately. Any specified files on the PWB command line that have not been opened are ignored.

Definition `_pwbquit := cancel arg exit <`

Cancel

Establishes a uniform "ground state" by canceling any selection or argument.

Arg Exit

Leaves PWB.

<

Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Arg, Askexit, Cancel, Exit, Savescreen**

_pwbrebuild

Key Unassigned

Menu Project menu, Rebuild All command

The **_pwbrebuild** macro forces a rebuild of everything in the current project.

For non-PWB projects, **_pwbrebuild** rebuilds the targets that were last specified by using the Build Target command on the Project menu. PWB redefines **_pwbrebuild** each time you use Build Target. If no target has been specified, NMAKE rebuilds the first target listed in the project makefile.

Definition `_pwbrebuild := cancel arg meta "all" compile <`

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Arg Meta "all" Compile

Rebuilds the `all` pseudotarget.

`<`

Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Arg, Cancel, Compile, Meta**

_pwbrecord

Key Unassigned

Menu Edit menu, Record On command

The **_pwbrecord** macro toggles macro recording on and off. If you have not set the recorded macro name and key, PWB displays the Set Macro Record dialog box so you can set them. Execute **_pwbrecord** again to start recording.

Definition `_pwbrecord := cancel record <`

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Record

Toggles macro recording on and off.

<

Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Cancel, Record**

_pwbredo

Key Unassigned

Menu Edit menu, Redo command

The **_pwbredo** macro restores the last modification that was reversed using Edit Undo or **Undo** (ALT+BKSP).

Definition `_pwbredo := cancel meta undo <`

Cancel

Establishes a uniform "ground state" by canceling any selection or argument.

Meta Undo

Restores the last "undone" modification.

<

Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Cancel, Meta, Undo**

_pwbrepeat

Key Unassigned

Menu Edit menu, Repeat command

The **_pwbrepeat** macro repeats the last editing operation once.

Definition `_pwbrepeat := cancel repeat <`

Cancel
Establishes a uniform “ground state” by canceling any selection or argument.

Repeat
Repeats the last operation one time.

`<`
Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Cancel, Repeat**

`_pwbresize`

Key CTRL+F8

Menu Window menu, Size command

The **`_pwbresize`** macro starts window-sizing mode for the active window. When in window-resizing mode, only the following actions are available:

Action	Key
Shrink one row	UP
Expand one row	DOWN
Shrink one column	LEFT
Expand one column	RIGHT
Accept the new size	ENTER
Cancel the resize	ESC

To size the window in larger increments, you can use the numeric forms of the **Resize** function.

Definition `_pwbresize := cancel resize <`

Cancel
Establishes a uniform “ground state” by canceling any selection or argument.

Resize
Starts window-sizing mode.

<
Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Arrangewindow, Cancel, Maximize, Minimize, Movewindow**

_pwbrestore

Key CTRL+F5

Menu Window menu, Restore command

The **_pwbrestore** macro restores the active window to its size before it was maximized or minimized.

Definition **_pwbrestore := cancel meta maximize**

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Meta Maximize

Restores the window to its previous size.

See **Cancel, Maximize, Meta, Minimize**

_pwbsaveall

Key Unassigned

Menu File menu, Save All command

The **_pwbsaveall** macro saves all modified disk files. Modified pseudofiles are not saved.

Definition **_pwbsaveall := cancel saveall <**

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Saveall

Writes all modified files to disk.

<

Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See**Cancel, Saveall**

_pwbsavefile**Key**

SHIFT+F2

Menu

File menu, Save command

The **_pwbsavefile** macro saves the current file to disk.

If the current file is a pseudofile (an untitled file), PWB displays the Save As dialog box so you can give the file a more meaningful name.

Definition**_pwbsavefile** := cancel arg arg setfile <**Cancel**

Establishes a uniform “ground state” by canceling any selection or argument.

Arg Arg Setfile

Writes the current file to disk.

<

Restores the function's prompt (if any). By default, function prompts are suppressed when a macro is running.

See**Arg, Cancel, Setfile**

_pwbsetmsg

Key Unassigned

Menu Project menu, Goto Error command

The **_pwbsetmsg** macro sets the message listed at the current location in the Build Results pseudofile as the current message and moves the cursor to the location specified by that message.

See **Nextmsg**

Definition `_pwbsetmsg := cancel arg arg nextmsg <`

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Arg Arg Nextmsg

Goes to the current message.

<

Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Arg, Cancel, Nextmsg**

_pwbshell

Key Unassigned

Menu File menu, DOS Shell command

The **_pwbshell** macro temporarily leaves PWB, starting a new operating-system shell. To return to PWB, type `exit` at the operating-system prompt.

Definition `_pwbshell := cancel shell <`

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Shell

Starts an operating-system shell.

<
Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See Asktrn, Cancel, Savescreen, Shell

_pwbstreammode

Key Unassigned

Menu Edit menu, Stream Mode command

The **_pwbstreammode** macro sets the selection mode to stream selection mode.

Definition `_pwbstreammode := :>more selmode ->more`

`:>more`
Defines the label `more`.

Selmode

Advances to the next selection mode.

`->more`
Branches to the label `more` if **Selmode** returns false.

The **Selmode** function advances the selection mode from box, to stream, to line. **Selmode** returns true when the mode is stream mode. The macro executes **Selmode** until it returns true (sets stream selection mode).

See Enterselmode, Selmode

_pwbtile

Key SHIFT+F5

Menu Window menu, Tile command

The **_pwbtile** macro tiles all unminimized windows on the desktop so that no windows overlap and the desktop is completely covered. Up to 16 unminimized windows can be tiled.

Definition	<code>_pwbtile := cancel meta arrangewindow <</code> Cancel Establishes a uniform “ground state” by canceling any selection or argument. Meta Arrangewindow Tiles all unminimized windows. < Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.
See	Arrangewindow, Cancel, Meta

_pwbundo

Key	Unassigned
Menu	Edit menu, Undo command The _pwbundo macro reverses the last modification made to the current file. The maximum number of modifications that can be undone for each file is determined by the Undocount switch.
Definition	<code>_pwbundo := cancel undo <</code> Cancel Establishes a uniform “ground state” by canceling any selection or argument. Undo Reverses the last modification. < Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.
See	Cancel, _pwbredo

_pwusern

Macro	Description	Key
<code>_pwuser1</code>	Run custom Run menu command 1	<code>[[ALT+Fn]]</code>
<code>_pwuser2</code>	.	<code>[[ALT+Fn]]</code>
<code>_pwuser3</code>	.	<code>[[ALT+Fn]]</code>
<code>_pwuser4</code>	.	<code>[[ALT+Fn]]</code>
<code>_pwuser5</code>	.	<code>[[ALT+Fn]]</code>
<code>_pwuser6</code>	.	<code>[[ALT+Fn]]</code>
<code>_pwuser7</code>	.	<code>[[ALT+Fn]]</code>
<code>_pwuser8</code>	.	<code>[[ALT+Fn]]</code>
<code>_pwuser9</code>	Run custom Run menu command 9	<code>[[ALT+Fn]]</code>

Menu

Run *command*

command Title of custom Run menu item.

The `_pwusern` macros execute custom commands in the Run menu.

To add a new command to the Run menu, use the Customize Run Menu command or assign a value to the **User** switch.

Definition

`_pwusern := cancel arg "n" usercmd <`

Cancel

Establishes a uniform “ground state” canceling any selection or argument.

Arg "n" Usercmd

Executes custom run menu item number *n*.

<

Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.

Example

`_pwuser1 := cancel arg "1" usercmd <`

This macro executes custom Run menu command number 1.

See

Arg, Cancel, Usercmd

_pwbviewbuildresults

Key Unassigned

Button The View Results button in the Build Operation Complete dialog box.

The **_pwbviewbuildresults** macro opens the Build Results window.

PWB executes this macro when you choose the View Results button in the Build Operation Complete dialog box.

You can redefine this macro to change the behavior of the View Results button. For example, if you want to move to the first message in the log and arrange windows, add `_pwbnextmsg _pwbarrangewindow` to the end of the macro definition.

Definition `_pwbviewbuildresults:=cancel arg "<COMPILE>" pwbwindow`

Cancel

Establishes a uniform "ground state" by canceling any selection or argument.

Arg "<COMPILE>" Pwbwindow

Opens the Build Results window.

See **Pwbwindow**

_pwbviewsearchresults

Key Unassigned

Button The View Results button in the Search Operation Complete dialog box.

The **_pwbviewSearchresults** macro opens the Search Results window.

PWB executes this macro when you choose the View Results button in the Search Operation Complete dialog box.

You can redefine this macro to change the behavior of the View Results button. For example, if you want to move to the first location in the log and arrange windows, add `_pwbnextsearch _pwbarrangewindow` to the end of the macro definition.

Definition `_pwbviewsearchresults:=cancel arg "<SEARCH>" pwbwindow`

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Arg "<SEARCH>" Pwbwindow

Opens the Search Results window.

See Pwbwindow

_pwbwindown

Macro	Description	Key
_pwbwindow1	Switch to window 1	ALT+1
_pwbwindow2	.	ALT+2
_pwbwindow3	.	ALT+3
_pwbwindow4	.	ALT+4
_pwbwindow5	.	ALT+5
_pwbwindow6	.	ALT+6
_pwbwindow7	.	ALT+7
_pwbwindow8	.	ALT+8
_pwbwindow9	Switch to window 9	ALT+9

Menu

Window *n file*

n Window number

file Current file in the window

The **_pwbwindown** macros each set a specific numbered window as the active window.

Definition

_pwbwindown := cancel arg "n" selwindow <

Cancel

Establishes a uniform “ground state” by canceling any selection or argument.

Arg "n" Selwindow

Moves to window number *n*.

<

Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.

Example `_pwbwindow1 := cancel arg "1" selwindow <`

This macro sets window number 1 as the active window.

See **Arg, Cancel, Selwindow**

7.6 PWB Switches

PWB provides the following switches to customize its behavior. You set switches by adding entries to the TOOLS.INI file or by using the Editor Settings, Key Assignments, and Colors commands on the Options menu.

Switch	Description
Askexit	Prompt before leaving PWB
Askrtm	Prompt before returning from a shell
Autoload	Load PWB extensions automatically
Autosave	Save files when switching
Backup	File backup mode
Beep	Issue audible or visible alerts
Build	Rules and definitions for the build process
Case	Make letter case significant in searches
Color	Color of interface elements
Cursormode	Block or underline cursor state
Dblclick	Double-click threshold
Deflang	Default language
Defwinstyle	Default window style
Editreadonly	Allow editing of files marked read-only on disk
Enablealtgr	Enable the ALTGR key on non-US keyboards
Entab	Tab translation mode while editing
Enterinsmode	Enter PWB in insert mode
Enterlogmode	Enter PWB with search logging turned on
Enterselmode	Enter PWB in specified selection mode
Envcursave	Save environment variables for PWB sessions
Envprojsave	Save environment variables for projects
Factor	Auto-repeat factor
Fastfunc	Functions for fast auto-repeat
Filetab	Width of tab characters in the file

Switch	Description
Friction	Delay between repetitions of fast functions
Height	Height of the display
Hike	Window adjustment factor
Hscroll	Horizontal scrolling factor
Infodialog	Set of information dialogs displayed
Keepmem	XMS/EMS memory kept during shell, build, and compile
Lastproject	Set the last project on startup
Load	PWB extension to load
Markfile	Name of the current mark file
Mousemode	Mouse configuration; disabled or swapped buttons
Msgdialog	Display a dialog box for build results
Msgflush	Keep only one set of build results
Newwindow	Create a new window when opening a file
Noise	Line counting interval
Printcmd	Command for printing files
Readonly	Command for saving disk read-only files
Realtabs	Preserve tab characters in the file
Restorelayout	Restore the window layout when a project is set
Rmargin	Right margin for word wrap
Savescreen	Preserve the operating-system screen
Searchdialog	Display a dialog box for search results
Searchflush	Keep only one set of search results
Searchwrap	Make searches wrap around the end of the file
Shortnames	Allow access to loaded files by base name
Softcr	Perform automatic indenting
Tabalign	Align the cursor in tab fields
Tabdisp	Character for displaying tab characters
Tabstops	Variable tab stops
Tilemode	Window tiling style
Timersave	Timer interval for saving files
Tmpsav	Number of files kept in file history
Traildisp	Character for displaying trailing spaces
Traillines	Preserve trailing lines
Traillinesdisp	Character for displaying trailing lines
Trailspace	Preserve trailing spaces
Undelcount	Maximum number of file backups
Undocount	Maximum number of edits per file to undo
Unixre	Use UNIX regular-expression syntax

Switch	Description
User	Custom Run menu item
Vscroll	Vertical scrolling factor
Width	Width of the display
Word	Definition of a word
Wordwrap	Wrap words as they are entered

Extension Switches

The standard PWB extensions define additional switches to control their behavior. You set these switches in tagged sections of TOOLS.INI specific to that extension.

PWB Extension	Description	TOOLS.INI Section Tag
PWBROWSE.MXT	Source Browser	[PWB-PWBROWSE]
PWBC.MXT	C/C++ Language	[PWB-PWBC]
PWBHELP.MXT	Microsoft Advisor Help	[PWB-PWBHELP]

The PWBROWSE switches are described in “Browser Switches” on page 309. The PWBC switches are described in “C and C++ Switches” on page 310. The PWBHELP switches are described in “Help Switches” on page 313.

Filename-Parts Syntax

The filename-parts syntax is used by PWB to pass the name of the current file to external programs or operating-system commands. You use this syntax in the **Printcmd**, **Readonly**, and **User** switches.

Syntax

% %

A literal percent sign (%).

Syntax

%s

The fully qualified path of the current file. If the current file is a pseudofile, **%s** specifies the name of a temporary disk file created for the external command to operate on. The temporary file is destroyed before returning to PWB and is never accessible to the editor.

Syntax

%l [[d]][[p]][[f]][[e]]F

Parts of the current filename. The parts of the name are drive, path, filename, and extension. If the current file is a disk file named:

C:\SCRATCH\TEST.TXT

or the pseudofile:

"<COMPILE>Build Results"

the given syntax yields:

Syntax	Disk File	Pseudofile
%lF	C:\SCRATCH\TEST.TXT	<COMPILE>
%ldF	C:	
%lpF	\SCRATCH	
%lFF	TEST	<COMPILE>
%leF	.TXT	
%lpfF	\SCRATCH\TEST	<COMPILE>
%s	C:\SCRATCH\TEST.TXT	C:\TMP\PWB00031.R00
% %	%	%

The title of a pseudofile cannot be specified with the filename-parts syntax, but it is accessible to macros by using the **Curfile** predefined macro.

Warning The %lF syntax always specifies the name of the current file in the active window. For some commands, such as the command specified in the **Readonly** switch, this may not be the desired file. Use %s for the **Readonly** switch.

See **Printcmd, Readonly, User**

Boolean Switch Syntax

You can use either one of the following syntaxes to set Boolean switches in PWB:

Syntax 1 *switch* : [[**yes** | **no** | **on** | **off** | **1** | **0**]]

switch

The name of a PWB switch.

yes, on, 1

Enable the feature controlled by *switch*.

no, off, 0

Disable the feature controlled by *switch*.

Syntax 2 [[**no**]]*switch* :

switch

Enable the feature controlled by *switch*.

no*switch*

Disable the feature controlled by *switch*.

Askexit

Type	Boolean
	The Askexit switch determines if PWB prompts for confirmation before returning to the operating system.
Syntax	Askexit: { yes no }
	yes Prompt for confirmation before leaving PWB.
	no Do not prompt before leaving PWB.
Default	Askexit:no
See	Exit

Askrtm

Type	Boolean
	The Askrtm switch determines if PWB prompts before returning to PWB after running a shell or external command.
Syntax	Askrtm: { yes no }
	yes Prompt for confirmation before returning to PWB. This setting allows you to review the contents of the operating-system screen before returning to the editor.
	no Do not prompt before returning to PWB.
Default	Askrtm:yes
See	Shell

Autoload

Type Boolean

The **Autoload** switch determines if PWB automatically loads its extensions on startup.

When the **Autoload** switch is *yes*, PWB automatically loads extensions whose names begin with "PWB" and are found in the same directory as PWB.EXE. PWB always loads extensions named in a **Load** switch.

If you disable automatic extension loading, you can load extensions as you need them by assigning a value to the Load switch as follows:

Arg load: *pwextension* Assign (ALT+A load:*pwextension* ALT+=).

The *pwextension* is the path of the extension's executable file. PWB automatically assumes the filename extension .MXT. You can specify an environment-variable path by using an environment-variable specifier.

Syntax **Autoload:**{ *yes* | *no* }

yes

Automatically load PWB extensions on startup.

no

Do not automatically load PWB extensions on startup. Load only those extensions named in **Load** switches in TOOLS.INI.

Default Autoload:yes

Update PWB 1.x extensions are not compatible with PWB 2.0. They are refused when you request that they be loaded. Old extensions must be recompiled with the new extension-support libraries and header files. In some cases, old extensions must also be modified for use with PWB 2.00.

Updated Microsoft PWB 1.x extensions not included with this product are available by contacting Microsoft Product Support Services in the United States or your local Microsoft subsidiary.

Autosave

Type Boolean

The **Autosave** switch determines if PWB automatically saves the current file without prompting whenever you move to another file, exit PWB, or execute an external operation such as a shell, build, or compile.

When the **Autosave** switch is set to no, PWB maintains the contents of files in memory for internal operations, and prompts to save modified files on exit or for external operations such as a build. With this setting, PWB never saves a file unless you explicitly save it.

Syntax **Autosave:**{ **yes** | **no** }

yes Automatically save files.

no Do not automatically save files.

Default Autosave:no

Update In PWB 1.x, the default value of **Autosave** is yes.

See **Shell**, **Timersave**

Backup

Type Text

The **Backup** switch determines what happens to the old copy of a file before the new version is saved to disk.

Syntax **Backup:**[[*undel* | *bak*]]

(none)

No backup: PWB overtypes the file.

undel

PWB moves the old file to a hidden directory so you can retrieve it with the UNDEL utility. The number of copies saved is specified by the **Undelcount** switch.

bak

The extension of the previous version of the file is changed to .BAK.

Default Backup:bak

Beep

Type Boolean

The **Beep** switch determines PWB's alerting method. When set to yes, PWB issues an audible sound. When no, PWB flashes the menu bar—a visual “beep.”

Syntax **Beep:**{ yes | no }

yes Generate an audible beep.

no Flash the menu bar.

Default Beep:yes

Case

Type Boolean

The **Case** switch determines if letter case is distinguished in searches.

The search functions that use the **Case** switch have “meta” forms that temporarily reverse the sense of the **Case** switch.

The **Unixre** and **Case** switches have no effect on the syntax of regular expressions used by the **Build** or **Word** switches. These switches always use case-sensitive UNIX regular expressions.

Syntax **Case:**{ yes | no }

yes

Case is significant in searches. Uppercase letters in search patterns do not match lowercase letters in text.

no

Case is not significant in searches. Uppercase letters match lowercase letters.

Default

Case:no

See

Meta, Mgrep, Mreplace, Msearch, Psearch, Replace.

Color

Type

Text

The **Color** switch specifies color of various parts of the PWB display.

Syntax

Color:*name value*

name

Identifies the part of PWB affected by the color value.

value

Two hexadecimal digits specifying the foreground and background color of the indicated item.

Color Names

PWB uses the following color names and default color values for the various parts of the PWB display:

Table 7.13 PWB Color Names

Name	Default Value	Description
Alert	70	Message box
Background	07	(Not visible)
Border	07	Window borders
Builderr	40	Build message line in active window
Buttondown	07	Button while it is down
Desktop	80	Desktop
Dialogaccel	7F	Dialog box accelerator
Dialogaccelbor	7F	Dialog box accelerator border
Dialogbox	70	Dialog box
Disabled	78	Disabled items in menus and dialogs
Elevator	07	Scroll box

Table 7.13 (continued)

Name	Default Value	Description
Enabled	70	Available items in menus and dialogs
Greyed	78	(Not visible)
Helpbold*	8F	Bold Help text
Helpitalic*	8A	Italic Help text and the characters
Helpnorm*	87	Plain Help text
Helpunderline*	8C	Emphasized Help text
Helpwarning*	70	Current hyperlink
Highlight	1F	Highlighted text; text found by searches
Hilitectrl	07	Highlighted control item
Info	3F	Special information
Itemhilitesel	0F	Highlighted character in selected item
Listbox	70	List box within a dialog box
Location	70	Location indicator in status bar
Menu	70	Menu bar
Menubox	70	Menu
Menuhilite	7F	Highlighted character in menu
Menuhilitesel	0F	Highlighted character in selected menu
Menuselected	07	Selected menu
Message	70	Message area of status bar
Pushbutton	70	Button that is not pressed
Pwbwindowborder	07	PWB window borders
Pwbwindowtext	87	PWB window text
Scratch	07	(Not visible)
Scrollbar	70	Gray area and arrows in scroll bar
Selection	71	Current selection
Shadow	08	Shadows
Status	7F	Indicator area of status bar
Text	17	Text in a window

* Defined by the Help extension. Define these settings in the [PWB-PWBHELP] section of TOOLS.INI.

Color Values

Color values for the **Color** switch are two hexadecimal digits that specify the color of the item. The first digit specifies the background color and the second digit specifies the foreground color, according to the following table:

Table 7.14 PWB Color Values

Color	Digit	Color	Digit
Black	0	Dark Gray	8
Blue	1	Bright Blue	9
Green	2	Bright Green	A
Cyan	3	Bright Cyan	B
Red	4	Bright Red	C
Magenta	5	Bright Magenta	D
Brown	6	Yellow	E
White	7	Bright White	F

For example, a setting of 3E displays a cyan background (3) and a yellow foreground (E).

Note that only color displays support all the colors listed. If you have a monochrome adapter or monochrome monitor, the only available colors are black (0), white (7), and bright white (F). All other colors are displayed as white.

Cursormode

Type Numeric

The **Cursormode** switch determines the shape of the cursor when PWB is in insert and overwrite mode, according to the following table:

Cursormode Value	Insert Mode Cursor	Overtype Mode Cursor
0	Underscore	Underscore
1	Block	Block
2	Block	Underscore
3	Underscore	Block

Default Cursormode:2

See **Status Bar**

Dbclick

Type Numeric

The **Dbclick** switch sets the double-click threshold for the mouse (the maximum time between successive clicks of the mouse button). The units for the **Dbclick** switch are 1/18 of a second.

Default Dbclick:10

See Mousemode

Deflang

Type Text

The **Deflang** switch determines the default file extension for file lists in PWB dialog boxes.

Syntax **Deflang:***language*

language One of the following settings:

Setting	Extension
NONE	*
Asm	.ASM
Basic	.BAS
C	.C
C++	.CPP
CXX	.CXX
COBOL	.CBL
FORTRAN	.FOR
LISP	.LSP
Pascal	.PAS

Default Deflang:NONE

Defwinstyle

Type Numeric

The **Defwinstyle** switch sets the default window style. The possible values for **Defwinstyle** are:

Value	Style
1	No scroll bars
3	Vertical scroll bar
5	Horizontal scroll bar
7	Both scroll bars

You can change the active window style by using the **Winstyle** function (CTRL+F6).

Default Defwinstyle:7

See **Maximize**

Editreadonly

Type Boolean

The **Editreadonly** switch determines if PWB allows you to edit a file marked read-only on disk.

Syntax **Editreadonly**:{ **yes** | **no** }

yes

Allow modification of files that are marked read-only on disk. When PWB attempts to save the modified file, PWB informs you that the file is marked read-only. It also prompts you to confirm that the command specified by the **Readonly** switch is to be run. If you decline to run the command, PWB gives you the opportunity to save the file with a different name.

no

Disallow modification of read-only files. For files that cannot be modified, PWB displays the letter R on the status bar. You can reenable modification of a read-only file by using the Read Only command on the Edit menu or the **Noedit** function.

Default Editreadonly:yes

Enablealtgr

Type Boolean

The **Enablealtgr** switch determines if PWB recognizes the ALTGR key (the right ALT key) on international keyboards as ALTGR (Graphic Alt) or ALT.

When ALTGR is enabled, pressing ALTGR+*key* produces the corresponding graphic character. ALTGR is never recognized as a key name for use in PWB key assignments.

Syntax **Enablealtgr**:{ yes | no }

yes Recognize the right ALT key as ALTGR.

no Recognize the right ALT key as ALT.

Default Enablealtgr:no

Entab

Type Numeric

The **Entab** switch controls how PWB converts white space on modified lines. PWB converts white space only on the lines that you modify.

When the **Realtabs** switch is set to yes, tab characters are converted. When set to no, tab characters are not converted.

The **Entab** switch can have the following values:

Value	Meaning
0	Convert all white space to space (ASCII 32) characters.
1	Convert white space outside quoted strings to tabs. A quoted string is any span of characters enclosed by a pair of single quotation marks or a pair of double quotation marks. PWB does not recognize escape sequences because they are language-specific. For well-behaved conversions with this setting, make sure that you use a numeric escape sequence to encode quotation marks in strings or character literals.
2	Convert white space to tabs.

With settings 1 and 2, if the white space being considered for conversion to a tab character occupies an entire tab field or ends at the boundary of a tab field, it is converted to a tab (ASCII 9) character. The width of a tab field is specified by the **Filetab** switch.

In all conversions, PWB maintains the text alignment as it is displayed on screen.

Default Entab:1

See **Filetab, Realtabs, Tabalign**

Enterinsmode

Type Boolean

The **Enterinsmode** switch determines if PWB is to start in insert mode or overtype mode. You can toggle the current mode by using the **Insertmode** function (INS).

When the current mode is overtype mode, the letter O appears on the status bar. Depending on the setting of the **Cursormode** switch, the shape of the cursor reflects the current mode.

Syntax	Enterinsmode: { yes no } yes Start PWB in insert mode. no Start PWB in overtype mode.
Default	Enterinsmode:yes

Enterlogmode

Type	Boolean The Enterlogmode switch determines if search logging is turned on or off when PWB starts up. The current search-logging mode can be changed at any time using the Log command on the Search menu or the Logsearch function (Unassigned).
Syntax	Enterlogmode: { yes no } yes Start PWB with search logging on. no Start PWB with search logging off.
Default	Enterlogmode:no

Enterselmode

Type	Text The Enterselmode switch determines the selection mode when PWB starts up.
Syntax	Enterselmode: { stream box line } stream Starts PWB in stream selection mode. box Starts PWB in box selection mode. line Starts PWB in line selection mode.

Default	Enterselmode:stream
See	Selmode

Envcursave

Type Boolean

The **Envcursave** switch determines if PWB saves and restores the current environment table for PWB sessions.

You can change environment variables by using the Environment command on the Options menu or the **Environment** function (Unassigned).

If you always want to use the operating-system environment, set both **Envcursave** and **Envprojsave** to no.

Syntax **Envcursave**:{ **yes** | **no** }

yes

Save and restore environment variables for PWB sessions. Use this setting if you want to use an environment that is specific to PWB. The PWB environment overrides the operating-system environment.

no

Do not save environment variables between PWB sessions.

Default Envcursave:no

Update:

In PWB 1.x, the INCLUDE, LIB, and HELPFILES environment variables were always saved for PWB sessions and projects.

Envprojsave

Type Boolean

The **Envprojsave** switch determines if PWB saves and restores the environment table for each project. A project's environment overrides both the PWB environment and the external (operating-system) environment.

If you always want to use the operating-system environment table, set both **Envcursave** and **Envprojsave** to no. You can change environment variables by using the Environment command on the Options menu or the **Environment** function (Unassigned).

Syntax **Envprojsave:**{ yes | no }

yes

Save environment variables for the project. Use this setting if you want to set project-specific environments.

no

Do not save environment variables for the project.

Default Envprojsave:yes

Update In PWB 1.x, the INCLUDE, LIB, and HELPFILES environment variables were always saved for PWB sessions and projects.

Factor

Type Text

The **Factor** switch, together with the **Friction** switch, controls how quickly PWB executes a fast function. A fast function is a PWB function whose action repeats as rapidly as possible while you hold down the associated keystroke.

Syntax **Factor:**{ *%percent* | *-constant* } [*count*]

percent

Percentage between 0 and 100 to reduce friction.

constant

Constant value between 0 and 65,535 to reduce friction.

count

Interval between reductions of friction.

PWB reduces friction by *percent* percent or *constant* every *count* repetition of a keystroke, until friction is zero.

Default Factor:%50 10

Example If you hold down the RIGHT ARROW key with the settings:

```
Right :RIGHT
Fastfunc:Right
Friction:1000
Factor :%75 7
```

PWB moves the cursor at the current speed until it has moved seven characters to the right. Then PWB changes the friction to 250 (75 percent reduction of the initial friction of 1000). When the cursor has moved 14 characters, the friction changes to 188 (75 percent reduction of the friction of 250). The cursor moves faster the longer you hold down the RIGHT ARROW key.

See **Fastfunc**

Fastfunc

Type Text

The **Fastfunc** switch specifies functions whose action is rapidly repeated by PWB as you hold down the associated key combination.

The **Friction** and **Factor** switches control the repeat speed and acceleration of fast functions.

Syntax **Fastfunc:***function* {**on** | **off**}

function PWB function to repeat.

on Enable fast repeat for *function*.

off Disable fast repeat for *function*.

Default Fastfunc:Down on
Fastfunc:Left on
Fastfunc:Mlines on
Fastfunc:Mpage on
Fastfunc:Mpara on
Fastfunc:Mword on
Fastfunc:Plines on
Fastfunc:Ppage on
Fastfunc:Ppara on
Fastfunc:Pword on
Fastfunc:Right on
Fastfunc:Up on

Filetab

Type Numeric

The **Filetab** switch determines the width of a tab field for displaying tab (ASCII 9) characters in the file. The width of a tab field determines how white space is translated when the **Realtabs** switch is set to no. The **Filetab** switch does not affect the cursor-movement functions **Tab** (TAB) and **Backtab** (SHIFT+TAB).

Default Filetab:8

See **Entab, Realtabs, Tabdisp**

Friction

Type Numeric

The **Friction** switch, together with the **Factor** switch, controls how quickly PWB executes a fast function. A fast function is a PWB function whose action repeats rapidly when you hold down the associated key.

The value of the **Friction** switch is a decimal number between 0 and 65,535 and specifies the delay between repetitions of a fast function. As the function is repeated, the delay is reduced according to the setting of the **Factor** switch.

Default Friction:40

See **Factor, Fastfunc**

Height

Type Numeric

The **Height** switch determines the number of lines on the PWB screen. The **Height** switch can have one of these values: 25, 43, 50, or 60. The last setting of this switch is saved and restored across PWB sessions and for each project.

Default Height: *first screen height*

When you start PWB for the first time, PWB uses the current screen height. Thereafter, PWB restores the previous setting until you explicitly assign a new value to the **Height** switch.

Note that when you change the setting for **Height** in the Editor Settings dialog box, the change does not take effect until you choose OK. Other switches takes effect immediately when you choose Set Switch.

See **Assign**

Hike

Type Numeric

The **Hike** switch determines the number of lines from the cursor to the top of the window after you move the cursor out of the window by more than the number of lines specified by the **Vscroll** switch.

The minimum value is 1. When the window occupies less than the full screen, the value is reduced in proportion to the window size.

Default Hike:4

See **Hscroll**

Hscroll

Type Numeric

The **Hscroll** switch controls the number of columns that PWB scrolls the text left or right when you move the cursor out of the window. When the window does not occupy the full screen, the amount scrolled is in proportion to the window size.

Text is never scrolled in increments greater than the size of the window.

Default Hscroll:10

See **Vscroll**

Infodialog

Type numeric

The **Infodialog** switch determines which information dialog boxes are displayed.

Syntax **Infodialog:hh**

hh

Two hexadecimal digits specifying a set of flags to indicate which information dialog boxes should be displayed. When a bit is on (1), the corresponding dialog box is displayed. When a bit is off (0), the corresponding dialog box is not displayed.

To set the value of **Infodialog**, add up the hexadecimal numbers listed in the table below for the dialog boxes you want to display.

Value	Information Dialog
01	<i>n</i> occurrences found <i>n</i> occurrences replaced
02	End of Build Results End of Search Results
04	' <i>pattern</i> ' not found
08	No unbalanced characters found
10	Changed directory to <i>directory</i> Changed drive to <i>drive</i>

Default Infodialog:0F

The default value of **Infodialog** tells PWB to display all information dialog boxes except for the “Changed...” dialog boxes.

Keepmem

Type numeric

The **Keepmem** switch specifies the amount of extended (XMS) memory or expanded (EMS) memory kept by PWB during a shell, compile, build, or other external command. Specify the value in units of kilobytes (1024 bytes).

A larger number means that shelling is faster and leaves less memory for tools that use extended or expanded memory. A smaller number means that shelling is slower and leaves more memory for tools. If the number you specify is not large enough, PWB uses no extended or expanded memory.

Default Keepmem:2048

Lastproject

Type Boolean

The **Lastproject** switch determines if PWB automatically opens the last project on startup. The /PN, /PP, /PL, and /PF command-line options override the setting of the **Lastproject** switch.

Syntax	Lastproject: { yes no } yes On startup, open the last project that was open. no Do not open the last project on startup.
Default	Lastproject:no
See	Project

Load

Type	Text The Load switch specifies the filename of a PWB extension to load. When this switch is assigned a value, PWB loads the specified extension. The initialization specified in the extension is performed, and the functions and switches defined by the extension become available in PWB. The extension can be loaded during initialization of a TOOLS.INI section. You can also interactively load an extension by using the Editor Settings command on the Options menu or by using the Assign function to assign a value to the Load switch.
Syntax	Load: [[<i>path</i>]] <i>basename</i> [[. <i>ext</i>]] <i>path</i> Can be a path or an environment-variable specifier. <i>basename</i> Base name of the extension executable file. <i>ext</i> Normally you do not specify a filename extension.
See	Autoload

Markfile

Type Text

The **Markfile** switch specifies the name of the file PWB uses to save marks.

When no mark file is open, marks are kept in memory, and they are lost when you exit PWB. When you open a mark file, marks in memory are saved in the mark file, unless a mark file is already open. When a mark file is already open, the marks in memory are saved in the open file.

To open a mark file, use the Set Mark File command on the Search menu or assign a value to the **Markfile** switch by using the Editor Settings command on the Options menu or the **Assign** function. To close a mark file without opening a new one, assign an empty value to the **Markfile** switch. That is, use the setting:

Markfile:

To set a permanent mark file that is used for every PWB session, place a **Markfile** definition in the [PWB] section of TOOLS.INI.

Syntax **Markfile:** *filename*

filename The name of the file containing mark definitions.

Default Markfile:

The Markfile switch has no default value and is initially undefined.

See **Assign, Mark**

Mark File Format

A mark file is a text file containing mark definitions of the form:

markname filename line column

The mark *markname* is defined as the location given by *line* and *column* in the file *filename*. The *markname* cannot contain spaces and cannot be a number.

Update With PWB 1.x, when you open a mark file and no mark file is currently open, the marks in memory are lost. With PWB 2.00, the marks are saved in the new mark file.

Mousemode

Type Numeric

The **Mousemode** switch enables or disables the mouse and sets the actions of the left and right mouse buttons.

Value	Description
0	The mouse is disabled and the mouse pointer is not visible.
1	Normal mouse control.
2	Exchanges the actions of the left and right mouse buttons.

Default Mousemode:1

See **Dbclick**

Msgdialog

Type Boolean

The **Msgdialog** switch determines if PWB brings up a dialog box summarizing build results or only beeps when a build is complete.

Syntax **Msgdialog**:{ **yes** | **no** }

yes Display a dialog box summarizing build results when a build is complete.
no Beep when a build is complete.

Default Msgdialog:yes

See **Beep**, **Compile**, **Searchdialog**

Msgflush

Type	Boolean
	The Msgflush switch determines if previous build results are retained in the Build Results window or flushed when a new build is started.
Syntax	Msgflush :{ yes no }
	yes Flush previous build results when a new build is started.
	no Save previous build results.
Default	Msgflush:yes
See	Nextmsg , Searchflush

Newwindow

Type	Boolean
	The Newwindow switch determines if certain PWB functions open a file in a new window or in the active window. The Newwindow switch provides the default state of the New Window check box in the Open File dialog box. This check box does not change the value of the Newwindow switch.
	When Newwindow is set to yes, PWB behaves like a Multiple Document Interface (MDI) application. That is, when you open a new file, PWB opens a new window for the file, except in certain situations as noted below.
	When Newwindow is set to no, PWB behaves like PWB 1.x. In this case, PWB opens files into the active window, creating a file history for that window. This mode is useful when working with large numbers of files.
	Some functions use the Newwindow switch to determine if a new window is created when opening a file.
	The following functions ignore the Newwindow switch, and either create a new window or open the file into the active window:

Function	Creates a New Window
----------	----------------------

Mreplace	No
Openfile	Yes
Setfile	No
Nextmsg	No
Nextsearch	No

When the active window is a PWB window, PWB always creates a new window. You cannot open a file into a PWB window.

Syntax**Newwindow:**{ yes | no }**yes**

Open a new window when a new file is opened. This setting makes PWB behave like other MDI applications such as Microsoft Word 5.5 and Microsoft Works.

no

Open files into the active window, adding the previous file to the window's file history. This setting makes PWB behave like PWB 1.x.

Default

Newwindow:yes

SeeExit, Mark, Mreplace, Newfile, Nextmsg, Nextsearch, Openfile, Setfile

Noise

Type

Numeric

The **Noise** switch specifies the number of lines counted at a time as PWB traverses a file while reading, writing, or searching. PWB displays the line counter on the right side of the status bar, in the area which usually shows the current line.

Set **Noise** to 0 to turn off the display of scanned lines.

Default

Noise:50

Printcmd

Type	Text
	The Printcmd switch specifies a program or operating system command that PWB starts when you choose the Print command from the File menu or execute the Print function (Unassigned).
Syntax	Printcmd: <i>command_line</i> <i>command_line</i> An operating-system command line. To pass the filename of the current file, specify %s in the command line. Specify %% to pass a literal percent sign. You can extract parts of the full filename using a special PWB syntax. See "Filename-Parts Syntax" on page 265.
Default	Printcmd:COPY %s PRN
See	Print

ReadOnly

Type	Text
	The ReadOnly switch specifies the operating-system command invoked when PWB attempts to write to a read-only file. When PWB attempts to overwrite a file that is marked read-only on disk, PWB informs you that the file is read-only. It also prompts you to confirm that the command specified in the ReadOnly switch is to be run. If you decline to run the ReadOnly command, PWB gives you the opportunity to save the file with a different name.
Syntax	ReadOnly: [[<i>command</i>]] <i>command</i> Operating-system command line. If no command is specified, PWB prompts you to enter a new filename to save the file. To pass the filename of the current file to the command, specify %s in the command line. Specify %% to pass a literal percent sign. You can extract parts of the full path using a special PWB syntax. See "Filename-Parts Syntax" on page 265.

Note that only `%s` is guaranteed to give the name of the read-only file. The `%IF` syntax gives the current filename (the file displayed in the active window), even when PWB is saving a different file.

Default

Readonly:

The default value specifies that PWB should run no command and should prompt for a different filename.

Example

The **Readonly** switch setting

```
Readonly:Attrib -r %s
```

removes the read-only attribute from the file on disk so PWB can overwrite it.

See

Editreadonly, Noedit

Realtabs

Type

Boolean

The **Realtabs** switch determines if PWB preserves tab (ASCII 9) characters or translates white space according to the **Entab** switch when a line is modified. **Realtabs** also determines if the **Tabalign** switch is in effect.

Syntax

Realtabs:{ **yes** | **no** }

yes Preserve tab characters when editing a line.

no Translate tab characters when editing a line.

Default

Realtabs:yes

See

Entab, Filetab, Tabalign

Restorelayout

Type	Boolean
	<p>The Restorelayout switch determines if PWB restores the saved window layout and file history from the project status file when you open a project or retains the active window layout and file history.</p> <p>This switch provides the default state of the Restore Window Layout check box in the Open Project dialog box.</p>
Syntax	Restorelayout: { yes no }
	<p>yes Restore a project's saved window layout and file history when the project is opened.</p> <p>no Do not restore the project's windows and file history.</p>
Default	Restorelayout:yes
See	Project

Rmargin

Type	Numeric
	<p>The Rmargin switch sets the right margin for word wrapping. It has an effect only when word wrapping is turned on.</p>
Default	Rmargin:78
Update	<p>In PWB 1.x, Rmargin sets the beginning of a six-character "probation" zone where typing a space wraps the line. After the zone, typing any character wraps the current word. This behavior is similar to that of a typewriter. PWB 2.00 uses a word-processor's style of wrapping.</p>

To maintain the same margins as PWB 1.x, increase your **Rmargin** settings by 6.

See **Softcr, Wordwrap**

Savescreen

Type Boolean

The **Savescreen** switch determines if PWB preserves the operating-system screen image and video mode.

Syntax **Savescreen:**{ yes | no }

yes

Save the operating-system screen when starting PWB, and restore it when leaving PWB.

no

Do not preserve the operating-system screen. When you leave PWB, the operating-system screen is blank, and the video mode is left in PWB's last video mode.

Default Savescreen:yes

Searchdialog

Type Boolean

The **Searchdialog** switch determines if PWB brings up a dialog box that summarizes logged search results or only beeps when a logged search is complete. The **Searchdialog** switch has an effect only while logging search results.

Syntax **Searchdialog:**{ yes | no }

yes

Display a dialog box summarizing search results when a logged search is complete.

no

Beep when a logged search is complete.

Default	Searchdialog:yes
See	Beep, Enterlogmode, Logsearch, Msgdialog

Searchflush

Type	Boolean
	The Searchflush switch determines if previous logged search results are flushed or retained when you start a new logged search.
	This switch has an effect only when PWB performs a logged search.
Syntax	Searchflush: { yes no }
	yes Flush the previous search results from the Search Results window when a new search is begun.
	no Preserve previous search results in the Search Results window.
Default	Searchflush:yes
See	Logsearch, Mgrep

Searchwrap

Type	Boolean
	The Searchwrap switch determines if search commands and replace commands wrap around the ends of a file.
Syntax	Searchwrap: { yes no }
	yes Searches wrap around the beginning and end of the file.
	no Searches stop at the beginning and end of the file.

Default Searchwrap:no
See Msearch, Psearch, Replace.

Shortnames

Type Boolean

The **Shortnames** switch determines if currently loaded files can be accessed by their short names (base name only).

Syntax **Shortnames:**{ yes | no }

yes
You can switch to a file currently loaded into PWB by specifying only the base name to the **Setfile** (F2) or **Openfile** (F10) functions.

no
You must specify the extension as well as the base name to switch to a file.

Default Shortnames:yes

See **Openfile, Setfile**

Softcr

Type Boolean

The **Softcr** switch controls indentation of new lines based on the format of surrounding text when you execute the **Emacsnewl** (ENTER) and **Newline** (SHIFT+ENTER) functions.

Syntax **Softcr:**{ yes | no }

yes
Indent new lines.

no

Do not indent new lines. After executing **Emacsnewl** or **Newline**, the cursor is placed in column 1.

Default

Softcr:yes

Tabalign

Type

Boolean

The **Tabalign** switch determines the positioning of the cursor when it enters a tab field. A tab field is the area of the screen representing a tab character (ASCII 9) in the file. The width of a tab field is specified by the **Filetab** switch.

The **Tabalign** switch takes effect only when the **Realtabs** switch is set to yes.

Syntax

Tabalign:{ yes | no }

yes

PWB aligns the cursor to the beginning of the tab field when the cursor enters the tab field. The cursor is placed on the actual tab character in the file.

no

PWB does not align the cursor within the tab field.

You can place the cursor on any column in the tab field. When you type a character at this position, PWB inserts enough leading blanks to ensure that the character appears in the same column.

Default

Tabalign:no

Tabdisp

Type

Numeric

The **Tabdisp** switch specifies the decimal ASCII code of the character used to display tab (ASCII 9) characters in your file. If you specify 0 or 255, PWB uses the space (ASCII 32) character.

It is sometimes useful to set **Tabdisp** to the code for a graphic character so that tabs can be distinguished from spaces.

Default	Tabdisp:32 The default value 32 specifies the ASCII space character.
See	Filetab, Realtabs, Traildisp, Traillinesdisp

Tabstops

Type	Text The Tabstops switch specifies variable tab stops used by the Tab and Backtab functions. Tab moves the cursor to the next tab stop; Backtab moves the cursor to the previous tab stop. Note that the Tabstops switch has no effect on the handling of tab (ASCII 9) characters in a file.
Syntax	Tabstops: <i>[[tabwidth]]... repeat</i> <i>tabwidth</i> The width of a tab stop. You can repeat <i>tabwidth</i> for as many tab stops as will fit on a PWB line (250 characters). <i>repeat</i> The width of every tab stop after the explicitly listed tab stops. A value of 0 for <i>repeat</i> specifies that there are no tab stops after the list of <i>tabwidth</i> settings. When the cursor is past the last tab stop, the Tab function does nothing.
Default	Tabstops:4
Update	In PWB 1.x, Tabstops is a numeric switch specifying a single value, equivalent to the repeat value in PWB 2.0. The default PWB 2.00 Tabstops setting mimics the default behavior of PWB 1.x.
Example	The Tabstops switch setting Tabstops:4 sets a tab stop every four columns.

Example The setting
 `Tabstops:3 4 7 8`
 sets a tab stop at columns 4, 8, 15, and every eight columns thereafter.

Example The setting
 `Tabstops:3 4 7 25 25 0`
 sets a tab stop at columns 4, 8, 15, 40, and 65. When the cursor is past column 65, the **Tab** function does nothing.

See **Backtab, Entab, Filetab, Realtabs, Tab**

Tilemode

Type Numeric

The **Tilemode** switch specifies the window tiling style. It can take one of the values below:

Value	Tiling Style
0	The first three windows are stacked one above the other.
1	The top two windows are tiled side-by-side.

When four or more windows are open, the tiling is the same in the two styles.

In stacked style (`Tilemode:0`), the top windows are placed one above the other, as shown in gray.

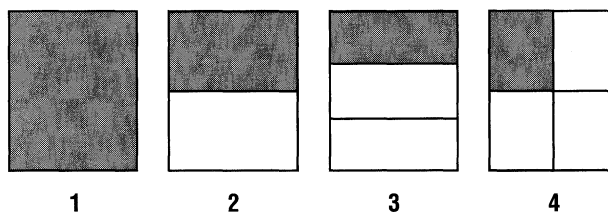


Figure 7.3 Vertical Tiling

In side-by-side style (Tilemode:1), the top two windows are tiled next to each other, as shown in Figure 7.3. This arrangement is good for comparing two files.

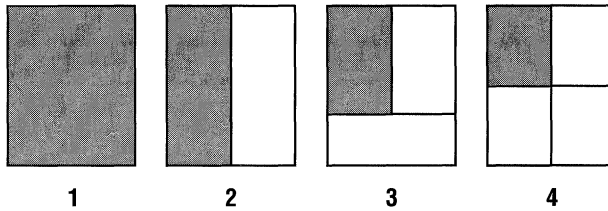


Figure 7.4 Horizontal Tiling

Default Tilemode:0

See Arrangewindow

Timersave

Type Numeric

The **Timersave** switch sets the interval in seconds between automatic file saves. The value must be in the range 0-65,535.

Set **Timersave** to 0 to turn off time-triggered autosave.

Default Timersave:0

See Autosave

Tmpsav

Type Numeric

The **Tmpsav** switch determines the maximum number of files kept in the file history between sessions.

When **Tmpsav** is 0, PWB lets the file history grow without limit; all files loaded into PWB appear in this list until you delete the CURRENT.STS file or change the value of the **Tmpsav** switch.

Default Tmpsav:20

Traildisp

Type Numeric

The **Traildisp** switch specifies the decimal ASCII code for the character used to display trailing spaces on a line. If you specify 0 or 255, PWB uses the space (ASCII 32) character.

Default Traildisp:0

See **Traillines, Trailspace, Traillinesdisp**

Traillines

Type Boolean

The **Traillines** switch determines if PWB preserves or removes empty trailing lines in a file when the file is written to disk.

You can make trailing lines visible by setting the **Traillinesdisp** switch to a value other than 0 or 32.

Syntax **Traillines:**{ **yes** | **no** }

yes Preserve trailing blank lines in the file.

no Remove trailing blank lines from the file.

Default Traillines:no

See **Traildisp, Trailspace**

Traillinesdisp

Type Numeric

The **Traillinesdisp** switch specifies the decimal ASCII code for the character displayed in the first column of blank lines at the end of the file. If you specify 0 or 255, PWB uses the space (ASCII 32) character.

Default Traillinesdisp:32

See **Traillines, Traildisp, Trailspace**

Trailspace

Type Boolean

The **Trailspace** switch determines if PWB preserves or removes trailing spaces from modified lines.

You can make trailing spaces visible by setting the **Traildisp** switch to a value other than 0 or 32.

Syntax **Traillinesdisp:{ yes | no }**

yes Preserve trailing spaces on lines as they are changed.

no Remove trailing spaces from lines as they are changed.

Default Trailspace:no

See **Traillines, Traillinesdisp**

Undelcount

Type Numeric

The **Undelcount** switch determines the maximum number of backup copies of a given file saved by PWB.

This switch is used only when the **Backup** switch is set to `undel`.

Default Undelcount:32767

Undocount

Type Numeric

The **Undocount** switch sets the maximum number of edits per file that you can reverse with **Undo** (ALT+BKSP).

Default Undocount:30

Unixre

Type Boolean

The **Unixre** switch determines if PWB uses UNIX regular-expression syntax or PWB's non-UNIX regular-expression syntax for search-and-replace commands.

The **Unixre** and **Case** switches have no effect on the syntax of regular expressions used by the **Build** and **Word** switches. These switches always use case-sensitive UNIX regular-expression syntax.

Syntax	Unixre: { yes no } yes Use UNIX regular-expression syntax when searching. no Use non-UNIX regular-expression syntax when searching.
Default	Unixre:yes

User

Type	Text The User switch adds a custom menu item to the PWB Run menu.
Syntax	User: <i>title</i> , <i>path</i> , [[<i>arg</i>]], [[<i>out</i>]], [[<i>dir</i>]], [[<i>help</i>]], [[<i>prompt</i>]], [[<i>ask</i>]], [[<i>back</i>]], [[<i>key</i>]] If any argument to the User switch contains spaces, it must be enclosed in double quotation marks. <i>title</i> Menu title for the program to be added. No other command can have the same title. Prefix the character to be highlighted as the access key with a tilde (~) or ampersand (&). If you do not specify an access key, the first letter of the title is used. <i>path</i> Full path of the program. If the program is on the PATH environment variable, you can specify just the filename of the program. <i>arg</i> Command-line arguments for the program. To pass the name of the current file to the program, specify %s in the command line. Default: no arguments. <i>out</i> Name of a file to store program output. If no file is specified and the program is run in the foreground, the current file in PWB receives the output. Default: no output file. <i>dir</i> Current directory for the program. Default: PWB's current directory. <i>help</i> Text that appears on the status bar when the menu item is selected. Default: no help text.

prompt

Determines if PWB prompts for command-line arguments. The value of *arg* is the default response. Specify **Y** to prompt or any other character to run the program without prompting for arguments. Default: no prompt.

ask

Determines if PWB is to prompt for a keystroke before returning to PWB. Specify **Y** to prompt or any other character to return to PWB immediately after running the program. Default: return without prompting.

back

Determines if the program is run in the background under a multithreaded environment. Specify **Y** to run the program in the background or any other character to run it in the foreground. If you run the program in the background, you must also specify *output*. Default: run the program in the foreground.

key

A single digit from 1 to 9, specifying a key from ALT+F1 to ALT+F9 as the shortcut key for the command. Default: no shortcut key.

Default

By default, no custom menu commands are defined.

Example

The **User** switch setting

```
User : "~Print", XPRINT, "/2 %s", LPT1, , \
      "Print the current file with XPRINT", y, n, n, 8
```

specifies the following custom Run menu command:

Option	Description
<i>title</i>	The menu title is <code>Print</code> with the accelerator <code>P</code> .
<i>path</i>	The XPRINT program is expected to be on the PATH.
<i>arg</i>	The default command line specifies the <code>/2</code> option and the current filename.
<i>out</i>	The program output is redirected to the LPT1 device.
<i>dir</i>	The XPRINT program is run in the current directory.
<i>help</i>	The Help line is <code>Print the current file with XPRINT</code> .
<i>prompt</i>	PWB prompts for additional arguments.
<i>ask</i>	PWB doesn't prompt before returning from XPRINT.
<i>back</i>	The XPRINT program is to run in the foreground.
<i>key</i>	ALT+F8 runs the XPRINT program after prompting.

The backslash at the end of the first line of the definition is a TOOLS.INI line continuation.

See **Printcmd, _pwbuser*n*, Usercmd**

Vscroll

Type Numeric

The **Vscroll** switch controls the number of lines scrolled up or down when you move the cursor out of the window. When the window is smaller than the full screen, the amount scrolled is in proportion to the window size.

The minimum value for **Vscroll** is 1. Text is never scrolled in increments greater than the size of the window.

The **Mlines** and **Plines** functions also scroll according to the value of the **Vscroll** switch.

Default Vscroll:1

See **Hscroll**

Width

Type Numeric

The **Width** switch controls the width of the display. Only an 80-column display is supported.

Default Width:80

See **Height**

Word

Type Text

Syntax **Word:** "*regular_expression*"

regular_expression

A macro string specifying a UNIX-syntax regular expression that matches a word.

The **Word** switch specifies a case-sensitive UNIX regular expression that matches a word. The **Unixre** and **Case** switches are ignored.

The **Word** switch accepts a TOOLS.INI macro string. The string can use escape sequences to represent nonprintable ASCII characters. Note that backslashes (\) must be doubled within a macro string.

The **Word** switch is used by functions that operate on words: **Mword**, **Pword**, **Pwbhelp**, right-clicking the mouse for Help, and double-clicking the mouse to select a word.

Default Word:"[a-zA-Z0-9_\$]+"

The default value mimics the behavior of PWB 1.x.

Examples The **Word** switch can be used to change the definition of a word. The following examples show some useful word definitions.

The following setting works the same way as the default setting, except that **Pword** and **Mword** stop at the end of a line:

```
Word:"\\{[a-zA-Z0-9_$]+\\!$\\}"
```

The default setting of the **Word** switch matches Microsoft C/C++ identifiers and unsigned integers. To restrict the definition of a word to match the ANSI C standard for identifiers, you would use the setting:

```
Word:"[a-zA-Z_][a-zA-Z0-9_]*"
```

Another useful setting is to define a word as a contiguous stream of non-space characters:

```
Word:"[^ \t]+"
```

The following **Word** setting defines a word as an identifier or unsigned integer, a stream of white space, a stream of other characters, or the beginning or end of the line. This causes the word-movement functions to stop at each boundary, and allows a double-click to select white space.

```
Word:"\\{[a-zA-Z0-9_]+\\|[ ]+\\|[^\a-zA-Z0-9_]+\\|!$\\|^\\}"
```

Wordwrap

Type	Boolean
	The Wordwrap switch determines if PWB performs automatic word wrap as you enter text.
	When word wrapping is turned on and you type a nonspace character past the column specified by Rmargin , PWB brings the current word down to a new line. A word is defined by the Word switch.
Syntax	Wordwrap :{ yes no }
	yes Wrap words as you enter text.
	no Do not wrap.
Default	Wordwrap:no
Update	See Rmargin

Browser Switches

The PWBBROWSE extension provides the following switches to control the behavior of the Source Browser in PWB.

Browcase

Type

Numeric

The **Browcase** switch determines the initial case sensitivity of the browser when a database is opened. The browser consults this switch only when it opens the database.

This switch must appear in the [PWB-PWBROWSE] tagged section of TOOLS.INI.

A dot appears next to the Match Case command on the Browse menu when the browser matches case. Choose Match Case to turn case-sensitive browsing on and off. Changing the current state does not affect the value of the **Browcase** switch.

Syntax

Browcase:{ 0 | 1 | 2 }

0

Use the case sensitivity stored in the database by BSCMAKE. The default case sensitivity matches the case sensitivity of the source language.

1

Match case for browse queries.

2

Ignore case for browse queries.

Default

Browcase:0

Browdbase

Text

The **Browdbase** switch specifies the browser database to use. When this switch is not set, or the setting is empty, the browser uses the database for the current

project (if any). You set this switch by using the Save Current Database command in the Custom Database Management dialog box.

This switch must appear in the [PWB-PWBROWSE] tagged section of TOOLS.INI.

Syntax

Browdbase: *database*

database

The full filename of the browser database (.BSC file) to use. When *database* is not specified, the browser uses the database for the open project.

C and C++ Switches

The PWBC extension provides the following switches to control the behavior of the C/C++ language extension.

PWBC Library Switches

Type

Text

The PWBC library switches specify whether specific libraries have been installed and, if installed, whether they have default or explicit names.

The setting for this switch appears in the [PWB-PWBC] tagged section in TOOLS.INI. To set any PWBC switch, choose PWBC from the Switch Owner box in the Editor Settings dialog box.

Switch	Libraries
Doslibs	MS-DOS libraries
Winlibs	Windows libraries
Windlllibs	Windows DLL libraries
Os2libs	OS/2 libraries
Dlllibs	OS/2 DLL libraries
Mtlibs	OS/2 multithread libraries

The library switches allow PWB to specify the appropriate library name at link time.

You should set the switch according to the naming convention you selected when setting up. The SETUP program allows you to select one of the following formats for library names:

- Default names of the form *xLIBCy.LIB*, where *x* specifies the memory model (S for small, M for medium, C for compact, or L for large) and *y* specifies the floating-point math package (E for emulator, 7 for 8087, or A for alternate math). The default name for a dynamic-link library (DLL) is *xDLLCy.LIB*, where *y* is either E or A. The default Windows DLL library names have the form *xDLLCyW.LIB*. The default QuickWin library names have the form *xLIBCyWQ.LIB*.
- Explicit names of the form *xLIBCy_o.LIB*, where *x* specifies the memory model, *y* specifies the floating-point math package, and *o* indicates that the library is for a specific operating-mode: R for MS-DOS and W for Windows.

SETUP places the correct settings for these switches in the [PWB-PWBC] section of the TOOLS.PRE file.

Syntax

switch: { **none** | **default** | **explicit** }

switch

One of the switch names listed in the preceding table.

none

Specifies that the library is not installed. PWB asks you to choose between default and explicit names when you select a project template that requires the library.

default

Specifies that the library has the default name. Note that this setting is not recommended for the Windows DLL libraries (*xDLLcyW.LIB*), OS/2 DLL library (*LLIBCDLL.LIB*), or OS/2 multithread library (*LLIBCMT.LIB*). Use explicit names for these libraries.

explicit

Specifies that the library has the explicit (fully qualified) name.

Default

switch: explicit

If the switch is not present in TOOLS.INI, the PWBC extension assumes default names for all libraries.

C_Softcr

Type Boolean

The **C_Softcr** switch determines if PWBC handles automatic indentation for files with the extensions listed in the **C_suffixes** switch. This switch is used only when the **PWB Softcr** switch is set to yes.

The setting for this switch appears in the [PWB-PWBC] tagged section in TOOLS.INI. To set any PWBC switch, choose PWBC from the Switch Owner box in the Editor Settings dialog box.

Syntax **C_Softcr:** { **yes** | **no** }

yes Turns on indentation performed by the PWBC extension.

no Turns off indentation performed by the PWBC extension.

Default C_Softcr: yes

See **C_suffixes**

C_suffixes

Type Text

The **C_suffixes** switch specifies the list of filename extensions for which to perform C auto-indentation. Indentation performed by the PWBC extension is enabled when the **C_Softcr** switch is set to yes, and the **PWB Softcr** switch is also set to yes.

The setting for this switch appears in the [PWB-PWBC] tagged section in TOOLS.INI. To set any PWBC switch, choose PWBC from the Switch Owner box in the Editor Settings dialog box.

Syntax **C_suffixes:** [[*.ext*]]...

[[*.ext*]]...

List of filename extensions of files that use C automatic indentation.

You can specify as many extensions as you want. Separate extensions with spaces. You must include the period (.) in each extension.

Default C_suffixes: .c .h .cxx .hxx .cpp .hpp

Help Switches

The PWBHELP extension provides the following switches to control the behavior of the Help system in PWB.

Color (Help Colors)

The PWBHELP extension defines the following Color switches to set the colors for items displayed in the Help window. These switches must appear in the [PWB-PWBHELP] tagged section of TOOLS.INI. When you choose OK in the Save Colors dialog box, PWB automatically writes the new settings to the correctly tagged section of TOOLS.INI.

Name	Default Value	Description
Color: Helpnorm	87	Plain Help text
Color: Helpbold	8F	Bold Help text
Color: Helpitalic	8A	Italic Help text and the characters
Color: Helpunderline	8C	Emphasized Help text
Color: Helpwarning	70	Current hyperlink

For a complete description of the **Color** switch, see **Color**.

Helpautosize

Boolean

The **Helpautosize** switch determines if PWB displays the Help window according to the size of the current topic or displays Help with its previous size and position.

This switch must appear in the [PWB-PWBHELP] tagged section of TOOLS.INI.

Syntax	Helpautosize: { yes no } yes When displaying a new topic, automatically resize the Help window to the size of the topic. no Do not automatically resize the Help window. The Help window is displayed with its previous size and position.
Default	Helpautosize:no
Update	In PWB 1.x, the Help window is always automatically resized. In PWB 2.00, the Help window is not resized by default.

Helpfiles

Type	Text The Helpfiles switch lists Help files or directories containing Help files that PWB should open in addition to the Help files listed in the HELPFILES environment variable. This switch must appear in the [PWB-PWBHELP] tagged section of TOOLS.INI.
Syntax	Helpfiles: <code>[[file]][;file]...</code> <i>file</i> The filename of a Help file to open or the name of a directory. If a directory name is used, all Help files in the directory are opened. Each <i>file</i> can contain wildcards or environment-variable specifiers.
Default	Helpfiles: By default, PWB uses only the Help files in the current directory and those listed in the HELPFILES environment variable.

Helplist

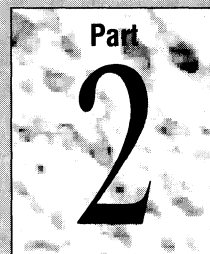
Type	Boolean
	The Helplist switch determines if PWB searches every Help file when you request Help or displays the first occurrence of the topic that it finds.
	This switch must appear in the [PWB-PWBHELP] tagged section of TOOLS.INI.
Syntax	Helplist: { yes no }
	yes Displays a list of Help files that contain the topic you requested Help on when the topic is defined more than once.
	no Does not display a list of topics. PWB displays the first Help associated with the requested topic. To see the other Help screens that define the topic, use the Next command on the Help menu.
Default	Helplist: yes

Helpwindow

(obsolete)

The PWB 1.x **Helpwindow** switch is obsolete and does not exist in PWB 2.00. PWB 2.00 always displays Help in the Help window.

The CodeView Debugger



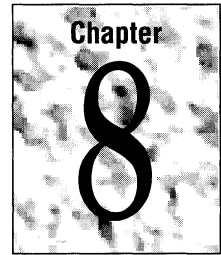
Chapter 8	Getting Started with CodeView	321
9	The CodeView Environment	345
10	Special Topics.....	377
11	Using Expressions in CodeView	399
12	CodeView Reference	417

The CodeView Debugger

The Microsoft CodeView debugger is a powerful diagnostic tool for finding errors in your programs. You can examine code and data and control your program's execution.

Chapter 8 introduces the CodeView debugger. It tells how to prepare programs for debugging and how to customize CodeView. Chapter 9 presents the menus, windows, commands, and other features of the CodeView environment. Chapter 10 explains how to debug Windows programs and p-code. It also describes how to configure CodeView for remote debugging and how to start a remote debugging session. Chapter 11 describes the C and C++ expression evaluators. Chapter 12 contains a reference of CodeView commands.

Getting Started with CodeView



Microsoft CodeView is a window-oriented debugging tool that helps you find and correct errors in Microsoft C/C++ and Macro Assembler (MASM) programs. With CodeView, you can examine source-level code and the corresponding compiled code at the same time. You can execute your code in increments and view and modify data in memory as your program runs.

Microsoft C/C++ includes CodeView for MS-DOS (CV.EXE) and CodeView for Windows (CVW.EXE). The names “CodeView,” “CodeView debugger,” and “the debugger” refer to both versions unless the discussion indicates otherwise.

This chapter shows you how to:

- Write programs to make debugging easier.
- Formulate a debugging strategy.
- Compile and link your programs to include Microsoft Symbolic Debugging Information.
- Set up the files CodeView needs.
- Configure CodeView with TOOLS.INI.
- Start CodeView and load a program.
- Use the CodeView command-line options.
- Use or disable the CURRENT.STS state file.

8.1 Preparing Programs for Debugging

You can use CodeView to debug any MS-DOS or Windows executable file produced from Microsoft C/C++ or Macro Assembler source code. “Compiling” means producing object code from source files. All references to compiling also apply to assembling unless stated otherwise.

General Programming Considerations

This section describes programming practices that make debugging with CodeView easier and more efficient.

Multiple Statements on a Line

CodeView treats each source-code line as a unit. For this reason, you cannot trace and set a breakpoint on more than one statement per line. You can change from Source display mode to Mixed or Assembly display mode (see “The Source Windows” on page 350) and then set breakpoints at individual assembly instructions. If a single statement is broken across multiple lines, you may be able to set breakpoints on only the starting or ending line of the statement.

Macros and Inline Code

Microsoft C, C++, and MASM support macro expansion. Microsoft C and C++ also support inline code generation. These features pose a debugging problem because a macro or an inlined function is expanded where it is used, and CodeView has no information about the source code. This means that you cannot trace or set breakpoints in a macro or inlined function when debugging at the source level.

To work around this condition, you can:

- Manually expand the macro to its corresponding source code.
- Rewrite the macro as a function.
- Suppress inline code generation with the `/Ob0` compiler option.

You can often rewrite macros as inline functions, then selectively disable inlining with a compiler option or pragma so that you can step and trace the routine. Rewriting macros as inlined functions can have additional benefits such as argument type checking. However, in some cases the best solution for debugging macros or inline code is to use Assembly or Mixed display mode.

Segment Ordering and Naming

For assembly-language programs, you must declare your segments according to the standard Microsoft high-level language format. MASM versions 5.10 and later provide directives to specify the standard segment order and naming.

Programs that Alter the Environment

Programs that run under CodeView can read the environment table, but they cannot permanently change it. When you exit CodeView, changes to the environment are lost.

Programs that Access the Program Segment Prefix

CodeView processes the command line in the program segment prefix (PSP) the same way as the C/C++ run-time library does. Quotation marks are removed, and exactly one space is left between command-line arguments. As a result, a program that accesses the PSP directly cannot expect the command line to appear exactly as typed.

Compiling and Linking

After you compile and link your program into a running executable file, you can begin debugging with CodeView. To take full advantage of CodeView, however, you must compile and link with the options that generate CodeView Symbolic Debugging Information. This book refers to this information as “CodeView information,” “debugging information,” or “symbolic information.”

The CodeView information tells CodeView about:

- All program symbols, including locals, globals, and publics
- Data types
- Line numbers
- Segments
- Modules

Without this information, you cannot refer to any source-level names, and you can view the program only in Assembly display mode. When CodeView loads a module that does not contain symbolic information, CodeView starts in Assembly mode and displays the message:

```
CV0101 Warning: No symbolic information for PROGRAM.EXE
```

You get this message if you try to debug an executable file that you did not compile and link with CodeView options, if you use a compiler that does not generate CodeView information, or if you link your program with an old version of the linker. If you retain an old linker version and it is first in your path, the proper information may not be generated for CodeView.

You can specify CodeView compiler and linker options from the command line, in a makefile, or from within the Microsoft Programmer’s Workbench (PWB). To compile and link your program with CodeView options from PWB, choose Build Options from the Options menu, and turn on Use Debug Options. By default, all project templates enable the generation of CodeView information for debug builds.

Compiler Options

Compile with /Zi for full CodeView information.

You can specify CodeView options when you compile a source file of a program you want to debug. Specify the /Zi option on the command line or in a makefile to instruct the compiler to include line-number and complete symbolic information in the object file.

Compile with /Zd to save space.

Symbolic information takes up a large amount of space in the executable file and in memory while debugging. If you do not need full symbolic information in some modules, compile those modules with the /Zd option. The /Zd option specifies that only line numbers and public symbols are included in the object file. In such modules you can view the source file and examine and modify global variables, but type information and names with local scope are not available.

You can also limit the amount of debugging information included in your program by using precompiled headers that do not include debugging information. For more information on generating and using precompiled headers, see Chapter 2 of the *Programming Techniques* manual.

For modules that are compiled with the /Zd option, all names in that module are displayed and can only be referred to using their “decorated name.” The decorated name is the form of the name in the object code produced by the compiler. With full debugging information, CodeView can translate between the source form of the name and the decorated name.

Name decoration encodes additional information into a symbol’s name by adding prefixes and suffixes. For example, the C compiler prefixes the names of functions that use the C calling convention with an underscore. You often see decorated names for library routines in disassembly or output from the Examine Symbols (X) command. For more information on decorated names, see “Symbol Formats” on page 408 and Appendix B.

Disable optimizations with /Od.

All Microsoft high-level language compilers are optimizing compilers that may rearrange and remove source code. As a result, optimizations destroy the correspondence between source lines and generated machine code, which can make debugging especially difficult. While you are debugging, you should disable optimizations with the /Od compiler option. When you finish debugging, you can compile a final version of your program with full optimizations.

Note The /Od option does not pertain to MASM.

Linker Options

Link with /CO.

When you are using Microsoft C/C++, you must use the Microsoft Segmented Executable Linker (LINK) version 5.30 or later to generate an executable file with CodeView information. If you include debugging options when you compile, the compiler automatically invokes the linker with the appropriate options. In turn, LINK runs the CVPACK utility, which compresses the symbolic information.

When compiling, you can specify the compile-only (/c) option to disable running LINK. To include debugging information when you link the object modules separately, specify the LINK /CO option. LINK automatically runs CVPACK when you specify /CO.

If you link with the /EXEPACK option, you must execute the program's startup code before setting breakpoints in the program. If you set breakpoints in a packed executable file before the startup code has executed, CodeView behavior is unpredictable.

An executable file that includes debugging information can be executed from the command line like any other program. However, to minimize the size of the final version of the program, compile and link without the CodeView options.

Examples

The following command sequence assembles and links two files:

```
ML /C /Zi MOD1
ML /C /Zd MOD2
LINK /CO MOD1 MOD2
```

This example produces the object file MOD1.OBJ, which contains line-number and complete symbolic information, and the object file MOD2.OBJ, containing only line-number and public-symbol information. The object files are then linked to produce a smaller file than the file that is produced when both modules are assembled with the /Zi option.

The following commands produce a mixed-language executable file:

```
CL /Zi PROG.CPP
CL /Zi /Od /c /AL SUB1.C
ML /C /Zi /MX SUB2.ASM
LINK /CO PROG SUB1 SUB2
```

You can use CodeView to trace through C, C++, and MASM source files in the same session.

8.2 Debugging Strategies

The process of debugging a program varies from programmer to programmer and program to program. This section offers some guidelines for detecting bugs. If you are familiar with symbolic, source-level debuggers, you can skip this section.

Identifying the Bug

If your program crashes or yields incorrect output, it has a bug. There are times, however, when a program runs correctly with some input but produces incorrect

output or crashes with different input. You can assume a bug exists, but finding it may be difficult.

Locating the Bug

You may not need to use CodeView to find bugs in simple programs. For more complex programs, however, using CodeView can save you debugging time and effort.

Setting Breakpoints

When you debug with CodeView, you usually cycle between two activities:

- Running a small part of the program
- Stopping the program to check its status

You use breakpoints to switch between these tasks. CodeView runs your program until it reaches a breakpoint. At that time, CodeView gives you control. You can then enter CodeView commands in the Command window or use the menus and shortcut keys to proceed.

To find an error, try the following:

- Set breakpoints around the place you think the bug might be. Execute the program with the Go command so that it runs at full speed until it reaches the area that you suspect harbors the bug. You can then execute the program step by step with the Program Step and Trace commands to see if there is a program execution error.
- Set breakpoints when certain conditions become true. You can, for example, set a breakpoint to check a range of memory starting at DS:00, the base of your program's data. If your program writes to memory using a null pointer, the breakpoint is taken, and you can see what statement or variables within the statement are in error.

Setting Watch Expressions

Watch expressions constantly display the values of variables in the Watch window. By setting a Watch expression, you can see how a variable or an expression changes as your program executes.

Try using watch expressions as follows:

- Set a Watch expression on an important variable. Then step through a part of the program where you suspect there is a bug. When you see a variable in the Watch window take on an unexpected value, you know that there is probably a bug in the line you just executed.

- Explore Watch expressions. A bug can appear when your program builds complex data structures. Both the Watch window and the Quick Watch dialog box allow you to explore the data structure by expanding arrays and pointers. Use this feature to make sure the program creates the data structure correctly. As soon as you execute code that destroys the structure, you have probably found a bug.

Arranging Your Display

Your display can be more effective if you arrange your windows so that they display the information you need. You will need at least one Source window. You can open a second Source window to see each assembly-language instruction.

You may also need one or more Memory windows to examine ranges of memory in various formats. You may want to change values in memory. For example, a program that does its own dynamic-memory allocation may need an initialized block of memory. You can edit memory directly in the Memory window or fill the block with zeros using the Memory Fill command. If a certain value is required for a mathematical function, you can type over values displayed in the Memory window or assign the value in the Command window. If you expect a value to appear at a certain location and it does not, you can use the Memory Search command to find it.

Use the Register window to see the CPU registers and the Local and Watch windows to keep track of changing variable values. Open the Calls menu to examine your program's stack to see what routines have been called.

You can set up CodeView's windows to display the information you want to see by using keyboard commands or the commands in the Window menu. For example, when you press `SHIFT+F5` or choose Tile from the Windows menu, CodeView arranges all open windows to fill the entire window area. When the windows are tiled, you can press `ALT+F5` or choose Arrange from the Windows menu. This allows you to move your open windows with a mouse so that you can view several or all of them at once.

8.3 Setting up CodeView

The Microsoft C/C++ SETUP program installs all the necessary CodeView files and correctly configures CodeView for your selected environment. Make sure that all of the CodeView executable files (.EXE and .DLL files) are in a directory listed in the PATH environment variable.

In addition, it is highly recommended that you merge the entries in the TOOLS.PRE file with your TOOLS.INI file. SETUP creates TOOLS.PRE in the

INIT directory that you specify when you run SETUP. If you do not already have a TOOLS.INI file, rename TOOLS.PRE as TOOLS.INI.

This file contains the recommended settings to run CodeView for MS-DOS and CodeView for Windows. For more information on the entries in TOOLS.INI, see “Configuring CodeView with TOOLS.INI” on page 329.

CodeView version 4.0 introduces a new, flexible architecture for the debugger. CodeView is made up of a main executable program: CV.EXE (CodeView for MS-DOS) or CVW.EXE (CodeView for Windows) and a collection of dynamic-link libraries (DLLs). Each DLL implements an aspect of the debugging process.

The following table summarizes CodeView’s component DLLs:

TOOLS.INI Entry	Component	Required	Example
Eval	Expression evaluator	Required	C or C++
Model	Additional nonnative execution model	Optional	P-code
Native	Native execution model	Required	MS-DOS or Windows
Symbolhandler	Symbol handler	Required	MS-DOS or Windows
Transport	Transport layer	Required	Local or remote

This architecture allows for the implementation of such improbable debugging configurations as a Windows-hosted debugger that debugs interpreted Macintosh programs across a network. The existing CVW.EXE could be used with new transport, symbol handling, and execution model DLLs. Instead of creating completely different programs for each combination of host and target, all that is needed is the appropriate set of DLLs.

CodeView Files

CodeView for Windows and CodeView for MS-DOS use several additional files. One of these is the executable program file that you are debugging. CodeView requires one executable (.EXE) file to load for debugging.

program.EXE

An .EXE-format program to debug. CodeView assumes the .EXE extension when you specify the program to load for debugging.

source.ext

A program source file. Your program may consist of more than one source file. When CodeView needs to load a source file for a module at startup or when you step into a new module, it searches directories in the following order:

1. The “compiled directory.” This is the source-file path specified when you invoke the compiler.
2. The directory where the program is located.

If CodeView cannot find the source file in one of these directories, it prompts you for a directory. You can enter a new directory or press `ENTER` to indicate that you do not want a source file to be loaded for the module. If you do not specify a source file, you can debug only in Assembly mode.

CV.HLP

ADVISOR.HLP

Help files for CodeView and the Microsoft Advisor. These two files are the minimum set of files required to use Help during a CodeView session. They must be in a directory listed in the `HELPPFILES` environment variable or in the **Helpfiles** entry of `TOOLS.INI`. You may also want to use the programming language and p-code help files.

TOOLS.INI

Specifies paths for CodeView `.DLL` files and other files that CodeView uses. The Microsoft C/C++ `SETUP` program creates the file `TOOLS.PRE` in the directory specified in your `INIT` environment variable. `TOOLS.PRE` provides a template for the settings you must use in your `TOOLS.INI` file to run CodeView in your operating environment.

If CodeView cannot find the modules it needs in its own directory, it looks for entries in `TOOLS.INI` that specify paths for the modules it needs. You can include other settings for CodeView in `TOOLS.INI`.

TOOLHELP.DLL

System support `.DLL` for CVW.

Remote debugging requires additional files and a different configuration. The files and configuration required for remote debugging are described in “Remote Debugging” in Chapter 10.

8.4 Configuring CodeView with TOOLS.INI

You can configure CodeView and other Microsoft tools including the Microsoft Programmer’s WorkBench (PWB) and NMAKE by specifying entries in the `TOOLS.INI` file. You must have separate sections in `TOOLS.INI` for each tool. `TOOLS.INI` sections begin with a “tag”—a line containing the base name of the executable file enclosed in brackets (`[]`). The tag must appear in column one. The CV and CVW section tags look like this:

```
[CV]
;      MS-DOS CodeView entries
. . .
[CVW]
;      Windows CodeView entries
. . .
```

In the `TOOLS.INI` file, a line beginning with a semicolon (`;`) is a comment.

CodeView looks for certain entries following the tag. Each entry may be preceded by any number of spaces, but the entire entry must fit on one line. You may want to indent each entry for readability.

CodeView TOOLS.INI Entries

You may want to specify or change entries in TOOLS.INI to customize CodeView. Table 8.1 summarizes the TOOLS.INI entries.

Table 8.1 CodeView TOOLS.INI Entries

Entry	Description
Autostart	Commands to execute on startup
Color	Screen colors
Cvdlldllpath	Path to CodeView .DLL files
Eval	Expression evaluator
Helpbuffer	Size of help buffer
Helpfiles	List of help files
Model	Additional execution model (such as p-code)
Native	Native execution model
Printfile	Default name for print command or file
Statefileread	Read or ignore CURRENT.STS state file
Symbolhandler	Symbol handler
Transport	Transport layer

Autostart

The **Autostart** entry specifies a list of Command-window commands that CodeView executes on startup.

Syntax

Autostart:*command*[[;*command*]]...

command

A command for CodeView to execute at startup. Separate multiple commands with a semicolon (;).

Example

The following entry automatically executes the program's run-time startup code. It specifies that CodeView always starts with the Screen Swap option off and the Trace Speed option set to fast.

```
Autostart:0F-;TF;Gmain
```

Color

The **Color** entry is retained only for compatibility with previous versions of CodeView. You should set screen colors with the Colors command on the Options menu.

Cvdlldllpath

The **Cvdlldllpath** entry specifies the default path for CodeView's dynamic-link libraries (DLLs). CodeView searches this path when it cannot find its DLLs in CodeView's directory or along the PATH environment variable. This entry is recommended.

Syntax

Cvdlldllpath:*path*

path

The path to the CodeView .DLL files.

Eval

The **Eval** entry specifies an expression evaluator. The expression evaluator looks up symbols, parses, and evaluates expressions that you enter as arguments to CodeView commands. If there is no **Eval** entry in TOOLS.INI, CodeView loads the C++ expression evaluator by default. CodeView uses the specified expression evaluator when you are debugging modules with source files ending in the specified extensions.

Syntax

Eval:*[[path\]]EEhost evaluator.DLL extension...*

path

The path to the specified expression evaluator.

host

The host environment.

Specifier	Operating Environment
D1	MS-DOS
W0	Windows

evaluator

The source language expression evaluator.

Specifier	Source Language
CAN	C or MASM
CXX	C, C++, or MASM

extension

A source-file extension. CodeView uses the specified expression evaluator when it loads a source file with the given extension. You can list any number of extensions.

Example

The following example loads both the C and C++ expression evaluators for the MS-DOS CodeView:

```
Eva1:C:\C700\DLL\EED1CAN.DLL .C .ABC .ASM .H  
Eva1:C:\C700\DLL\EED1CXX.DLL .CPP .CXX .XYZ .HXX
```

With the entries in this example, when you trace into a module whose source file has the extension `.C`, `.ABC`, or `.ASM`, CodeView uses the C expression evaluator. When you trace into a source file with a `.CXX`, `.CPP`, or `.XYZ` extension, CodeView switches to the C++ expression evaluator. Note that the C++ expression evaluator alone is sufficient for most C, C++, and MASM programs.

You can load expression evaluators after CodeView has started by using the Load command from the Run menu. You can override CodeView's automatic choice of expression evaluator by using the Language command on the Options menu or the **USE** command in the Command window.

For more information about choosing an appropriate expression evaluator and how to use expressions in CodeView, see Chapter 11, "Using Expressions in CodeView."

Helpbuffer

The **Helpbuffer** entry specifies the size of the buffer CodeView uses to decompress help files. You can set **Helpbuffer** to 0 to disable Help and maximize the amount of memory available for debugging. Otherwise, specify a value between 1 and 256.

Syntax

Helpbuffer::*size*

size

The number of kilobytes (K) of memory to use for decompressing help files. The default help buffer size is 24K. Specify 0 to disable help.

The following table shows values you can specify and the actual size of the buffer that is allocated:

Value Specified	Help Buffer Size
1–24	24K
25–128	128K
129–256	256K

The smallest buffer size is 24K, and the largest is 256K.

Helpfiles

The **Helpfiles** entry lists help files for CodeView to load. These files are loaded before any files listed in the HELPFILES environment variable.

Syntax

Helpfiles:*file*[[*;**file*]]...

file

A directory or help file. If you list a directory, CodeView loads all files with the .HLP extension in that directory. Separate multiple files or directories with a semicolon (;).

Model

The **Model** entry specifies an additional execution model that CodeView uses when you are debugging nonnative code such as p-code. The execution model handles tasks specific to the type of executable code that you are debugging.

Syntax

Model:[[*path*\]]*NM**host model*.DLL

path

The path to the specified file.

host

The host environment must be one of the following:

Specifier	Operating Environment
D1	MS-DOS
W0	Windows

model

A nonnative execution model. The p-code execution model (PCD) is required if you plan to debug p-code.

Example

Model:NMD1PCD.DLL

Native

The **Native** entry specifies the native execution model. This DLL handles tasks that are specific to the machine and operating system on which you are running (the host) and specific to the native code (the target).

Syntax

Native:[[*path*\]]*EM**host target*.DLL

path

The path to the specified native execution model.

host

The host environment must be one of the following:

Specifier	Operating Environment
D1	MS-DOS
W0	Windows

target

The target environment must be one of the following:

Specifier	Operating Environment
D1	MS-DOS
W0	Windows

Printfile

The **Printfile** entry lists the default device name or filename used by the Print command on the File menu. This can be a printer port (for example, LPT1 or COM2) or an output file. If **Printfile** is omitted, CodeView prints to a file named CODEVIEW.LST in the current directory. This entry is ignored by CVW, which does not have the Print command.

Syntax

Printfile:*path*

path

The path to the specified output file or the name of a device.

Statefileread

The **Statefileread** entry tells CodeView to read or ignore the CodeView state file (CURRENT.STS) on startup. You can toggle this setting from the command line using the /TSF (Toggle State File) option. These options have no effect on writing CURRENT.STS. CodeView always saves its state on exit.

Syntax

Statefileread:[*y* | *n*]

y (yes)

CodeView reads CURRENT.STS on startup.

n (no)

CodeView ignores CURRENT.STS on startup.

Symbolhandler

The **Symbolhandler** entry specifies a symbol handler. The symbol handler manages the CodeView symbol and type information.

Syntax

Symbolhandler:[[*path*\]]**SH***host*.**DLL**

path

The path to the symbol handler.

host

The host environment must be one of the following:

Specifier	Operating Environment
D1	MS-DOS
W0	Windows

Transport

The **Transport** entry specifies a transport layer. A transport layer provides the data link for communication between the host and target during debugging.

Syntax

Transport:*path***TL***host transport*.**DLL**

path

The path to the specified transport layer.

host

The host environment must be one of the following:

Specifier	Operating Environment
D1	MS-DOS
W0	Windows

transport

Specifies a transport layer.

Specifier	Transport Layer
LOC	Local transport layer
COM	Serial remote transport layer

You specify the local transport layer (LOC) when the debugger and the program you are debugging are running on the same machine. With the appropriate transport layer, CodeView can support remote debugging across serial lines or networks. For more information on remote debugging, see Chapter 10.

The following example specifies the transport layer for debugging a program that is running on the same machine.

Example `Transport:C:\C700\DLL\TLW0LOC.DLL`

8.5 Memory Management and CodeView

CodeView for MS-DOS (CV) requires at least two megabytes of memory. The memory must be managed by a Virtual Control Program Interface (VCPI) server, DOS Protected-Mode Interface (DPMI) server, or extended memory (XMS) manager. These drivers manage memory at addresses above 1 megabyte on an 80286, 80386, or 80486 machine. CodeView loads itself and the debugging information for the program into high memory. In this way, CodeView uses only approximately 17K of conventional MS-DOS memory.

CodeView can use the following memory managers:

- A VCPI server such as EMM386.EXE or EMM386.SYS. With a VCPI server, your program is also able to use EMS memory. To use this memory manager you must have a command in your CONFIG.SYS file such as:

```
DEVICE=C:\DOS\EMM386.EXE ram
```

- A DPMI server such as MSDPMI.EXE. This server is installed by the Microsoft C/C++ SETUP.
- An Extended Memory Standard (XMS) driver such as HIMEM.SYS. To use this memory manager you must have a command in your CONFIG.SYS file such as:

```
DEVICE=C:\DOS\HIMEM.SYS
```

For more information about using memory managers, see your memory manager's documentation. When you make new entries in your CONFIG.SYS file, remember to reboot your system so that your changes take effect.

8.6 The CodeView Command Line

You can specify CV or CVW options when you start them from the command line. You can also specify commands from within the CodeView environment to modify these startup arguments.

Syntax `CV[[W]] [[options]] [[program [[arguments]]]]`
`CV[[W]] @file [[program [[arguments]]]]`

W

Indicates the Windows version of CodeView.

options

One or more options. The CodeView options are described in the “Command-Line Options” section on page 338.

program

Program to be debugged. Specifies the name of an executable file to be loaded by the debugger. If you specify *program* as a filename with no extension, CodeView searches for a file with the extension .EXE. If you do not specify a program, CodeView starts up and displays the Load dialog box where you can specify a program and its command-line arguments.

arguments

The program’s command-line arguments. All remaining text on the CodeView command line is passed to the program you are debugging as its command line. If the program you are debugging does not accept command-line arguments, you do not need to specify any. Once you’ve started debugging, you can change the program’s command-line arguments.

@file

File of command-line arguments. You can also specify arguments in a text file. The file contains a list of arguments, one per line. An argument file lets you specify a large number of arguments without exceeding the operating-system limit on the length of a command line. This is especially useful when starting a session that uses many DLLs.

After CodeView loads its DLLs, processes the debugging information, and loads the source file, the CodeView display appears. If you do not specify a program to debug or CodeView cannot find all of its required DLLs, CodeView prompts for the necessary files.

After starting up, CodeView is at the beginning of the program startup code, and you are ready to start debugging. At this point, you can enter an execution command (such as Trace or Program Step) to execute through the startup code to the beginning of your program.

Leaving CodeView

To exit CodeView at any time, choose the Exit command from the File menu. You can also press ALT+F4, or type Q (for “Quit”) in the Command window.

At this point, you may want to skip ahead to the next chapter, “The CodeView Environment” for information on CodeView’s menus and windows. The rest of this chapter describes each command-line option in detail, then continues with a description of how PWB and CodeView use the CURRENT.STS file.

Command-Line Options

CV and CVW accept some of the same options for debugging. Table 8.2 summarizes the CodeView command line options.

Table 8.2 CodeView Command-Line Options

Option	CV	CVW	Description
<i>/2</i>	Yes	Yes	Use two displays
<i>/8</i>	No	Yes	Use 8514 and VGA displays
<i>/25, /43, /50</i>	Yes	Yes	Set 25-line, 43-line, or 50-line mode
<i>/B</i>	Yes	Yes	Use black-and-white display
<i>/Ccommands</i>	Yes	Yes	Execute commands
<i>/F</i>	Yes	No	Flip video pages
<i>/G</i>	Yes	Yes	Control snow on CGA displays
<i>/I[[0 1]]</i>	Yes	Yes	Trap NMIs and 8259 interrupts
<i>/Ldll</i>	No	Yes	Load DLL or application symbols
<i>/M</i>	Yes	Yes	Disable mouse
<i>/N[[0 1]]</i>	Yes	Yes	Trap nonmaskable interrupts
<i>/S</i>	Yes	No	Swap video buffers
<i>/TSF</i>	Yes	Yes	Read or ignore state file

The remainder of this section describes each option in detail.

Use Two Displays (CV, CVW)

Option

/2

The */2* option permits the use of two monitors. The program display appears on the default monitor, while CodeView displays on the secondary monitor. You must have two monitors and two adapters to use the */2* option. The secondary display must be a monochrome adapter.

If you are debugging a Windows application and have an IBM PS/2 with an 8514 primary display and a Video Graphics Adapter (VGA) secondary display, use the */8* option.

Use 8514 and VGA Displays (CVW)

Option

/8

If your system is an IBM PS/2, you can configure it with an 8514 as the primary display and a VGA as the secondary display. To use this configuration, specify the /8 (8514) option on the CVW command line.

If your VGA monitor is monochrome, it is recommended to use the /B (black-and-white) option. The 8514 serves as the Windows screen and the VGA as the debugging screen.

By default, the debugging screen operates in 50-line mode in this configuration. If you specify the /8 option, you can specify /25 or /43 for 25-line or 43-line mode on the debugging screen.

Warning Results are unpredictable if you attempt to run non-Windows applications or the DOS shell while you are running CVW with the /8 option.

Set Line-Display Mode (CV, CVW)

Options

/25

/43

/50

If you have the appropriate display adapter and monitor, you can display 25, 43, or 50 lines when you are running CV, and 25 or 50 lines when you are running CVW. The mode you specify is saved in the CURRENT.STS file so that it is still in effect the next time you run CodeView.

If you specify a mode that is not supported by your adapter and your monitor, CodeView displays 25 lines. For example, only the 25-line display mode is available when you are running CVW with an Enhanced Graphics Adapter (EGA).

Use Black-and-White Display (CV, CVW)

Option

/B

When you start CodeView, it checks the kind of display adapter that is installed in your computer. If the debugger detects a monochrome adapter, it displays in black and white; if it finds a color adapter, it displays in color. The /B option tells CodeView to display in black and white even if it detects a color adapter.

If you use a monochrome display or laptop computer that simulates a color display, you may want to disable color. These displays may be difficult to read with CodeView's color display.

You can also customize CodeView's colors by choosing the Colors command from the Options menu. For more information, see "Colors" on page 370.

Execute Commands (CV, CVW)

Option

/Ccommands

You type commands in the CodeView Command window. You can also specify Command-window commands when you start CodeView. The */C* option allows you to specify one or more CodeView Command-window commands to be executed upon startup. If you specify more than one command, you must separate each one with a semicolon (;).

If the commands contain spaces or redirection symbols (< or >), enclose the entire option in double quotation marks ("). Otherwise, the debugger interprets each argument as a separate CodeView command-line argument rather than as a Command-window command.

For complete information on CodeView Command-window commands, see Chapter 12, "CodeView Reference."

Examples

The following example loads CV with `CALCPR` as the executable file and `/p TST.DAT` as the program's command line:

```
CV /CGmain CALCPR /p TST.DAT
```

Upon startup, CV executes the high-level language startup code with the command `Gmain`. Since no space is required between the command (`G`) and its argument (`main`), there is no need to enclose the option in double quotation marks.

The next example loads CV with `CALCPR` as the executable file and `/p TST.DAT` as the program's command line. It starts CodeView with a long list of startup commands.

```
CV "/C VS &;G signal_1pd;MDA print_buffer L 20" CALCPR /p TST.DAT
```

CodeView starts with the Source window displaying in Mixed mode (`VS &`). Then it executes up to the function `signal_1pd` with the command `G signal_1pd`. Next, it dumps 20 characters starting at the address of `print_buffer` with the command `MDA print_buffer L 20`. Since some of the commands use spaces, the entire */C* option is enclosed in quotation marks.

In this example, the command directs CV to take Command-window input from the file `SCRIPT.TXT` rather than from the keyboard:

```
CV "/C<SCRIPT.TXT" CALCPR TST.DAT
```

Although the option does not include any spaces, you must enclose it in quotation marks so that the less-than symbol (<) is read by CodeView rather than by the operating-system command processor.

Set Screen-Exchange Method (CV)

Options /F
 /S

CodeView allows you to move between the output screen, which contains your program display output, and the CodeView screen, which contains the debugging display. In MS-DOS, CodeView can perform this screen exchange in two ways: screen flipping or screen swapping. The /F (flipping) and /S (swapping) options allow you to choose the method from the command line. These two methods are:

Flipping

Flipping is the default for a computer with a graphics adapter. CodeView uses the graphic adapter's video-display pages to store each screen of text. Flipping is faster than swapping and uses less memory, but it cannot be used with a monochrome adapter or to debug programs that use graphic video modes or the video-display pages. CodeView ignores the /F option if you have a monochrome adapter.

Swapping

Swapping is the default for computers with monochrome adapters. It has none of the limitations of flipping, but it is slower than flipping and requires more memory. To swap screens, CodeView creates a buffer in memory and uses it to store the screen that is not displayed. When you request the other screen, CodeView swaps the screen in the display buffer for the one in the storage buffer. When you use screen swapping, the buffer is 16K bytes for all adapters. The amount of memory CodeView uses is increased by the size of the buffer.

Suppress Snow (CV, CVW)

Option /G

The /G option suppresses snow that can appear on some CGA displays. Use this option if your CodeView display is unreadable because of snow.

Specify Interrupt Trapping (CV, CVW)

Options /I[[0 | 1]]
 /N[[0 | 1]]

The /I option tells CV whether to handle nonmaskable-interrupt (NMI) and 8259-interrupt trapping. The /N option controls only CodeView's handling of NMIs and does not affect handling of interrupts generated by the 8259 chip. The following table summarizes the options and their effects:

Option	Effect
<code>/I0</code>	Trap NMIs and 8259 interrupts
<code>/I1, /I</code>	Do not trap NMIs or 8259 interrupts
<code>/N0</code>	Trap NMIs
<code>/N1, /N</code>	Do not trap NMIs

You may need to force CodeView to trap interrupts with `/I0` on computers that CodeView does not recognize as IBM compatible. Using `/I0` enables the `CTRL+C` and `CTRL+BREAK` interrupts on such computers.

Load Other Files (CVW)

Option

`/Ldll`
`/Lexe`

To load symbolic information from a dynamic-link library (DLL) or from another application, use the `/L` option when you start CodeView. Specify `/L` for each DLL or application that you want to debug.

When you place a module in a DLL, neither code nor debugging information for that module is stored in an application executable (.EXE) file. Instead, the code and symbols are stored in the library and are not linked to the main program until run time. The same is true for symbols in another application running with Windows. Thus, CVW needs to search the DLL or other application for symbolic information. Because the debugger does not automatically know which libraries to look for, use the `/L` option to preload the symbolic information.

Example

The following command starts CodeView for Windows:

```
CVW /LPRIORITY.DLL /LCAPPARSE.DLL PRINTSYS
```

CVW is used to debug the program `PRINTSYS.EXE`. CVW loads symbolic information for the dynamic-link libraries `PRIORITY.DLL` and `CAPPARSE.DLL`, as well as the file `PRINTSYS.EXE`.

Disable Mouse (CV, CVW)

Option

`/M`

If you have a mouse installed on your system, you can tell CodeView 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 there is a usage conflict between the program and CodeView.

Nonmaskable-Interrupt Trapping (CV, CVW)

Option /N

For information on the /N option, see “Specify Interrupt Trapping” on page 341.

Set Screen Swapping (CV)

Option /S

The /S option sets the CodeView screen-exchange method to swapping. For complete information on CodeView screen-exchange methods, see “Set Screen-Exchange Method” on page 341.

Toggle State-File Reading

Option /TSF

The Toggle State File (/TSF) option either reads or ignores CodeView’s state file and color files, depending on the **Statefileread** entry in the CodeView sections of TOOLS.INI. The /TSF option reverses the effect of the **Statefileread** entry. The **Statefileread** entry is set to yes by default.

These options have no effect on writing the files. CodeView always saves its state on exit.

The effect of different combinations of **Statefileread** and /TSF are summarized in the following table:

/TSF	Statefileread	CodeView Result
Specified	y (or omitted)	Do not read files
Specified	n	Read files
Not specified	y (or omitted)	Read files
Not specified	n	Do not read files

The state file is CURRENT.STS. The color files are CLRFILE.CV4 for CV and CLRFILE.CVW for CVW.

8.7 The CURRENT.STS State File

CodeView and PWB save settings and state information in the CURRENT.STS file. The file contains information about the current state of the two environments.

When you restart CodeView or PWB, they read `CURRENT.STS` and restore their previous state. CodeView uses additional files to save your most recent color settings. These files are `CLRFILF.CV4` for CV and `CLRFILF.CVW` for CVW.

CodeView and PWB search for these files in the directory that the `INIT` environment variable specifies. If no `INIT` environment variable exists, CodeView and PWB search the current directory. If no state file is found, new `CURRENT.STS` and `CLRFILF.CV4` or `CLRFILF.CVW` files are created in the `INIT` directory or the current directory if no `INIT` variable is set.

Information about CodeView stored in `CURRENT.STS` includes:

- Window layout
- Breakpoints
- Watch expressions
- Source, Local, and Memory display options
- Global CodeView options such as case sensitivity, screen exchange method, radix, and expression evaluator

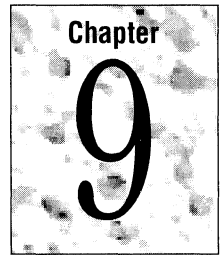
You can set CodeView options in `TOOLS.INI` or on the command line and then modify them during a session. They are saved in `CURRENT.STS` when you exit CodeView. During each CodeView session, these features are set in the following order:

1. From `TOOLS.INI`
2. From the CodeView command line
3. From `CURRENT.STS`
4. During the debugging session

The following items are not saved between sessions:

- The current location (CS:IP).
- The expansion state of watch expressions.
All watch expressions and their format specifiers are restored, but they appear in their contracted state.
- Absolute-address breakpoints.
Breakpoints set at an absolute *segment:offset* address are not saved. CodeView saves breakpoints only at specific line numbers or symbols.
- Memory window addresses.
Each memory window is restored with its display type and options, but CodeView does not save the starting address. Instead, Memory windows show the start of the data segment (address DS:00).

The CodeView Environment



CodeView provides a powerful environment in which to debug programs and dynamic-link libraries (DLLs). Its rich set of commands helps you track program execution and changing data values.

In CodeView you can “point-and-click” your source code to start and stop execution or modify bytes in memory. You can also use more traditional keyboard commands. You can use function keys to execute common commands, such as tracing and stepping through a program. When you quit CodeView, it remembers your breakpoints, window arrangement, watch expressions, and option settings.

This chapter describes the CodeView display, shows you how to use the menu commands, and how to interact with the different types of windows.

9.1 The CodeView Display

The CodeView screen is divided into three parts:

- The menu bar across the top of the screen
- The window area between the menu bar and status bar
- The removable status bar across the bottom of the screen

Figure 9.1 shows a typical CodeView screen with several open windows. The figure shows selected elements of the display, which are described in the sections that follow.

The screenshot displays the CodeView debugger interface with the following components:

- Menu Bar:** File, Edit, Search, Run, Data, Options, Calls, Windows, Help.
- Local Variables Window:**
 - [11] local
 - [BP+00041]+char *name = 0x4508:0x11C6 "extra.txt"
 - [BP-0002] short nMax = 16112
 - [BP-00041]+_iobuf near *File = 0x4508:0x0368
 - [BP-0006] short InWord = 0
 - constant short TRUE = 1
- Registers Window:**
 - [I7]register
 - EAX = 0000368
 - EBX = 00000408
 - ECX = 00000011
 - EDX = 000011C6
 - EFL = 00003202
- Source Code Window:**

```

source1 CS:IP COUNT.C
75:      }
76:
77:      InWord = FALSE;
78:      // Read file buffers
79:      while( ( nMax = fread( Buffer, 1, BUFSIZE, File ) ) )
80:          InWord = CountWords( InWord, nMax );
81:
82:      // Calculate and print the results.
83:      printf( "\n\nFile statistics for %s\n\n", name );

```
- Command Window:**
 - [19] command
 - 1) E "{,COUNT.C,count.EXE} .56"
 - 2) E "{,COUNTBUF.C,count.EXE} .16"
 - >
- Footer:**
 - <F8=Trace> <F10=Step> <F5=Go> <F3=Src1 Fmt>
 - DEC

Figure 9.1 CodeView Display

The Menu Bar

The menu bar displays the names of the CodeView menus. To open a menu, use one of the following methods:

- Click a menu title with the mouse.
- Press ALT plus the menu title's highlighted letter.
- Press and release ALT, use the arrow keys to select a menu, and then press DOWN ARROW or ENTER to open it.

Each command in a menu has a highlighted letter. To choose that command, press the highlighted letter. Many commands also list a shortcut key that you can press at any time instead of opening a menu and choosing a command.

A command that does not apply to a particular situation is dimmed on the menu. When you press the corresponding shortcut key, no action is performed.

The Window Area

Most of your debugging takes place in the window area, where you can open, close, move, size, and overlap the various types of CodeView windows. Although each window serves a different function for debugging, the windows have a number of common features. The Close, Maximize, Restore, and Minimize boxes work in the same way as they do in PWB. The scroll bars also work the same as in

PWB. For information on the window border controls, see Chapter 4, “User Interface Details.”

Only one window can be active at a time. You always use the currently active window, which appears with a highlighted border and a shadow on the screen. The text cursor always appears in the active window.

The Status Bar

The status bar contains information about the active window. It usually includes a row of buttons you can click to execute commands. You can also use the shortcut keys shown on the buttons.

To remove the status bar and gain an extra line for the window area, choose Status Bar from the Options menu, or type the OA- command in the Command window. To restore the status bar, choose Status Bar from the Options menu, or type the OA+ command in the Command window. For more information on this command, see the “Options” command on page 445.

9.2 CodeView Windows

CodeView windows organize and display information about your program. This section describes each CodeView window, the information you can display, and how you can change information and enter commands in the Command window. It also explains how to move among the windows and manipulate them.

How to Use CodeView Windows

Each CodeView window has a different function and operates independently of the others. Only one window can be active at a time. Commands you choose from the menus or by using shortcut keys affect the active window. The following list briefly describes each window’s function:

Source

Displays the source or assembly code for the program you are debugging. You can open a second Source window to view an include file or any ASCII text file.

Command

Accepts debugging commands from the keyboard. CodeView displays the results, including error messages, in the Command window. When you enter a command in a dialog box, CodeView displays any resulting errors in a pop-up window.

Watch

Displays the values of variables and expressions you select. You can modify the value of watched variables, browse the contents of structures and arrays, and follow pointers through memory.

Local

Lists the values of all variables local to the current scope. You can set Local window options to show other scopes. You can modify the values of variables displayed in the Local window.

Memory

Displays the contents of memory. You can open a second Memory window to view a different section of memory. You can set Memory window options to select the format and address of displayed memory. You can directly change the displayed memory by typing in the Memory window.

Register

Displays the contents of the machine's registers and flags. You can directly edit the values in the registers, and you can toggle flags with a single keystroke or mouse click.

8087

Displays the registers of the hardware math coprocessor or the software emulator.

Help

Displays the Microsoft Advisor Help system.

The first time you run CodeView, it displays three windows. The Local window is at the top, the Source window fills the middle of the screen, and the Command window is at the bottom. The Local window is empty until you trace into the main part of the program.

You can open or close any CodeView window. However, at least one Source window must remain open. When you exit CodeView, it records which windows are open and how they are positioned, along with their display options. These settings become the default the next time you run CodeView.

To open a window, choose a window from the Windows menu. Some operations, such as setting a watch expression or requesting help, open the appropriate window automatically.

You can change how CodeView displays information in the Source, Memory, and Local windows. Choose the appropriate window options command from the Options menu. When the cursor is in one of these windows, you can press CTRL+O to display that window's options dialog box.

CodeView automatically updates the windows as you debug your program. To interact with a particular window (such as entering a command or modifying a

variable), you must select it. The selected window is the “active” window. The active window is marked in the following ways:

- The window’s frame is highlighted.
- The window casts a shadow over other windows.
- The cursor appears in the window.
- The horizontal and vertical scroll bars move to the window.

To make a window active, click anywhere in the window or in the window frame. You can also press F6 or SHIFT+F6 to cycle through the open windows, making each one active in turn. You can also choose a window from the Windows menu or press ALT plus a window number. In addition, some CodeView commands make a certain window active.

Moving Around in CodeView Windows

To move the cursor to a specific window location, click that location. You can also use the keyboard to move the cursor as shown in Table 9.1.

Table 9.1 Moving Around with the Keyboard

Action	Keyboard
Move cursor up, down, left, and right	UP ARROW, DOWN ARROW, LEFT ARROW, RIGHT ARROW
Move cursor left and right by words	CTRL+LEFT, CTRL+RIGHT
Move cursor to beginning of line	HOME
Move cursor to end of line	END
Page up and down	PAGE UP, PAGE DOWN
Page left and right	CTRL+PAGE UP, CTRL+PAGE DOWN
Move cursor to beginning of window	CTRL+HOME
Move cursor to end of window	CTRL+END
Move to next window	F6
Move to previous window	SHIFT+F6
Restore window	CTRL+F5
Move window	CTRL+F7
Size window	CTRL+F8
Minimize window	CTRL+F9
Maximize window	CTRL+F10
Close window	CTRL+F4
Tile windows	SHIFT+F5
Arrange windows	ALT+F5

The Source Windows

The Source windows display the source code. You can open a second Source window to view other source files, header files, the same source file at a different location, or any ASCII text file. To open a Source window, use one of the following methods:

- From the Windows menu, choose Source 1 or Source 2.
- In the Command window, type the View Source (**VS**) command.
- Press ALT+3 to open Source window 1.
- Press ALT+4 to open Source window 2.

You cannot edit source code in CodeView, but you can temporarily modify the machine code in memory using the Assemble (**A**) command. For more information on the Assemble command, see page 424.

Source windows can display three different views of your program code in three different modes:

- Source mode shows your source file with numbered lines.
- Assembly mode shows a disassembly of your program's machine code.
- Mixed mode shows each numbered source line followed by a disassembly of the machine code for each line.

Note When you are debugging p-code while Native mode is off, CodeView displays p-code instructions rather than disassembled machine instructions. See the "Options" command on page 445. For more information on p-code, see "Debugging P-code" on page 389.

CodeView automatically switches to Assembly mode when you trace into routines for which no source is available, such as library or system code. The debugger switches back to the original display mode when you continue tracing into code for which source code is available.

For more information on setting display modes, see the "View Source" command on page 457. For detailed information about the Source window display options, see page 368.

The Watch Window

The Watch window displays the value of program variables or the value of expressions you specify in a high-level language. For each expression or variable, you can change the format of the data that is displayed. You can expand aggregate variables, such as structures and arrays, to show all the elements of an aggregate and contract them to save space in the Watch window. You can follow chains of

pointers to display and help debug more complex structures, such as linked lists or binary trees.

To open a Watch window, use one of the following methods:

- From the Windows menu, choose Watch.
- In the Command window, type the Add Watch (**W?**) command followed by the variable or expression name.
- Press ALT+2.

To add expressions to the Watch window, use the Add Watch command from the Data menu or the Quick Watch dialog box (SHIFT+F9). You can also add watch expressions using the Add Watch (**W?**) and Quick Watch (**??**) commands.

Note Do not edit a string in the Watch window.

Type new values for variables in the Watch window.

To change the value of any variable displayed in the Watch window, move the cursor to the value, delete the old value, and type the new value. To change the format in which a variable is displayed or to specify a new format, move the cursor to the end of the variable name and type a new format specifier.

To toggle between insert and overwrite modes, press the INS key.

For information on expanding and contracting aggregate types and following pointers, see the “Quick Watch” command on page 478. For detailed information on specifying and using watch expressions, see the “Codeview Expression Reference” on page 417 and Chapter 11, “Using Expressions in CodeView.”

The Command Window

You type CodeView commands in the Command window to execute code, set breakpoints, and perform other debugging tasks. You can use the menus, mouse, and keyboard for many debugging tasks, but you can use some CodeView commands only in the Command window.

When you first start the debugger, the Command window is active, and the cursor is at the CodeView prompt (>). To return to the Command window after you make another window active, click the command window, or press ALT+9.

Using the Command window is similar to using an operating-system prompt, except that you can scroll back to view previous results and edit or reuse previous commands or parts of commands.

How to Enter Commands and Arguments

You enter commands in the Command window at the CodeView prompt when the Command window is active. Type the command followed by any arguments and press ENTER. Some commands, such as the Assemble (A) command, prompt for an indefinite series of arguments until you enter an empty response. CodeView may display errors, warnings, or other messages in response to commands you enter in the Command window.

You can type commands when the Source window or the Command window is active.

If a Source window is active and the Command window is open, you can still type Command-window commands. When you begin typing, the cursor moves to the Command window and remains there until you press ENTER. The cursor returns to the Source window, and CodeView executes the command. If you have begun typing but do not want to execute a command, press ESC to clear the text and place the cursor at the prompt. After you press ESC, the Command window becomes active.

Command Format

The format for CodeView commands is as follows:

```
command [[arguments]] [[;command2]]
```

The *command* is the command name, and *arguments* are control options or expressions that represent values or addresses to be used by the command. The first argument can usually be placed immediately after *command* with no intervening spaces. Arguments may be separated by spaces or commas, depending on the command. For more information, see Chapter 12, “CodeView Reference.”

To specify additional commands on the same line, separate each command with a semicolon (;).

Commands are always one, two, or three characters long. They are not case sensitive, so you can use any combination of uppercase and lowercase letters. Arguments to commands may be case sensitive, depending on the command.

Example

The following example shows three commands separated by semicolons:

```
MDB 100 L 10 ; G .178 ; MDB 100 L 10
```

The first command (MDB 100 L 10) dumps 10 bytes of memory starting at address 100. The second command (G .178) executes the program up to line 178 in the current module. The third command is the same as the first and is used to see if the executed code changed memory.

Example

This example demonstrates the Comment (*) command:

```
U extract_velocity ;* Unassemble at this routine
```

The first command is the Unassemble (U) command, given the argument `extract_velocity`. The next command is the Comment command. Comment commands are used throughout the CodeView examples in this book.

How to Copy Text for Use with Commands

Copy and paste text instead of retyping.

Text that appears in any CodeView window can be copied and used in a command. For example, an address that is displayed in a Memory window or the Register window can be copied and used in a breakpoint command.

► To copy and use text:

1. Select the text with the mouse or the keyboard.

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

To select text with the keyboard, move the cursor to the desired text, hold down the SHIFT key, and move the cursor with the ARROW keys.

2. Choose the Copy command from the Edit menu or press CTRL+INS.
3. Move the cursor to the location where you want to use the text and choose the Paste command from the Edit menu, or press SHIFT+INS.
4. Edit the command if desired, and press ENTER to execute the command.

Because all input to CodeView windows is line oriented, you cannot copy more than a single line. If you select more than a single line, the Copy command in the Edit menu is unavailable, and CTRL+INS has no effect. However, you can still select more than one line for use with the Print command on the File menu. For more information about the Print command, see “Print” on page 359.

When editing a command, you can toggle between insert and overwrite modes by pressing the INS key.

How to Use the Command Buffer

CodeView keeps the last several screens of commands and output in the Command window. You can scroll the Command window to view the commands you entered earlier in the session. This is particularly useful for viewing the output from commands, such as Memory Dump (MD) or Examine Symbols (X), whose output exceeds the size of the window.

Press TAB to visit previous commands.

The TAB key provides a convenient way to move among the previously entered commands. Press TAB to move the cursor to the beginning of the next command, and press SHIFT+TAB to move to the beginning of the previous command. If the cursor is at the beginning or the end of the command buffer, the cursor wraps around

to the other end. To return to the current command prompt, you can press CTRL+END or press TAB repeatedly.

You can also reuse any command that appears in the Command window without copying and pasting. Move the cursor to the command or press TAB, edit the command if desired, and press ENTER to execute it. When you press ENTER, CodeView restores the original command, copies the new command to the current prompt, and executes the command. If you make a mistake while editing a command, press ESC to restore the line.

The Local Window

You can enter new values for variables in the Local window.

The Local window shows all local variables in the current scope. The Local window is similar to the Watch window, except that the variables that are displayed change as the local scope changes. A variable in the Local window is always shown in its default type format. When you edit in the Local window, you can toggle between insert and overtype modes by pressing the INS key.

You can expand and contract pointers, structures, and arrays the same way you do in the Watch window. You can also change the values of the variables as in the Watch window.

The keyboard shortcut to open or switch to the Local window is ALT+1.

You can see the local variables of each active routine in the stack by selecting the routine from the Calls menu. For more information on this feature, see “The Calls Menu” on page 372.

By default, the Local window shows the addresses of the local variables on the left side of the window. You can turn this address display on or off using the Options (O) command. For more information on the Options command, see page 445.

The Register Window

The Register window displays the names and current values of the native CPU registers and flags. When you are debugging p-code, it displays names and values of the p-code registers and flags. You can change the value of any register or flag directly in the Register window.

To open the Register window, choose Register from the Windows menu, press ALT+7, or F2. You can also view and modify registers by using the Register (R) command. For more information about the Register command, see page 450.

When a register value changes after a program step or trace, CodeView highlights the new value so you can see how your program uses the CPU registers. Depending on the current instruction, the Register window also displays the effective

address at the bottom of the window. This display shows the location of an operand in physical memory and its value.

If you are debugging on an 80386 or 80486 machine, you can view and modify the 32-bit registers. To turn on the 32-bit Registers option, choose the 386 command from the Options menu or use the O3+ command. The 32-bit registers are available if you are debugging on an 80386 or 80486 machine.

When you are debugging p-code, CodeView displays the p-code registers: DS, SS, CS, IP, SP, BP, PQ, TH, and TL.

If your program has taken an unexpected turn, you may be able to compensate for the problem and continue debugging if you change the value of a register or a flag. You can change a flag value before a dump or looping instruction to test a different branch of code, for example. You can change the instruction pointer (CS:IP) to jump to any code in your program or to execute code you have assembled elsewhere in memory.

To change the value of any register, move the cursor to the register value you want to change and overwrite the old value with the new value. The cursor automatically moves to the next register.

Although you cannot change the value of the flag register numerically in the Register window, you can conveniently toggle the values of each flag using either the mouse or the keyboard:

- To toggle a flag with the mouse, double-click the flag.
- To toggle a flag using the keyboard, move the cursor to the flag and press any key except ENTER, TAB, or ESC. After toggling a flag, CodeView moves the cursor to the next flag.

To restore the value of the last flag or register that you changed, choose Undo from the Edit menu or press ALT+BACKSPACE. If you happen to lose the cursor somewhere in the register window, press TAB. The TAB key moves the cursor to the next register or flag that can be changed.

The 8087 Window

The 8087 window displays the current status of the math coprocessor's registers and flags. If you are debugging a program that uses the software-emulated coprocessor, the emulated registers are displayed. To open the 8087 window, choose 8087 from the Windows menu or press ALT+8.

The display in the 8087 window is the same as the display produced by the 8087 (7) command, except that the window is continually updated to show the current status of the math coprocessor. For more information about the display, see the "8087" command on page 473.

If your program uses floating-point libraries provided by several Microsoft languages, or if your program does not use floating-point arithmetic, the 8087 window and 8087 command display the message:

```
Floating point not loaded
```

CodeView displays this message until at least one floating-point instruction has been executed.

The Memory Windows

Memory windows display memory in a number of formats. CodeView allows two Memory windows to be open at the same time. You can open a Memory window in several ways:

- From the Windows menu, choose Memory 1 or Memory 2.
- From the Options menu, choose Memory1 Window when no Memory windows are open.
- In the Command window, enter the View Memory (**VM**) command.
- At the keyboard, press ALT+5 or ALT+6.

By default, memory is displayed as bytes or as the last type specified by a Memory Enter (**ME**), Memory Dump (**MD**), or View Memory (**VM**) command. The byte display shows hexadecimal byte values followed by an ASCII representation of those byte values. For values that are outside the range of printable ASCII characters (decimal 32 to 127), CodeView displays a period (.).

How to Change Memory Display Format

It is not always most convenient to view memory as byte values. If an area of memory contains character strings or floating-point values, you might prefer to view them in a directly readable form.

To change the display format of a Memory window, choose Memory1 Window or Memory2 Window from the Options menu. CodeView displays a dialog box where you can choose from a variety of display options. When the cursor is in a Memory window, you can press CTRL+O to display the corresponding Memory Window Options dialog box. You can also set memory display options using the View Memory (**VM**) command. For detailed information about the display options, see “View Memory” on page 455.

To cycle through the display formats, click the <Sh+F3=Mem1Fmt> or <Sh+F3=Mem2 Fmt> buttons on the status bar, or press SHIFT+F3. Pressing CTRL+SHIFT+F3 displays the cycle in reverse order.

When you first open the Memory window, it displays memory starting at address DS:00. To change the starting address, use one of the commands to set Memory window options. You can specify the starting address or enter an expression to use as the starting address.

You can also type over the *segment:offset* addresses shown in the left column of the Memory window to change the displayed addresses. Move the cursor to an address in the window, or repeatedly press TAB until the cursor is on an address, and type a new address.

How to Change Memory Directly

To change the values in memory, overwrite the value you want to change. To move quickly from field to field in the Memory window, press TAB. You can change memory by entering new values for the format that is displayed or by typing over the raw bytes in the window. CodeView ignores the input if you press a key that does not make sense for the current format (for example, if you type the letter X in anything but ASCII format).

To undo a change to memory, choose Undo from the Edit menu, or press ALT+BACKSPACE.

How to View Memory at a Dynamic Address

Live expressions make it easy for you to watch a dynamic view of an array or pointer in the Memory window. “Live” means that the starting address of memory in the window changes to reflect the current value of an address expression.

To create a live expression, choose the Memory1 Window or Memory2 Window command from the Options menu. In the Memory Window Options dialog box, type in an address expression, then turn on the Re-evaluate Expression Always (Live) option.

It is usually more convenient to view an item in the Watch window than in the Memory window. However, some items are more easily viewed using live expressions. For example, you can examine what is currently on top of the stack by entering SS:SP as the live expression.

The Help Window

In CodeView, you can request Help:

- From the Help menu.
- By pressing F1 when the cursor is on the keyword, menu, or dialog box for which you want Help.

- By clicking the right mouse button on a Help keyword.
- Using the Help (**H**) command.
- By choosing Help from the Windows menu. You can also press ALT+0 for Help on CodeView windows.

The Microsoft Advisor Help window is displayed whenever you request Help for CodeView. For information about getting the most out of the Microsoft Advisor Help system, see Chapter 23.

9.3 CodeView Menus

Many commands that you are likely to use frequently are in the CodeView menus. This section describes the menus and the commands or options in each menu. Not all commands are available in both versions of the CodeView debugger. When applicable, the menu descriptions discuss command availability.

The File Menu

The File menu contains commands to load source files and other ASCII text files into the Source window, print from the active window, start an operating-system shell, and end the debugging session. Only the Open Source, Open Module, and Exit commands are available in CVW.

The following table summarizes the commands on the File menu:

Command	Purpose
Open Source	Opens a source, include, or other text file
Open Module	Opens a source file for a module in the program
Print	Prints all or part of the active window
DOS Shell	Goes to the operating-system prompt temporarily
Exit	Exits CodeView

Open Source

The Open Source command displays the Open Source File dialog box. You can select the name of the source file, include file, or other text file to display in the active Source window.

Open Module

The Open Module command displays the Open Module dialog box. This dialog box provides an easy way to load the source file for any module in your program.

The dialog box lists the source files that make up the modules in the program you are debugging. Only those modules that include line-number or full symbolic information are listed.

CodeView displays the source file you choose in the active Source window.

Print

In CodeView for MS-DOS only, the Print command displays the Print dialog box, which offers several options to write information in the active window to a device or a file. You can print text in the active window in the following ways:

- Window view, which prints text that currently appears in the active window
- Complete window contents, which prints the contents of the active window, including what has scrolled out of the window

To print to a file, specify a filename in the dialog box. To append the printed text to the end of the file, select Append. To overwrite a file that already exists, select Overwrite. If you specify a device instead of a file, you can choose either Append or Overwrite.

To print directly to a printer, specify the name of the printer port such as LPT1 or COM2. You must specify a filename or a device name. CodeView reports an error if you omit the name.

DOS Shell

In MS-DOS only, you can use the DOS Shell command to leave CodeView temporarily and go to the operating-system prompt.

When you choose the Shell command, CodeView starts a second copy of the command processor specified by the COMSPEC environment variable. If there is not enough memory to open a new shell, a message appears. Even if you do have enough memory to start a command shell, you may not have enough memory to execute large programs from the shell.

While in the shell, do not start any terminate-and-stay-resident (TSR) programs, such as MSHERC.COM, and do not delete files you are working on during your debugging session. Also, do not delete any files used by CodeView, such as the CURRENT.STS file.

To return to CodeView, type `exit` at the operating-system prompt to close the shell. For more information about starting a shell, see the “Shell Escape” command on page 468.

Exit

The Exit command saves the current CodeView environment and returns to the program that called CodeView, such as COMMAND.COM, PWB, or another editor. CodeView saves the window arrangement, watch expressions, option settings, and most breakpoints in the state file, CURRENT.STS. It saves current color settings in CLRFILE.CV4 if you are using CV and in CLRFILE.CVW if you are using CVW.

When you start the debugger at a later time, CodeView restores these settings. To prevent CodeView from restoring the information it stores in CURRENT.STS, start the debugger with the /TSF option or use the **Statefileread** entry in your TOOLS.INI file.

The Edit Menu

The Edit menu contains commands to undo changes to window's fields, copy selected text to the clipboard, and paste the contents of the clipboard into a window. For more details on editing in CodeView's windows, see "CodeView Windows" on page 347.

The following table summarizes the commands on the Edit menu:

Command	Purpose
Undo	Reverses the last editing change
Copy	Copies the selected text to the clipboard
Paste	Inserts the contents of the clipboard at the cursor

Undo

The Undo command (ALT+BACKSPACE) reverses the last editing action.

Copy

The Copy command (CTRL+INS) copies selected text to the clipboard. Because input to CodeView is restricted to single lines, you can copy only a single line of text. If you select more than a single line of text, the Copy command is disabled and pressing CTRL+INS has no effect.

Paste

The Paste command (SHIFT+INS) inserts text from the clipboard at the cursor in the Command window.

The Search Menu

The Search menu provides commands to find strings and regular expressions in source files and to locate the definitions of labels and routines.

The following table summarizes the commands on the Search menu:

Command	Purpose
Find	Searches for a text string or pattern in the source file
Selected Text	Searches for the selected text in the source file
Repeat Last Find	Repeats the last text search
Label/Function	Searches for a label or function definition in the program

Find

The Find command displays the Find dialog box. In the Find What text box, type the text or pattern you want to find. You can also select text in a window and then choose Find. The text you selected is shown in the dialog box.

You can select options in the dialog box to modify the way CodeView searches for text. The following options are available:

Whole Word

CodeView matches the text only when it occurs as a word by itself. For example, when you search for the pattern `print` with the Whole Word option, CodeView finds `print("eep")`, but not `error_print("eep")`.

Match Case

CodeView matches the text when each letter in the pattern has the same case as the source file. For example, the pattern `fish` matches `fish`, but not `Fish`.

Regular Expression

CodeView treats the text as a regular expression. Regular expressions provide a powerful way to specify patterns that match several different sections of text. For more information about regular expressions, see Appendix A.

To search for a regular expression in the active Source window using the Command window, you can type the Search (*/*) command followed by the string. CodeView searches the file starting at the current position. CodeView places the cursor on the next occurrence of the search pattern. If the end of the file is reached without finding a match, CodeView wraps around and continues searching from the beginning of the file.

Selected Text

The Selected Text command (CTRL+A) searches for the next occurrence of the selected text in the Source window.

Repeat Last Find

The Repeat Last Find command (ALT+) searches for the next occurrence of the search pattern, including search options, you last specified.

Label/Function

The Label/Function command lets you search the program's symbolic information for the definition of a label or routine. When you choose Label/Function, the Find Label/Function dialog box appears. The currently selected text or the word at the cursor comes up in the Label/Function Name text box. You can search for this name or type in a different label or routine name.

When you choose OK, CodeView searches the symbolic information in the program for the name. When the label or routine name is found, CodeView positions the cursor at the name in the source file.

To view the current program location after searching, choose the first item in the Calls menu or type the Current Location (.) command in the Command window.

The Run Menu

The Run menu consists of commands to restart the program, animate the program in slow motion, change the program's arguments, load a new program, or configure the modules CodeView is using.

The following table summarizes the commands on the Run menu:

Command	Purpose
Restart	Restarts the program
Set Runtime Arguments	Changes the program's run-time arguments and restarts the program
Animate	Executes the program in slow motion
Load	Loads a new program to debug, sets run-time arguments, and configures CodeView's modules

Restart

The Restart command resets execution to start at the beginning of the program. After you issue the command, CodeView:

- Initializes all program variables.
- Resets the pass counts for all breakpoints.

- Preserves existing breakpoints, watch expressions, and the program's command-line arguments.

You can use Restart any time after execution stops: at a breakpoint, while stepping or tracing, or when your program ends. If your program redefines interrupts, Restart may not work correctly because it does not execute any cleanup or exit-list code in the program. If your program requires this code to be executed, let the program run to the end before restarting, or use the Display Expression (?) command in the Command window to call the cleanup routines. For more information on calling program routines, see "Display Expression" on page 477.

Set Runtime Arguments

The Set Runtime Arguments command lets you change your program's command-line arguments. When you set new arguments, CodeView restarts the program.

Animate

The Animate command executes your program in slow motion. CodeView highlights each statement in the Source window as your program executes. This allows you to see the flow of execution. To stop animation, press any key.

You can set the animation speed with the Trace Speed command from the Options menu or with the Trace Speed (T) Command-window command.

Load

The Load command displays the Load dialog box, which you can use to:

- Load executable (.EXE or .DLL) files.
- Change the program's command-line arguments.
- Specify different CodeView components from those specified in TOOLS.INI, such as a different expression evaluator or the p-code execution model.

Loading Programs or DLLs To load program or DLL symbols into the debugger, type a filename in the File to Debug text box, or use the mouse or keyboard to select a file from the File List box. Use the Drives/Dirs list box to change to a different drive or directory.

Set Command-Line Arguments Use the Arguments text box to change the command-line arguments to the program you are debugging or to set entirely new arguments. Type the arguments to your program as you would on the command line.

Configure CodeView Modules CodeView uses a default setting for an execution model, transport layer, and expression evaluator if any of these is not specified in TOOLS.INI. Click the Configure button to load different CodeView DLLs. The Configure dialog box lists the DLLs that CodeView has loaded. CodeView loads several DLLs that are required to debug your programs. These DLLs include:

- Expression evaluators for various languages and environments.
- Execution models for various operating systems.
- Execution models for p-code.
- Transport layers.

To load new DLLs, click the Change buttons on the right side of the dialog box.

The Data Menu

The Data menu provides commands to add and delete watch expressions and breakpoints. Watch expressions allow you to observe how variables change as your program executes and also to expand arrays and dereference pointers. Breakpoints allow you to stop execution of your program to check the values of your variables, determine execution flow, and change how your program executes.

For more information about watch expressions, see Chapter 11, “Using Expressions in CodeView” and the “Add Watch Expression” command on page 460.

The following table summarizes the commands on the Data menu:

Command	Purpose
Add Watch	Adds an expression to the Watch window
Delete Watch	Deletes an expression from the Watch window
Set Breakpoint	Sets a breakpoint in the program
Edit Breakpoints	Modifies or removes existing breakpoints
Quick Watch	Displays a quick view of a variable or expression

Add Watch

The Add Watch command (CTRL+W) displays the Add Watch dialog box, which shows the selected text or the word at the cursor in the Expression text box. You can enter a different expression or add a format specifier to the expression. When you choose OK, the expression is added to the end of the Watch window.

Delete Watch

The Delete Watch command (CTRL+U) displays the Delete Watch dialog box, which displays a list of the watch expressions in the Watch window. Select the expression you want to delete from the list and choose OK. Click the Clear All button to remove all expressions from the Watch window.

You can also delete expressions directly from the Watch window. Use the mouse or the cursor keys to move the cursor to the expression you want removed, and press CTRL+Y.

Set Breakpoint

The Set Breakpoint command displays the Set Breakpoint dialog box, which allows you to select from several kinds of breakpoints and set options for each type. The following list describes the breakpoints you can set:

Break at Location

This is the simplest type of breakpoint. You specify an address or a line number where you want execution to pause. To specify a line number, precede it with a period (.); otherwise, CodeView will interpret it as an address. When your program's execution reaches the breakpoint location, your program stops temporarily, and you can enter CodeView commands.

Break at Location if Expression is True

You specify a location and an expression. Whenever execution reaches that location, CodeView checks the expression. If the expression is true (nonzero), the breakpoint is taken. Otherwise, execution continues.

Break at Location if Expression has Changed

You specify a location and an expression that represents a variable or any portion of memory. To specify a range of memory, enter the length of the range in the Length text box. CodeView checks the variable or the range of memory when execution reaches the breakpoint location. If the value of any byte has changed since the last time CodeView checked, the breakpoint is taken. Otherwise, execution continues.

Break When Expression is True

This breakpoint is taken whenever the expression becomes true. CodeView evaluates the expression after every line or every instruction, instead of only at a certain location. As a result, this type of breakpoint can greatly slow your program's execution.

Break When Expression has Changed

CodeView checks the variable or the range of memory as each line or each instruction is executed. You can also specify a range with the Length text box. This type of breakpoint can also slow your program's execution.

Each breakpoint is numbered, beginning with 0. For each type of breakpoint, you can set several options. If you try to use an option that does not apply to a certain breakpoint, CodeView displays *N/A* in the edit box and ignores that option. The options are:

Location

Specifies where CodeView should evaluate the breakpoint.

Expression

Specifies an expression that causes a break when it becomes true or a location that is to be watched for changes.

Length

Specifies a range of memory (starting at the address in the Expression text box) to watch for changes.

Pass Count

Specifies the number of times to pass over the breakpoint when it otherwise would be taken. For example, a pass count of 10 tells CodeView to ignore the breakpoint ten times.

Commands

Specifies a list of Command-window commands, separated by semicolons, that are executed when the breakpoint is taken. If several breakpoints with commands are taken, the commands are queued and executed in first-in, first-out order.

As shortcuts, you can also set simple (break at location) breakpoints with the following methods:

- Double-click the line in the Source window.
- Move the cursor to the breakpoint location in the Source window and press F9.

A line with a breakpoint is highlighted. In the Mixed and Assembly modes, an assembly-language comment that displays the breakpoint number appears. For example:

```
0047:0b30 57          push di                ;BK0
```

In this example, breakpoint number 0 is set at the address 0047:0B30.

You can also set breakpoints with the Breakpoint Set (**BP**) command. See the “Breakpoint Set” command on page 429.

Edit Breakpoints

The Edit Breakpoints command displays the Edit Breakpoints dialog box, where you can add, remove, change, enable, and disable breakpoints. Select a breakpoint from the list of breakpoints, then choose one of the command buttons on the right side of the dialog box.

The list of breakpoints in the Edit Breakpoints dialog box shows the current state of each breakpoint in your program. For more information on the format of the breakpoint list, see the “Breakpoint List” command on page 429.

The command buttons in the Edit Breakpoints dialog box are described in the following table:

Button	Description
Add	Adds a new breakpoint
Remove	Removes the selected breakpoint
Modify	Modifies the same breakpoint
Enable	Activates a disabled breakpoint
Disable	Disables an active breakpoint
Clear All	Removes all breakpoints

If you select the Modify button, CodeView displays the Set Breakpoint dialog box with the appropriate options set for the breakpoint you selected. You can then modify the options and set the breakpoint just as you do with the Set Breakpoint command.

When you disable a breakpoint by selecting the Disable button, CodeView does not evaluate the breakpoint. Program execution continues as if the breakpoint was never set.

You may encounter several occasions where it is useful to disable a breakpoint. Sometimes a certain breakpoint is not practical when you are debugging a routine nested deeply in your program. You can reenable the breakpoint later when you really need it. Also, conditional breakpoints are evaluated at every program step and can slow execution. You can disable some conditional breakpoints in areas of your program where you’re certain you won’t need them.

Quick Watch

The Quick Watch command (SHIFT+F9) displays the Quick Watch dialog box, which shows the variable at the cursor position or the selected expression. The Quick Watch dialog box is similar to the Watch window. However, you mainly use Quick Watch for a quick exploration of the current values in an array or a pointer-based data structure, rather than as a method to constantly display the values.

The Quick Watch dialog box automatically expands structures, arrays, and pointers to their first level. You can expand or contract an element just as you can in the Watch window. If the expanded item needs more lines than the Quick Watch dialog box can display, you can scroll the view up and down.

Choose the Add Watch button to add a Quick Watch item to the Watch window. Expanded items appear in the Watch window as they are displayed in the Quick Watch dialog box.

For complete information on using the Quick Watch dialog box, see the “Quick Watch” command on page 478.

The Options Menu

The Options menu contains commands to change the default behavior of CodeView commands and the display status of CodeView windows. You can also set display options with various Command-window commands. When the cursor is in one of the Source, Memory, or Local windows, you can press CTRL+O to display the window’s Options dialog box.

For menu items that are toggles, a bullet appears to the left of the item when the option is turned on. No bullet appears when it is turned off.

The following table summarizes the commands on the Options menu:

Command	Purpose
Source1 Window	Sets Source window 1 display options
Source2 Window	Sets Source window 2 display options
Memory1 Window	Sets Memory window 1 display options
Memory2 Window	Sets Memory window 2 display options
Local Options	Sets Local window display options
Trace Speed	Sets animation speed
Language	Sets the expression evaluator
Horizontal Scrollbars	Toggles horizontal scroll bars on windows
Vertical Scrollbars	Toggles vertical scroll bars on windows
Status Bar	Toggles the status bar display
Colors	Changes colors of CodeView screen elements
Screen Swap	Toggles screen exchange
Case Sensitivity	Toggles case sensitivity of symbols
32-Bit Registers	Toggles display of 32-bit registers
Native	Toggles display of p-code or machine code instructions

Source Window

The Source Window command displays the Source Window Options dialog box. In this dialog box, you can set the source display mode and other options for the current Source window. These options are as follows:

Option	Description
Follow CS:IP thread of control	Keeps the current program location visible in the active Source window.
Source	Displays the source code for the program.
Mixed Source and Assembly	Displays each source line followed by the disassembly of the code generated for that line.
Assembly	Displays a disassembly of the machine code in your program.
Tab Length	Sets the number of spaces to which tab characters expand in the source file.
Show Machine Code	Shows the address and hexadecimal representation of the machine code in Mixed and Assembly modes.
Show Symbolic Name	Shows the symbol name in assembly-language displays instead of the numeric value of the symbol.

Memory Window

The Memory Window command displays the Memory Window Options dialog box, where you can set the starting address and display format of the active Memory window. For details, see “The Memory Windows” on page 356 and the “View Memory” command on page 455.

Local Options

You can specify the scope of variables to be displayed in the Local window. When you select Local Options from the Options menu, a dialog box appears in which you can select any combination of lexical, function, module, executable, and global scopes. You can also toggle the display of addresses in the Local window from the Local Options dialog box. When you turn Show Addresses on, the BP-relative address of each local variable is shown in the Local window. Otherwise, the Local window shows only the names of the variables.

You can also use the Options (**OL**) command in the Command window to specify the scope of variables to be displayed in the Local window. For information about the Options command, see page 445.

Trace Speed

The Trace Speed command displays the Trace Speed dialog box, which presents a list of three speeds from which you can select.

When you use the Animate command to run your program in slow motion, CodeView pauses execution between each step. The duration of the pause is set with the Trace Speed command. **Slow** pauses for 1/2 second. **Medium** pauses for

1/4 second. **Fast** runs the program as fast as possible while still updating CodeView windows and evaluating breakpoints and watch expressions.

Language

The Language command displays the Language dialog box, which presents a list of the expression evaluators that CodeView has loaded, plus the **Auto** option.

In your TOOLS.INI file, you can configure CodeView to load a number of different expression evaluators. You can also load expression evaluators by choosing Load from the Run menu. Only one expression evaluator can be active at a time.

The **Auto** setting is the default. It tells CodeView to set the expression evaluator automatically based on the extension of the source file you are debugging in the current Source window. For more information on expression evaluators, see “Configuring CodeView with TOOLS.INI” on page 329.

For more information on using expression evaluators, see Chapter 11, “Using Expressions in CodeView.”

Horizontal Scrollbars

The Horizontal Scrollbars command toggles the horizontal scroll bars on and off. When scroll bars are off, you can drag the bottom window frame, as well as the size box, to resize the window.

Vertical Scrollbars

The Vertical Scrollbars command toggles the vertical scroll bars on and off. When scroll bars are off, you can drag the right window frame, as well as the size box, to resize the window.

Status Bar

The Status Bar command toggles the status bar on and off. When the status bar is off, you gain an extra line of space for windows.

Colors

The Colors command displays a dialog box that lets you change the colors of CodeView screen elements. The Item list box displays all the elements of the debugging screen. The Foreground and Background list boxes show the current color settings for the highlighted element in the Item list box.

To change the color of a screen element, choose an element in the Item list box, then choose foreground and background colors. When you are done, click the OK button. Your new color settings take effect as soon as you exit the dialog box.

If you make a number of changes and want to go back to your previous color settings, click the Reset button. You can then start changing colors again. To close the dialog box without making any changes, click the Cancel button. To reset to the standard CodeView colors, click the Use Default button.

When you specify colors using the Colors command in CodeView, the colors are saved in CLRFILE.CVW if you are using CodeView for Windows and in CLRFILE.CV4 if you are using CodeView for DOS. CodeView saves these files in the directory specified by the INIT environment variable or in the current directory if no INIT environment variable is set. These settings become the new default colors.

Screen Swap

The Screen Swap command toggles screen exchange on or off. By default, CodeView switches to your program's output screen whenever you execute code in the program. CodeView uses either screen flipping or screen swapping, depending on the command-line options you used to start the debugger. See "Set Screen-Exchange Method" on page 341.

If your program sends no output to the screen, you'll probably want to turn Screen Swap off. This setting continuously displays CodeView's screen while your program executes.

If Screen Swap is off and your program writes to the screen, a portion of the CodeView display may be overwritten. If this happens, type the Refresh (@) command in the Command window.

Case Sensitivity

The Case Sensitivity command toggles case sensitivity on or off. When Case Sensitivity is on, CodeView treats symbol names as case sensitive (that is, a lowercase letter is different from its corresponding uppercase letter). This option affects only commands that deal with symbols in your program; it does not affect the text-searching commands.

32-Bit Registers

The 32-Bit Registers command toggles 386 mode on and off. When 386 mode is on, a bullet appears next to the command on the menu, and CodeView displays the 32-bit registers in the Register window. In this mode, CodeView can also assemble instructions that use 32-bit registers or memory operands.

Native

When you are debugging a program that uses p-code, you use the **Native** command to toggle between p-code instructions and native machine instructions. With Native mode on, CodeView displays your program's native CPU instructions. With Native mode off, CodeView displays the instructions in p-code.

For more information on debugging p-code, see page 389.

The Calls Menu

The Calls menu shows what routines have been called into your program during debugging. Its contents change to reflect their current status. The current routine is at the top of the menu; the routine that called it appears just below. Routines are listed in the reverse order in which they were called. At the bottom of the list is your program's main routine. In C, for example, **main** appears at the bottom. When you are debugging a Windows application, **winmain** is at the bottom of the list.

The Calls menu is empty until the program enters at least one routine that creates a stack frame. Listed with each routine name are the arguments to each routine in parentheses. The menu's width expands to accommodate the widest entry. Arguments are shown in the current radix, except for pointers, which are always shown in hexadecimal.

When you choose a routine from the Calls menu, CodeView displays the source code for that routine and updates the Local window to show the local variables in that routine. The cursor moves to the return location to show the next line or instruction that will be executed when control returns to that routine.

To step out of deeply nested code, choose a routine and then press F7.

Choosing a routine from the Calls menu does not affect program execution; it provides you with a convenient way to view a routine's source code and local variables. However, since the cursor is positioned at the return location, you can press F7 to execute through the stack of nested calls to that line. This is especially convenient when you find you've accidentally traced into a deeply nested set of routines which you know to be bug-free. Rather than continue a tedious trace until you work your way out of the stack of routines, you can choose a routine from the Calls menu and press F7. CodeView executes through the nested routines until control returns to the point you chose.

A routine may not be visible in the Calls menu under the following circumstances:

- You have traced only startup or termination routines from the run-time library.
- Routine calls are nested so deeply that not all routines appear on the menu.
- The stack has been corrupted.

- CodeView cannot trace through the stack frame because the BP register is overwritten.

The Windows Menu

If you get lost among your windows, try the Arrange command.

The Windows menu contains commands that activate, open, close, tile, arrange, and manipulate CodeView windows. There is also a command to view your program's output screen. A bullet appears to the left of the active window when you open this menu.

All the windows are numbered. You can quickly open or switch to a window by pressing ALT plus the window's number.

The following table summarizes the commands on the Windows menu and the corresponding shortcut keys:

Command	Shortcut Key	Purpose
Restore	CTRL+F5	Restores the active window to its size and position before it was maximized or minimized
Move	CTRL+F7	Moves the active window using the keyboard
Size	CTRL+F8	Sizes the active window using the keyboard
Minimize	CTRL+F9	Shrinks the active window to an icon
Maximize	CTRL+F10	Enlarges the active window to full screen
Close	CTRL+F4	Closes the active window
Tile	SHIFT+F5	Arranges all open windows to fill the entire window area
Arrange	ALT+F5	Arranges all open windows to an effective layout for debugging
Help	ALT+0	Opens or switches to the Help window
Local	ALT+1	Opens or switches to the Local window
Watch	ALT+2	Opens or switches to the Watch window
Source 1	ALT+3	Opens or switches to Source window 1
Source 2	ALT+4	Opens or switches to Source window 2
Memory 1	ALT+5	Opens or switches to Memory window 1
Memory 2	ALT+6	Opens or switches to Memory window 2
Register	ALT+7	Opens or switches to the Register window
8087	ALT+8	Opens or switches to the 8087 window
Command	ALT+9	Opens or switches to the Command window
View Output	F4	Swaps the CodeView screen for the program's output screen

Source and Memory Windows

You can open as many as two Source and two Memory windows. At least one Source window must be open at all times. To close a window, use the Close command (CTRL+F4).

Help, Local, Watch, Register, 8087, and Command Windows

CodeView can display one of each of these windows. The Register window has an additional shortcut key (F2) you can use to open or close it.

When you open the Help window, CodeView displays the last Help screen you viewed. If you have not yet viewed Help during the session, CodeView displays the top-level contents in the Microsoft Advisor.

View Output

To view your program's output screen, choose View Output or press ALT+F4. CodeView displays the output screen until you press a key.

The Help Menu

The Help menu contains commands to access the Microsoft Advisor Help system. When you choose a Help command, CodeView opens the Help window if it is not already open and displays the appropriate part of the Microsoft Advisor.

When the Help window is open, you can browse through Help with mouse and keyboard commands. All Microsoft environments provide the same mouse and keyboard commands to access the Microsoft Advisor. For more information on getting the most out of Help, see Chapter 23.

The following table summarizes the commands on the Help menu:

Command	Purpose
Index	Displays the table of Microsoft Advisor indexes
Contents	Displays the Microsoft Advisor contents screen
Topic	Displays Help on the current word
Help on Help	Displays Help on using the Microsoft Advisor
About	Displays CodeView copyright and version information

Index

The Index command displays a table of available indexes. Each part of the Help system has its own index.

Contents

The Contents command (SHIFT+F1) displays the contents for the entire Help system. This screen lists the table of contents for each Help system component.

Topic

The Topic command (F1) displays help on the word at the cursor or the selected text. When you open the Help menu, CodeView displays the topic in the menu. When you choose the Topic command, CodeView displays information on the indicated topic in the Help window.

Help on Help

The Help on Help command displays information on the Microsoft Advisor itself. It describes how the system is organized, how the mouse and keyboard commands are used to browse through Help, and how to use the various kinds of buttons you encounter.

About

The About command displays the CodeView copyright and version information in a dialog box.

10.1 Debugging in Windows

The Microsoft CodeView for Windows debugger (CVW) is a powerful tool for analyzing the behavior of Microsoft Windows programs. With CVW, you can test the execution of your application and examine your application's data. You can isolate problems quickly because you can display any combination of variables—global or local—while you halt or trace your application's execution.

Comparing CVW with CV

The CVW windows, menus, and commands are used in the same way as for CV. See Chapter 9, “The CodeView Environment,” for details on the format of CodeView windows and how to use the windows and menus. Like the MS-DOS CodeView, CVW allows you to display and modify any program variable, section of addressable memory, or processor register. However, CodeView for Windows differs from CV in the following ways:

- Because Windows has a special use for the ALT+/ key combination used by CV to repeat a search, CVW uses CTRL+R.
- CVW tracks your application's segments and data as Windows moves them in memory. Thus, when you refer to an object by name, CVW always supplies the correct address.

CVW also provides six additional Command-window commands for Windows debugging, which are summarized in the following list:

Windows Display Global Heap (**WDG**)

Displays memory objects in the global heap.

Windows Display Local Heap (**WDL**)

Displays memory objects in the local heap.

Windows Dereference Local Handle (**WLH**)

Dereferences a local heap handle to a pointer.

Windows Dereference Global Handle (WGH)

Dereferences a global heap handle to a pointer.

Windows Display Modules (WDM)

Displays a list of the application and DLL modules currently loaded in Windows.

Windows Kill Application (WKA)

Terminates the task that is currently executing by simulating a fatal error.

For details on using these commands, see “CVW Commands” on page 382.

The following CV features are not available in CVW.

- The Print command from the File menu.
- The DOS Shell command from the File menu and the corresponding Shell (!) Command-window command.
- The Screen Swap command from the Options menu and the corresponding Options (OF) Command-window command.

Preparing to Run CVW

Before beginning a CVW debugging session, you must ensure that your system is configured correctly and the Windows application you are going to debug is compiled and linked with the options that generate CodeView debugging information.

For information on setting up your system and configuring CodeView, see “Setting up CodeView” on page 327. For information on preparing programs for use with CodeView, see “General Programming Considerations” on page 322 and “Compiling and Linking” on page 323.

Starting a Debugging Session

Like most Windows applications, CVW can be started in several ways. You can double-click the CVW icon and respond to CVW’s prompts for arguments, or you can run CVW by using the Run command from the Program Manager File menu.

To specify CVW options, choose the Run command from the Program Manager File menu. Windows displays a dialog box where you can enter the appropriate options for your debugging session. For specific information on CodeView command-line syntax and options, see “The CodeView Command Line” on page 336.

You can run CVW to perform the following tasks:

- Debug a single application
- Debug multiple instances of an application

- Debug multiple applications
- Debug dynamic-link libraries (DLLs)

This section describes the methods you use to perform these tasks and summarizes the syntax of the CVW command line for each task.

Starting CVW for a Single Application

After you start CVW from Windows, CodeView displays the Load dialog box.

► To start debugging a single application:

1. Type the name of the application in the File to Debug text box.
CVW assumes the .EXE filename extension if you do not include an extension for the application name. You can also pick the program that you want to debug by choosing it from the Files List box.
2. If you want to specify command-line arguments, move the cursor to the Arguments text box and type the program's command line.
3. Choose OK.
CVW loads the application and displays the source code for the application's **WinMain** routine.
4. Set breakpoints in the code if you desire.
5. Use the Go (G) command (F5) to begin executing the application.

► To avoid the startup dialog boxes:

1. Choose the Run command from the Windows File menu.
2. Type the application name and arguments on the CVW command line. Use the following syntax to start debugging a single application:
CVW *[[options]] appname[.EXE] [[arguments]]*
3. Choose OK.

Starting CVW for Multiple Instances of an Application

Windows can run multiple instances of an application, which can cause problems. For example, each instance of an application might corrupt the other's data. To help you solve such problems, CVW allows you to debug multiple instances of an application. The breakpoints you set in your application apply to all of the instances. To determine which instance of the application has the focus in CVW, examine the DS register.

► **To debug multiple instances of an application:**

1. Start CVW as usual for one instance of your application.
2. Run additional instances of your application by choosing the Run command from the Windows File menu.

You cannot specify the application name more than once on the CVW command line. Any additional application names are passed as arguments to the first application.

Starting CVW for Multiple Applications

You can debug two or more applications at the same time, such as a dynamic data exchange (DDE) client and server.

► **To debug several applications at the same time:**

1. Start CVW as usual for a single application.
2. Choose Load from the Run menu and choose other applications that you also want to debug.
3. Set breakpoints in either or both applications. You can use the Open Module command from the CVW File menu to display the source code for the different modules. If you know the module and the location or function name, you can use the context operator ({ }) to directly set breakpoints in the other applications.
4. Use the Go (G) command (F5) to start running the first application.
5. Choose the Run command from the Windows File menu to start running the second application.

You can also use the /L option on the CVW command line to load the symbols for additional applications, as shown in this example:

```
CVW /Lsecond.exe /Lthird.exe first
```

The /L option and name of each additional application must precede the name of the first application on the command line. You must specify the .EXE filename extension for the additional applications. Repeat the /L option for each application to be included in the debugging session.

Once CVW starts, choose the Run command from the Windows File menu to start executing the additional applications.

Note Global symbols with the same name in several applications (such as WinMain) may not be distinguished. You can use the context operator to specify the exact instance of a symbol.

Starting CVW for DLLs

You can debug one or more DLLs while debugging an application.

► To debug a DLL at the same time as an application:

1. Start CVW as usual for the application.
2. Choose Load from the Run menu and type the name of the DLL.
3. Set breakpoints in the application or DLL. You can use the Open Module command from the CVW File menu to display the source code for the different modules.
4. Use the Go (G) command (F5) to continue executing the application.

You can also use the /L option on the CVW command line to specify the DLLs, as shown in this example:

```
CVW /Lappd11 appname
```

The /L option must precede the name of the application. Repeat the /L option for each DLL you want to debug.

Debugging the LibEntry DLL Initialization Routine CVW allows you to debug the **LibEntry** initialization routine of a DLL. If your application implicitly loads the library, however, a special technique is required to debug the **LibEntry** routine.

An application implicitly loads a DLL if the library routines are imported in the application's module-definition (.DEF) file or if your application imports library routines through an import library when you link the application. An application explicitly loads a DLL by calling the **LoadLibrary** routine.

If your application implicitly loads the DLL and you specify the application in the Command Line dialog box, Windows automatically loads the DLL and executes the **LibEntry** routine when it loads the application. This gives you no opportunity to debug the **LibEntry** routine since it is executed when the application is loaded and before CVW gains control.

To gain control before the **LibEntry** routine is executed, you must set a breakpoint in the **LibEntry** routine before the DLL is loaded.

► To set this breakpoint:

1. In the CVW Load dialog box, provide the name of a "dummy" application that does not load the library instead of the name of your application. The WINSTUB.EXE program is provided for this purpose.
2. Load the DLL by using the Load command from the Run menu.

3. Choose the Open Module command from the CVW File menu and select the module containing the **LibEntry** routine.
4. Set at least one breakpoint in the **LibEntry** routine.
5. Use the Go (**G**) command (F5) to start the dummy application.
6. Run your application using the Run command from the Windows File menu. CVW resumes control when the breakpoint in the **LibEntry** routine is taken.

You can also specify the dummy application and the DLL on the CVW command line.

► **To begin a DLL debugging session from the command line:**

1. Type the command line:

```
CVW /Lmydll winstub
```

2. After CVW starts, follow steps 3 through 6 above to begin debugging.

CVW Commands

CVW recognizes several commands for Windows debugging in addition to the Command-window commands recognized by CV.

These commands allow you to inspect objects in the Windows global and local heaps, list the currently loaded modules, trace and set breakpoints on the occurrence of Windows messages, and terminate the currently executing task.

Windows Display Global Heap

The Windows Display Global Heap (**WDG**) command lists the memory objects in the Windows global heap.

Syntax

WDG [*ghandle*]

If *ghandle* is specified, **WDG** displays the first five global memory objects that start at the object specified by *ghandle*. The *ghandle* argument must be a valid handle to an object allocated on the global heap.

If *ghandle* is not specified, **WDG** displays the entire global heap in the Command window.

Global memory objects are displayed in the order in which Windows manages them, which is typically not in ascending handle order. The output has the following format:

Format *handle address size PDB locks type owner*

Any field may not be present if that field is not defined for the block.

Field	Description
<i>handle</i>	Value of the global memory block handle.
<i>address</i>	Address of the global memory block.
<i>size</i>	Size of the block in bytes.
PDB	Block owner. If present, indicates that the task's Process Descriptor Block is the owner of the block.
<i>locks</i>	Count of locks on the block.
<i>type</i>	The memory-block type.
<i>owner</i>	The block owner's module name.

Windows Display Local Heap

The Windows Display Local Heap (**WDL**) command displays the entire Windows heap of local memory objects. This command's syntax takes no arguments.

Syntax **WDL**

The output has the following format:

Format *handle address size flags locks type heapttype blocktype*

Any field may not be present if that field is not defined for the block.

Field	Description
<i>handle</i>	Value of the global memory block handle
<i>address</i>	Address of the block
<i>size</i>	Size of the block in bytes
<i>flags</i>	The block's flags
<i>locks</i>	Count of locks on the block
<i>type</i>	The type of the handle.
<i>heapttype</i>	The type of heap where the block resides
<i>blocktype</i>	The block's type

Windows Display Modules

The Windows Display Modules (**WDM**) command displays a list of all the DLL and task modules that Windows has loaded. To see a list of known modules, type the **WDM** command in the Command window.

Syntax**WDM**

Each entry in the list is displayed with the following format:

handle refcount module path

Field	Description
<i>handle</i>	The module handle
<i>refcount</i>	The number of times the module has been loaded
<i>module</i>	The name of the module
<i>path</i>	The path of the module's executable file

Watching Windows Messages

You can trace the occurrence of a Windows message or an entire class of Windows messages by using the Breakpoint Set (**BP**) command. You can stop at each message, or you can execute continuously and display the messages in the Command window as they are received.

To trace a Windows message or message class, set a breakpoint using the following options:

Syntax

BP *winproc* [/M{*msgname*|*msgclass*}] [/D]

winproc

Symbol name or address of a window function.

msgname

The name of a Windows message, such as WM_PAINT. The *msgname* is case sensitive.

msgclass

A case-insensitive string of characters that identifies one or more classes of messages to watch. Use the following characters to indicate the class of Windows message:

Class	Type of Windows Message
m	Mouse
w	Window management
n	Input
s	System
i	Initialization
c	Clipboard
d	DDE
z	Nonclient

/D

When specified, CodeView displays the message in the command window, but your program continues executing. The message is displayed similar to the following example:

```
HWND:1c00 wParam:0000 lParam:000000 msg:000F WM_PAINT
```

For each matching message that is sent to the specified *winproc*, CVW lists the hexadecimal values of the window handle (HWND), word parameter (wParam), long parameter (lParam), and message (msg) arguments, along with the name of the message.

You can also specify a pass count and commands to be executed when the breakpoint is taken. For details on the full Breakpoint (**BP**) command syntax, see “BP (Breakpoint Set)” on page 429. Note that you can also use the Breakpoint Set command from the Data menu to set all types of breakpoints.

Windows Kill Application

The Windows Kill Application (**WKA**) command terminates the currently executing task by simulating a fatal error. Since a fatal error terminates the application without performing any of the normal program exit processing, use **WKA** with caution.

To terminate your application, type the following command in the Command window:

WKA

As a result of the simulated fatal error, Windows displays an Unexpected Application Error message box. After you close this message box, Windows may not release subsequent mouse input messages from the system queue until you press a key.

If this happens, the mouse pointer moves on the Windows screen but Windows does not respond to the mouse. After you press any key, Windows responds to the queued mouse events.

The currently executing task is not necessarily your application, so you should use the **WKA** command only when your application is the currently executing task. You can be sure that your application is the currently executing task when CVW shows the current location at a breakpoint in your application.

For more information on using the **WKA** command, see “Terminating Your Program” on page 387.

Syntax

CVW Debugging Techniques

Debugging Windows programs can be a challenging experience. Objects move around in memory. The thread of execution can be a twisting maze where it is difficult to know what code is executing or to control what code in your program is executed.

This section describes the **WLH** and **WGH** commands that you use to examine movable memory objects by their handles. It also describes ways to control your application's execution, how to interrupt and resume debugging your application, how to handle abnormal termination from fatal errors and general protection faults, and how to resume debugging your application after a normal termination.

Dereferencing Memory Handles

In a Windows application, the **LocalLock** and **GlobalLock** routines are used to lock memory handles so that they can dereference them into near or far pointers.

In a debugging session, you may know the memory object's handle. However, you may not know what near or far address the handle references unless you are debugging in an area where the program has just completed a **LocalLock** or **GlobalLock** routine call. To get the near and far pointer addresses for unlocked local and global handles, use the **WLH** and **WGH** commands.

For detailed information on the **WLH** and **WGH** commands, see “WGH (Windows Dereference Global Handle)” on page 463 and “WLH (Windows Dereference Local Handle)” on page 466.

Controlling Application Execution

In CVW, all of the CodeView execution commands (Go, Program Step, Trace, and Animate) can be used to control your application's execution. However, you should keep these restrictions in mind while using CVW:

- Attempting to use the Program Step or Trace commands to execute Windows startup code in Assembly mode causes unpredictable results. To step through your application in Assembly mode, first set a breakpoint at the **WinMain** routine and begin stepping through the program only after the breakpoint is taken.
- Directly calling a Windows application procedure or dialog routine in the Watch window, in the Quick Watch dialog box, or with the Display Expression (?) command can have unpredictable results.

The rest of this section describes techniques and special considerations for controlling program execution in CVW.

Interrupting Your Program There may be times when you want to halt your program immediately. You can interrupt your program by pressing `CTRL+ALT+SYSREQ`. After you press `CTRL+ALT+SYSREQ`, CVW gains control and displays code corresponding to the current CS:IP location. You then have the opportunity to examine registers and memory, set breakpoints and watch expressions, and modify variables. To resume execution, use one of the CodeView program execution commands.

Don't step or trace system code.

You should take care when you interrupt execution. If you interrupt execution while Windows code or other system code is executing, attempting to use the Program Step or Trace commands can produce unpredictable results. When you interrupt execution, it is safest to set breakpoints in your code and then resume continuous execution with the Go command, rather than using the Program Step, Trace, or Animate commands.

For example, an infinite loop in your code presents a special problem. Since you should avoid using the Program Step or Trace commands after interrupting your application, you should try to locate the loop by setting breakpoints in places you suspect are in the loop, then resume continuous execution. When one of these breakpoints is taken, you can be sure that the currently executing code is your application code.

Terminating Your Program At times (such as when your application is executing an infinite loop), you may have to terminate the application. The Windows Kill Application (**WKA**) command terminates the currently executing task. Since this task is not necessarily your application, you should use the **WKA** command only when your application is the currently executing task.

If your application is the currently executing task and is executing a module containing CodeView information, the Source window highlights the current line or instruction. However, if your application contains modules that are compiled without CodeView information, it is more difficult to determine whether the assembly-language code displayed in the Source window belongs to your application or to another task.

Use the WDG command to find out which task terminated.

In this case, use the Windows Display Global Heap (**WDG**) command with the value in the CS register as the argument. CVW displays a listing that indicates whether the code segment belongs to your application.

If the current code is in your application, you can safely use the **WKA** command without affecting other tasks. However, the **WKA** command does not perform all the cleanup tasks associated with the normal termination of a Windows application. For example, global objects created during program execution but not destroyed before you terminated the program remain allocated in the system-wide global heap. This reduces the amount of memory available during the rest of the Windows session. For this reason, you should use the **WKA** command to terminate the application only if you cannot terminate it normally.

Note The **WKA** command simulates a fatal error in your application, causing Windows to display an Unexpected Application Error message box. After you close this message box, Windows may not release subsequent mouse input messages from the system queue until you press a key.

If this happens, the mouse pointer moves on the Windows screen, but Windows does not respond to the mouse. After you press any key, Windows responds to the queued mouse events.

Handling Abnormal Termination of the Application Your application can terminate abnormally in one of two ways while you are debugging it with CVW. It can cause a fatal exit, or it can cause a general protection fault. In both cases, CVW gains control, giving you the opportunity to examine the state of the system when your application terminated. CVW allows you to view registers, display the global and local heaps, display memory, and examine your source code.

Handling a General Protection Fault

If the abnormal termination is caused by a general protection fault (GPF) while executing your application code, CVW displays the line of code where the error occurred. Also, the Command window displays the following message:

```
Trap 13 (0DH) -- General Protection Fault.
```

If the general protection fault occurred while executing Windows code, the CVW Command window displays a stack trace that is useful for finding the error in your source code.

Restarting a Debugging Session

You can terminate your application without leaving CVW. Windows notifies CVW that it is terminating the application, and CVW displays the following message:

```
Program terminated normally (0)
```

The value in parentheses is the return value of the **WinMain** routine. This value is usually the *wParam* parameter of the **WM_QUIT** message, which in turn is the value of the *nExitCode* parameter passed to the **PostQuitMessage** routine.

You can then use the **Go** command to continue the debugging session for additional DLLs or applications. You can also restart the application by using the **Restart** command on the **Run** menu.

10.2 Debugging P-Code

Certain Microsoft compilers can generate space-saving p-code instead of machine code. P-code cannot be run by the processor itself because it is not native machine code. However, when you compile a program into p-code, LINK and the Make P-Code (MPC) utility add an interpreter to your program that reads and interprets p-code instructions.

The interpreter implements a “stack machine.” The p-code instructions generally assume operands on the stack rather than take explicit registers or addresses. Because p-code instructions do not explicitly specify operands, they are extremely small. The trade-off for compact code is reduced execution speed. You use p-code when saving space is more important than speed.

CodeView allows you to debug p-code in the same way you debug native code. At the source level, debugging works the same way for p-code as it does for native code. With CodeView’s p-code execution model, you can view p-code instructions in Mixed and Assembly modes just as you view native machine instructions. The Register window displays the p-code registers and the top eight entries of the p-code stack. If your program contains both p-code routines and native routines, CodeView automatically switches between p-code display and native display. You can also force CodeView to stay in Native mode when you want to view the native machine code of the p-code interpreter itself.

The rest of this section describes:

- How to configure CodeView to use the p-code execution model.
- How to prepare p-code programs for debugging.
- Techniques for debugging p-code.
- Limitations while debugging p-code.

Requirements

To debug a program that contains p-code, make sure you set up CodeView with the p-code execution model. To do so, you will need a **Model** entry under the CodeView tag in TOOLS.INI.

The p-code execution model gives CodeView information about p-code instructions, addressing modes, registers, and so forth, which you need to debug p-code. With this execution model, you can debug p-code just as you can debug native machine code. Without the p-code execution model, you cannot view the source lines for p-code routines, unassemble p-code instructions, or view the p-code registers or stack. For information on the syntax of the **Model** entry, see page 333.

There is a dynamic-link library (DLL) for each p-code execution model, depending on the operating environment. The following list shows the filenames of the DLLs and the environment with which they run:

Filename	Description
NMD1PCD.DLL	Execution model for MS-DOS p-code
NMW0PCD.DLL	Execution model for Windows p-code

Specify the appropriate filename in the **Model** entry. For example, if you are debugging a Windows application that contains p-code, add an entry to the [CVW] section of TOOLS.INI such as:

```
Model:NMW0PCD.DLL
```

The exact syntax can vary, depending on your system configuration and other settings in TOOLS.INI.

Preparing Programs

To debug an application that contains p-code, you must first successfully compile, link, and run the MPC utility on the application. Chapter 3 in the *Programming Techniques* manual explains how to build p-code applications and how to mix p-code with native machine code.

During compilation into p-code, the compiler saves space by using p-code quoting. P-code quoting reduces program size by minimizing repeated sequences of instructions. It replaces all but one of the sequences with a special quote instruction which calls the retained sequence.

Turn off quoting with the /Of option.

Quoting makes debugging difficult because each routine jumps to other routines that contain the quoted instructions. When you compile a program for debugging, specify the /Of option to turn quoting off. When you build a release version of the program, specify /Of to turn quoting back on so that the compiler can generate the smallest possible code.

By default, the compiler sorts local variables by frequency of use. It arranges them on the stack so that the program can access the most frequently used variables with the shortest instructions. This optimization is called frame sorting.

Turn off frame sorting with the /Ov- option.

Frame sorting can make debugging more difficult because local variables do not appear on the stack in the order in which you declared them. You should turn off frame sorting by specifying the /Ov- option to the compiler. When you build a release version, specify /Ov to turn frame sorting on so that the compiler generates the smallest possible code.

P-Code Debugging Techniques

Debugging p-code is like debugging native machine code. If you are examining your program at the instruction level, you should be familiar with the machine's operation. With p-code, this is the stack machine implemented by the p-code interpreter.

For general information on the interpreter and p-code instructions, see Chapter 3, "Reducing Program Size with P-Code" in *Programming Techniques*. For information on the p-code instruction set, choose the P-Code Help button from the Microsoft Advisor's top-level contents. Help is available on each p-code instruction.

When you are debugging native code, you normally view two levels of execution: source code and machine code. P-code introduces another level between the two. You can debug at any of these levels by setting the right combination of Source, Mixed, Assembly, and Native display modes.

The next section shows how to choose the different levels and describes what happens when you trace between native and p-code.

The Native Command

The Native command from the Options menu toggles CodeView's display of native machine code and p-code. When Native mode is turned on, a bullet appears to the left of the command on the menu.

Native mode on

With Native mode turned on, CodeView displays native machine instructions in the Source and Mixed display modes. The Register window and the Register command show the native CPU registers.

Native mode off

With Native mode turned off, CodeView displays:

- Native machine instructions in those parts of your program that contain native code.
- P-code instructions in those parts of your program that contain p-code.

Also, the Register window and the Register command show the native CPU registers when debugging a native routine, and they display the p-code interpreter's registers when debugging a p-code routine.

The distinction between Native mode on and off becomes important when you trace from a native routine into a p-code routine or from a p-code routine to a native routine. Generally, you turn Native mode off to view p-code instructions. Turn Native mode on when you want to see the action of the p-code interpreter.

Tracing From Native Code to P-Code With Native mode turned off, tracing into a p-code routine causes CodeView to display p-code instructions. You can

animate, step, and trace each p-code instruction in your program. You can also set breakpoints at individual p-code instructions. When tracing p-code, the Register window displays the registers and stack of the p-code machine.

With Native mode turned on, tracing into a p-code routine causes CodeView to display the native machine code of the p-code interpreter. Because the p-code interpreter is a library module that does not contain debugging information, CodeView switches to Assembly mode.

Tracing From P-Code to Native Code With Native mode turned off, tracing from a p-code routine to a native routine causes CodeView to display native machine instructions. The Register window displays native CPU registers.

With Native mode turned on, you don't trace from p-code to native code. You trace out of the p-code interpreter and into your program's native code.

Unassembling P-Code

You can use the View Source and Unassemble commands to display p-code instructions in the Source window. With the View Source command, change to Mixed or Assembly display mode. The Unassemble command automatically displays p-code instructions when Native mode is turned off.

CodeView can display p-code and native code in the Source window at the same time. If you use the View Source or Unassemble commands in an area with both p-code and native code, CodeView displays both types of instructions. This commonly occurs when you view a routine with a native entry point as well as a p-code entry point. The different sections of code are separated by the assembly-language **Data** directive.

If you try to unassemble p-code with Native mode turned on, CodeView interprets p-code as native code and displays meaningless instructions.

P-Code Debugging Limitations

While CodeView makes debugging p-code as similar to debugging native machine code as possible, there are some limitations. The following list describes the commands that you cannot use with p-code:

- You cannot assemble p-code instructions.
The Assemble command allows you to assemble instructions at any location in your program, but it accepts only native machine mnemonics. It does not recognize p-code mnemonics. If you accidentally overwrite p-code, use the Restart command. The Restart command restores your program's code.
- You cannot call p-code functions.

With native code, you can use the Display Expression command to call any function. However, the Display Expression command cannot call p-code functions.

10.3 Remote Debugging

Microsoft CodeView versions 4.00 and later support remote debugging. This allows you to debug using two machines. CodeView runs on your development machine (the host), and the program you are debugging runs on another machine (the target). You run a remote monitor program on the target machine to control the program you are debugging. The monitor communicates with CodeView through a serial connection.

Remote debugging isolates CodeView from the program being debugged so that errors in the program do not affect the debugger, and the debugger does not affect the target system. If the program crashes the remote system, your development system continues to run.

The remote monitor demands fewer system resources than the full debugger and has fewer dependencies on the hardware and operating system. It does not use the display, the keyboard, extended memory, or expanded memory. After starting and loading the program to be debugged, it does not use the file system. Therefore, the monitor has no effect on these resources that can change your program's behavior.

You can debug large programs or programs that destabilize the operating system. You can also debug programs on older hardware or smaller systems such as laptops that cannot support the full debugger. Some bugs that you cannot reproduce while running under the full debugger appear under the remote monitor.

The process of debugging a program on a remote machine is almost the same as for local debugging. The only difference is in how you start the session. The following sections describe the hardware and files required for remote debugging and how to configure the debugger components on the host and target machines. Also included are the command-line syntax for the remote monitor and the steps you take to start a remote debugging session.

Requirements

Remote debugging requires two computers. The host system must support the Microsoft C/C++ development system. The target system needs only enough resources to run the remote monitor and your program. You run the MS-DOS CodeView on the host system, and you run either the MS-DOS remote monitor or the Windows remote monitor, according to the type of program you are debugging.

You connect the host and target machines with a null-modem cable plugged into the serial ports on the two machines. A null modem is a serial cable that connects the transmitting line at each end to the receiving line at the opposite end. For CodeView, you can tie all control lines to a TRUE signal. Note that such a cable may not be suitable for use with other software. You cannot use an extension cable with “straight-through” connections.

Any good computer store can assemble a null-modem cable for you with the correct wiring and the appropriate connectors for your host and target machines.

CodeView’s serial transport layers use interrupt-driven input and output, which is supported in MS-DOS only with the COM1 and COM2 ports. Therefore, your machines must be connected using the COM1 or COM2 ports. You can use different ports on the two machines.

If you plan to debug with two machines, you must have the correct files in the correct locations on the host and target. You can start a remote session with a TOOLS.INI file that configures CodeView for local debugging. However, it is recommended that you configure CodeView for remote debugging in TOOLS.INI.

MS-DOS Host Files

For remote debugging, you must have the CodeView debugger CV.EXE and its associated DLLs on the host machine. The SETUP program copies all the required files when you install the development system.

You configure CodeView for remote debugging by setting entries in the TOOLS.INI configuration file. The settings for CodeView appear in the [CV] tagged section of TOOLS.INI. Your settings should specify the DLLs for remote debugging. Most of the entries are the same for local and remote debugging. The only differences are the **Native** and **Transport** entries.

The remote debugging configuration is described in the following table:

Entry	Value	Description
Symbolhandler	SHD1.DLL	MS-DOS symbol handler.
Eval	EED1lang.DLL	Expression evaluator. You must load at least one expression evaluator. Use EED1CAN.DLL for C or MASM. Use EED1CXX.DLL for C++, C, or MASM.
Model	NMD1PCD.DLL	P-code execution model. To debug p-code, you must load the p-code nonnative execution model. Specify this entry only if you are debugging p-code.

Entry	Value	Description
Transport	TLD1COM.DLL	The serial transport layer. (For local debugging, use TLD1LOC.DLL.)
Native	EMD1D1.DLL EMD1W0.DLL	Execution model. The execution model that you use depends on the target. Use EMD1D1.DLL for MS-DOS targets or EMD1W0.DLL for Windows targets.

For more information on configuring CodeView, see “Configuring CodeView with TOOLS.INI” on page 329.

You must have your program’s executable file on both the host and target machines. The program must have the same path on the host and target machines, including drive letter and all directories. The filenames must be identical. For Windows applications, you must also have your application’s DLLs (if any). The DLLs that you want to debug must also have the same path on the host and target machines.

MS-DOS Target Files

For remote debugging of an MS-DOS program, you need the MS-DOS remote monitor RCVCOM.EXE on the target machine along with your program’s executable file. The program must have the same path on the host and target machines, including drive letter and all directories. The filenames must be identical.

You can set default parameters for the remote monitor in the [RCVCOM] section of a TOOLS.INI file on the target machine. For more information, see “Remote Monitor Settings in TOOLS.INI” on page 396.

Windows Target Files

For remote debugging of a Windows application, you need the Windows remote monitor RCVWCOM.EXE and its support DLLs on the target machine along with your application’s executable files. The application and DLL files that you are debugging must have the same path on the host and target machines, including drive letter and all directories. The filenames must be identical.

The Windows remote monitor (RCVWCOM.EXE) and its support DLLs (TOOLHELP.DLL and DMW0.DLL) must be in a directory listed in the PATH environment variable.

You can set default parameters for the remote monitor in the [RCVWCOM] section of a TOOLS.INI file on the target machine. For more information, see “Remote Monitor Settings in TOOLS.INI” on page 396.

Remote Monitor Command-Line Syntax

Syntax { **RCVCOM** | **RCVWCOM** } [[/P *port*:*rate*]] [[/R]]

Option	Description
RCVCOM	Remote monitor for MS-DOS.
RCVWCOM	Remote monitor for Windows.
/P	Parameters. The specified settings override any settings made in TOOLS.INI.
<i>port</i> :	Communications port. Must be COM1: or COM2:. The default setting is COM1:.
<i>rate</i>	Bit rate. Specifies the rate at which to drive the port, up to 19,200 bits per second (bps). To specify a rate, you must also specify a port. There can be no space between <i>port</i> : and <i>rate</i> . You must specify the same bit rate for the host and target. The default rate is 9600 bits per second. The possible rates are 50, 75, 110, 150, 300, 1200, 1600, 1800, 2000, 2400, 3600, 4800, 7200, 9600, and 19,200 bps.
/R	Resident. The monitor stays running when the host debugger exits. When /R is not specified, the monitor terminates when the host debugger exits.

Example To start the remote monitor for several MS-DOS debugging sessions that use the COM2 port at 2400 bits per second, type the command:

```
RCVCOM /P COM1:2400 /R
```

Remote Monitor Settings in TOOLS.INI

You can set default parameters for the MS-DOS and Windows remote monitors in TOOLS.INI. If you do not specify parameters on the command line, the monitors look for TOOLS.INI in the directory specified by the INIT environment variable. You must place the settings for the remote monitors in tagged sections of TOOLS.INI. Settings for RCVCOM appear in the [RCVCOM] tagged section; settings for RCVWCOM appear in the [RCVWCOM] tagged section.

The remote monitors recognize a single **Parameters** entry. The syntax for this entry is:

Syntax **Parameters:** [[*port*:*rate*]]

You specify the *port* and *rate* as for the /P command-line option. The command-line option overrides the TOOLS.INI settings.

Starting a Remote Debugging Session

After the CodeView components are in their locations and properly configured, you can begin a remote debugging session.

► **To start a remote debugging session:**

1. Transfer your program and its DLLs to the target machine.

You can copy the files to a floppy disk, transfer them across a network, or transfer them across the serial line using communications software or serial file-transfer software.

Make sure that the full path of the program on the target machine exactly matches the full path of the program on your host machine. The directory structures for your program's files on the host and target machines must also match exactly. If the paths of the files do not match, the remote monitor is unable to locate the program.

2. Start the remote monitor. If you are debugging a Windows application, double-click the Windows remote monitor icon or use the Run command from the Program Manager File menu. For an MS-DOS program, start the monitor from the command line.

The remote monitor starts and begins polling the communications port. It waits for the host debugger to initiate the debugging session.

3. Start CodeView on the host machine. How you start CodeView depends on your settings in TOOLS.INI.

If you have configured CodeView for remote debugging in TOOLS.INI, you can specify a program on the CodeView command line or use the Load command on the Run menu to load the program. You have already configured the transport layer and execution model.

If you have configured CodeView for local debugging, you can start a remote session as described in the following section, "Starting CodeView for a Different Configuration."

CodeView starts, loads your program, and initiates communication with the remote monitor. You are now ready to debug.

Once the debugging session is started, you can use CodeView just as you would for a local debugging session. When you quit CodeView, the remote monitor quits (unless you specified /R when you started the monitor).

If your system has trouble maintaining the communications link between the host and target machines, reduce the bit rate.

Starting CodeView for a Different Configuration

If you have CodeView configured for local debugging in TOOLS.INI, start CodeView without specifying a program on the command line. This allows you to change CodeView's configuration before it loads your program. It is recommended that you configure CodeView for remote debugging.

► **To start a remote session from a local configuration:**

1. Transfer your program and its DLLs to the target machine.
2. Start the remote monitor on the target machine.
3. On the host machine's command line, start CodeView with the following syntax:

```
CV [[options]]
```

Do not specify the program's filename or arguments.

CodeView starts and displays the Load dialog box. Instead of specifying a program and its arguments, you must first reconfigure CodeView for remote debugging.

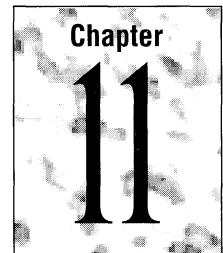
4. Choose Configure Remote. CodeView displays the Configure Remote dialog box. Load the remote transport layer and target execution model, as follows:
 - a. Choose the TLD1COM.DLL transport layer.

Select the communications port and bit rate for the session. Make sure that the bit rate is the same on the host and target machines.
 - b. Choose the execution model for the appropriate target:
 - EMD1D1.DLL for debugging an MS-DOS program.
 - EMD1W0.DLL for debugging a Windows application.
 - c. Choose OK.

CodeView returns to the Load dialog box.
5. Type the name of your program in the File to Debug text box, or select the name in the Files List box. Type your program's command-line arguments in the Arguments text box.
6. Choose OK.

CodeView starts and initiates the remote session.

Using Expressions in CodeView



The arguments to most CodeView commands are expressions. A source-level expression is a reference to a variable or a function call or one or more operations involving constants, variables, addresses, or function calls. A physical location is a register, a memory address or range, or a source-code line number that CodeView maps to an address.

To interpret expressions while maintaining its own programming language independence, CodeView uses dynamic-link libraries (DLLs) to look up symbols, parse, and evaluate expressions. These DLLs are called “expression evaluators.” This release of CodeView has two expression evaluators—one for source-level expressions in Microsoft and ANSI C and the other that handles C++. If you do not specify an expression evaluator, CodeView uses the C++ expression evaluator by default.

The C and C++ expression evaluators recognize most C operators and provide additional debugging operators that are not part of the languages. The C++ expression evaluator places certain restrictions on C++ expressions. Although there is no expression evaluator for the Microsoft Macro Assembler (MASM), the C and C++ expression evaluators support operators that simulate essential assembly-language operations. You use one of these expression evaluators when debugging MASM code.

11.1 Common Elements

When debugging, you use a few common elements in arguments to CodeView commands that are independent of the source language or the current expression evaluator. You often refer to line numbers in source files, and, less often, to lines in text files. You also specify registers and addresses. Some CodeView commands such as Memory Fill (**MF**) accept address ranges.

This section presents the ways to specify line numbers, refer to objects in memory, and use values stored in the processor registers. It also describes the syntax for memory ranges.

Line Numbers

Line numbers are useful for source-level debugging. In Source mode, you see a program displayed with each line numbered consecutively. CodeView allows you to use line numbers to specify the address of code generated for a line or to specify a certain line in a text file.

Syntax

```
[[context]] .linenumber  
[[context]] @linenumber
```

The optional *context* is the context operator used to specify a certain file. When it is omitted, CodeView assumes that the line is in the current source file. The *linenumber* specifies the line in the file (numbered starting at 1). Some commands, such as the Breakpoint Set (**BP**) command, display an error message if the compiler does not generate code for the specified line. For more information on the context operator, see “The Context Operator” on page 406.

With most CodeView commands, the two forms are interchangeable because CodeView automatically maps between source lines and code addresses. The *.linenumber* form specifies a line offset from the beginning of a file. Use this form with the View Source (**VS**) command to display any text file, including files that are not source files for the program you are debugging. The *@linenumber* form specifies the address of the beginning of the code generated by the compiler for the specified line. Use this form with the Breakpoint Set (**BP**) command.

Examples

The following example uses the Go (**G**) command to execute the program from the current location up to line 100. Since no file is indicated, CodeView assumes the current source file.

```
>G @100
```

The following commands use the View Source (**VS**) command to display text files at specific lines as follows: line 10 of the current file, line 301 of EXAMPLE.CPP, and line 22 of TESTFILE.TXT.

```
>VS .10  
>VS {,EXAMPLE.CPP}.301  
>VS {,TESTFILE.TXT}.22
```

Registers

Syntax

```
[[@]]register
```

The *register* is the name of a CPU or p-code 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. However, they are frequently used for lower-level debugging.

When you specify an identifier, CodeView first checks the program's symbol table for the name. If the debugger does not find the name, it checks to see if the name is a register. If you want the identifier to name a register regardless of any name in the symbol table, use an at sign (@) before the register name.

For example, if your program has a symbol called `AX`, specify `@AX` to refer to the `AX` register. You can avoid this conflict by making sure that your program does not use register names as identifiers.

Table 11.1 lists the registers known to CodeView. The p-code registers are available when you are debugging p-code. The 32-bit registers are available on 80386 and 80486 machines when you turn the 386 option on.

Table 11.1 Registers

Register Type	Register 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
16-bit high word*	TH
16-bit low word*	TL
Quoting*	PQ
32-bit general purpose†	EAX, EBX, ECX, EDX
32-bit pointer†	ESP, EBP
32-bit index†	ESI, EDI

* Available only when debugging p-code

† Available only when 386 option is turned on

Addresses

Syntax

`[[context]][[symbol]]`

The *context* is the context operator and specifies the point at which to begin searching for *symbol*. If *context* is omitted, the current location is used. The *symbol* is a label, variable, or function name.

Syntax `[[context]][[@linenumber]]`

The *linenumber* is the number of a line in the specified file. If *context* is omitted, CodeView assumes the current file. Line numbers start at 1.

Syntax `[[segment:]]offset`

A full address is a *segment* and an *offset*, separated by a colon. The *segment* and *offset* can be numeric expressions, symbols, or register names. A partial address has only an *offset*; CodeView assumes a default segment address, depending on the command. Commands that refer to data (Memory Dump, Memory Enter, for example) assume the value of the data segment (DS) register. Commands that refer to code (such as Assemble, Breakpoint Set, and Go) assume the value of the code segment (CS) register.

In source-level debugging, full *segment:offset* addresses are seldom necessary. Occasionally they may be convenient for referring to addresses outside the program, such as display memory.

Examples

In the following example, the Memory Dump Bytes (**MDB**) command dumps memory starting at offset address 100. Since no segment is given, the data segment (the default for Memory Dump commands) is assumed.

```
>MDB 100
```

In the following example, the **MDB** command dumps memory starting at the address of the element `array[COUNT]`.

```
>MDB array[count]
```

In the following example, the Unassemble (**U**) command shows a disassembly of memory starting at a point 10 bytes beyond the symbol `label`.

```
>U label+10
```

In this example, the **MDB** command dumps memory at the address having the segment value in the ES register and the offset address 200 in the current radix.

```
>MDB ES:200
```

Address Ranges

Syntax `start [[end]]`

A range is a pair of addresses that defines the boundary of a sequence of contiguous memory locations. You can specify a range by giving the starting address and

the ending address. In this case, the range covers *start* to *end*, inclusively. If a command takes a range but you do not supply a second address, CodeView displays enough data to fill the current size of the window.

Syntax

start **L** *length*

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 a “length range.” In a length range, *start* is the address of the first object, **L** indicates that this is a length range, and *length* specifies the number of objects in the range.

The size of the object is the size taken by the command. For example, the Memory Dump Bytes (**MDB**) command dumps bytes, the Memory Dump Words (**MDW**) command dumps words, the Unassemble (**U**) command unassembles instructions (which can vary in size), and so on.

Examples

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

```
>MDB buffer
```

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

```
>MDB buffer buffer+20
```

The following example uses a length range to dump the same range of memory as in the previous example.

```
>MDB buffer L 20
```

The following example uses the Memory Fill (**MF**) command to fill memory with dollar sign (\$) characters starting 30 bytes before `right_half` and continuing to `right_half`.

```
>MF right_half-30 right_half '$'
```

11.2 Choosing an Expression Evaluator

CodeView loads all the expression evaluators that you specify with **Eval** entries in `TOOLS.INI`. However, you need to load only one expression evaluator for most debugging tasks. This section discusses how to choose the appropriate one for your debugging environment.

If you place more than one **Eval** setting in `TOOLS.INI`, CodeView loads all the expression evaluators. You can specify the active evaluator by using the Language command on the Options menu or with the **USE** command. By default, CodeView automatically selects the appropriate expression evaluator based on the current source file's extension. For more information on the Language command, see page 370. For details on the **USE** command, see page 452. For information on the **Eval** entry and complete instructions for configuring CodeView, see "Setting Up CodeView" on page 327.

When you are debugging C or MASM source code, you can normally use either the C or the C++ expression evaluator. C++ is mostly a superset of C at the expression level, and both evaluators support operators for debugging MASM code. Therefore, CodeView loads the C++ expression evaluator by default when no other expression evaluators are specified.

However, you might want to use only the C expression evaluator. If you are debugging C or MASM source code, it is recommended that you specify only the C expression evaluator. If your C program uses C++ keywords as variable, function, or label names, you must use the C expression evaluator. C variable names that are C++ keywords are not recognized as variables by the C++ expression evaluator. The C++ expression evaluator requires more memory than the C evaluator. Therefore, load only the C expression evaluator when running CodeView in an environment with limited memory.

You must use the C++ expression evaluator to debug C++ because the C evaluator does not recognize C++ expressions or keywords and cannot translate the decorated names produced by the C++ compiler. If you try to debug a C++ application with the C expression evaluator, C++ expressions generate an error, and you must use the decorated symbol names. For more information on decorated names, see Appendix B.

11.3 Using the C and C++ Expression Evaluators

When you specify the C or C++ expression evaluator, you can use most Microsoft or ANSI C and many Microsoft C++ expressions as arguments to CodeView commands.

CodeView evaluates C and C++ expressions according to the same rules as the compiler, including operator associativity and order of precedence. There are, however, a few additional operators and some exceptions to the standard syntax. See the *C Language Reference* and *C++ Language Reference* for descriptions of C and C++ expression syntax.

Additional Operators

Both expression evaluators support the following additional operators:

- The “context” operator (`{ }`) to specify the context of a symbol.
- The colon operator (`:`) to form addresses. The colon operator has the same precedence as the multiplication, division, and remainder operators.
- The memory operators (**BY**, **WO**, and **DW**) to access memory. Each of the memory operators has the same precedence, which is the lowest of any operator recognized by the expression evaluators.

The colon and memory operators are used mostly to debug assembly-language code. For information about the colon operator, see “Addresses” on page 401. The memory operators are described in “Memory Operators” on page 412. For more information about using the context operator, see “The Context Operator” on page 406.

Unsupported Operators

The comma operator (`,`) and the conditional (`?:`) operator are not supported by the C and C++ expression evaluators. The C++ operators `.*` and `->*` are also unsupported.

The ampersand (`&`) is not supported as a bitwise AND operator. However, both expression evaluators recognize the ampersand (`&`) as an address-of operator. The C++ expression evaluator also recognizes the ampersand in type casts to create a reference type. For example, `(int &)curIndex` casts the `curIndex` variable to an **int** reference type.

Restrictions and Special Considerations

When you are debugging C and C++ programs, the following general restrictions apply:

- When you use an expression as an argument to a CodeView command that takes multiple arguments, such as the Memory Fill (**MF**) command, the expression cannot contain spaces. For example, `&count+6` is allowed, but `&count + 6` is interpreted as three separate arguments. Some commands, such as the Display Expression (**?**) command, permit spaces in expressions.
- CodeView command names are not case sensitive, but C and C++ identifiers are case sensitive unless you turn off case sensitivity with the commands on the Options menu or the Options (**O**) command.
- You cannot call an intrinsic function or an inlined function in a CodeView expression unless it appears at least once as a normal function.

- CodeView limits casts of pointer types to one level of indirection. For example, `(char *)sym` is accepted, but `(char **)sym` is not. An expression such as `char far *(far *)` is also not supported.
- The C++ scope operator (`::`) has lower precedence in CodeView expressions than in the C++ language. In CodeView, its precedence falls between the base (`:`) and postfix (`++`, `--`) operators and the unary operators (`!`, `&`, `*`, for example). In C++, it has the highest precedence.

CodeView imposes additional restrictions on C++ expressions. These restrictions and other special considerations when debugging C++ are described in “Using C++ Expressions” on page 409.

The Context Operator

The context operator (`{ }`) is unique to CodeView. It is not part of the C or C++ languages. You use it to specify the exact context of an expression or line number that appears in more than one place in your source code. For example, you might use this operator to specify a symbol defined in an include file when the file is included more than once or to specify a name in an outer scope that is otherwise hidden by a local name.

When you use a symbol in a CodeView expression, the C and C++ expression evaluators search for that symbol in the following order:

1. Lexical scope outward. The expression evaluator searches for the symbol starting with the current block (a series of statements enclosed in curly braces) and continuing with the enclosing block. The current block is the code containing the current location (CS:IP address).
2. Function scope. The expression evaluator searches for the symbol in the current function.
3. Class scope. If debugging C++ and within the scope of a member function, the expression evaluator searches symbols of that member function’s class and all its base classes. The C++ expression evaluator uses the normal dominance rules.
4. Current module. The expression evaluator searches all symbols in the current module.
5. Global symbols. The expression evaluator searches all global symbols in the program.
6. Other modules. The expression evaluator searches the global symbols in all other modules in the program.
7. Public symbols in the program.

If the name is not found in any of these places, and the name is not the name of a register, CodeView displays an error message.

The context operator lets you specify the starting point of the search and bypass the current location. Note that you cannot specify a block because a block has no name. You cannot specify a class, but you can specify a member function of the class and let the expression evaluator search outward.

Syntax

```
{[[function]],[[module]],[[dllexe]]} [[object]]
```

function

The name of a function in the program.

module

The name of a source file. You must specify a source file if the function is outside the current scope. If the file is not in the current directory, you must specify the path.

dllexe

The path of a program's DLL or .EXE file.

object

A line number or symbol.

The context operator has the same precedence and associativity as the type-cast operator. You can omit *function*, *module*, or *dllexe*, but you must specify all leading commas. You can omit trailing commas. If a name contains a comma, you must enclose the name in parentheses.

Example

The following example displays the value of the variable `Pos`, which is local to the function `make_box`, which is defined in the source file `DRAWBOX.C`. Assuming that there is more than one source file called `DRAWBOX.C`, the third parameter specifies that the source file containing the function `make_box` is the one used by `DISPTXT.DLL`.

```
? {make_box,C:\TREE1\DRAWBOX.C,C:\TREE2\DISPTXT.DLL}Pos
```

Numeric Constants

Numbers used in CodeView commands represent integer constants. They are expressed in octal, hexadecimal, or decimal radix; the default is the current radix. The default input radix for the C expression evaluator is decimal. However, you can use the Radix (N) command to specify a different radix, as explained on page 444. CodeView displays the current radix in the lower-right corner of the status bar.

To override the current radix, you can use the C and C++ syntax for entering a constant of a different radix. In addition, CodeView supports the **Ondigits** syntax to specify decimal numbers independently of the current radix.

The following table summarizes the syntax for different radices:

Syntax	Radix
<i>digits</i>	Current radix
0 <i>digits</i>	Octal (base 8)
0n <i>digits</i>	Decimal (base 10)
0x <i>digits</i>	Hexadecimal (base 16)

Symbols take priority over numbers.

When hexadecimal is the current radix, it is possible to enter a value that could be either a symbol or a hexadecimal number. CodeView resolves the ambiguity by first searching for a symbol with that name. If no symbol is found, the value is a number. If you want to enter a number that is the same as a symbol in your program, use the explicit hexadecimal format (**0x***digits*).

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

String Literals

Syntax

"string"

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

A string that you specify in a CodeView command is volatile, and you cannot rely on its existence for longer than the execution of the command. This means that you can pass a string to a function, but you cannot assign a string to a character pointer variable. For example, the command:

```
? pChar = "string"
```

is not valid. However, you can change a pointer to refer to a different string in your program. Also, if the pointer addresses a section of memory large enough to accommodate the string, you can use the Memory Enter (**ME**) command to fill the memory with a new string.

Symbol Formats

For modules that are compiled with full CodeView debugging information (`/Zi`), the expression evaluators automatically translate the decorated names into source form. You specify and view names as they appear in your source. Therefore, debugging is easier when all modules in the program are compiled with full

CodeView debugging information. For large programs, however, you may need to compile some modules to include only line numbers and public symbols (`/Zd`).

CodeView accepts and displays public symbol names as “decorated” names. The decorated name is the form of the name in the object code produced by the compiler. Public symbols are names in library routines or names in modules compiled without CodeView information (that is, compiled with the `/Zd` option, or compiled without any line or symbolic information and linked with the `/CO` option).

To get a listing of all names in their decorated and undecorated forms, specify the `LINK /MAP:FULL` option.

Name decoration is the mechanism used to enforce type-safe linkage. This means that only the names and references with precisely matching spelling, case, calling convention, and type are linked together.

Names declared with the C calling convention (either implicitly or explicitly using the `_cdecl` keyword) begin with an underscore (`_`). For example, the function `main` can be displayed as `_main`. Pascal names are converted to uppercase and have no prefix. Names declared as `_fastcall` are converted to uppercase and begin with an at sign (`@`).

For C++, the decorated name encodes the type of the symbol in addition to the calling convention. This form of the name can be long and difficult to read. The name begins with at least one question mark (`?`). For C++ functions, the decoration includes the function’s scope, the types of the function’s parameters, and the function’s return type.

For more information on decorated names, see Appendix B.

11.4 Using C++ Expressions

The C++ expression evaluator accepts almost all C++ expressions, with some restrictions and some additions. This section describes these special considerations.

Access Control

All class members are accessible.

You can examine any member of a class object including base classes and embedded member objects. In CodeView, all members are available without regard to access control (public, protected, or private visibility). For example, if `myDate` has a private data member named `month`, you can examine it with the following command:

```
>? myDate.month
3
```


Ambiguous References

If an expression makes an ambiguous reference to a member name, you must use the class name to qualify it. For example, suppose that class `C` inherits from both class `A` and class `B`, and that `A` and `B` define a member function named `expand`. If `Cthing` is an instance of class `C`, the following expression is ambiguous:

```
Cthing.expand()
```

The following expression resolves the ambiguity and uses `B`'s `expand` function:

```
Cthing.B::expand()
```

The C++ expression evaluator applies normal dominance rules regarding member names to resolve ambiguities.

Inheritance

When you display a class object that has virtual base classes, the members of the virtual base class are displayed for each inheritance path, even though only one instance of those members is stored. You can access members of an object through a pointer to the object, and you can call virtual functions through a pointer.

For example, when the `Employee` class defines a virtual function that is named `computePay`, which is redefined in the class that inherits from `Employee`, you can call `computePay` through a pointer to `Employee` and have the proper function executed:

```
>? empPtr->computePay()
```

You can cast a pointer to a derived class object into a pointer to a base class object; the reverse conversion is not permitted. For example, if the class `List` is derived from the class `Collection`, the cast `(Collection *)pListCustomer` is valid, but the cast `(List *)pCollectClients` is illegal.

Constructors, Destructors, and Conversions

You can set a breakpoint on a class's constructor or a destructor (unless they are inline functions). The breakpoint is taken whenever an object of that class is created or destroyed. You can specify a breakpoint that halts execution so that you can examine your program's status. You can also specify a breakpoint that executes a command, such as displaying a message in the Command window or incrementing a counter, and then continues execution. This technique is especially useful for monitoring the creation and destruction of temporary objects created by the compiler.

You cannot call a constructor or destructor for an object, either explicitly or implicitly, by using an expression that calls for construction of a temporary object. For example, the following illegal command explicitly calls a constructor and results in an error message:

```
>? Date( 2, 3, 1985 )
```

You cannot call a conversion function if the destination of the conversion is a class because such a conversion involves the construction of an object. For example, suppose that `myFraction` is an instance of the `Fraction` class, which defines the conversion function `operator FixedPoint`. The following command results in an error:

```
>? (FixedPoint)myFraction
```

However, you can call a conversion function if the destination of the conversion is a built-in type. For example, suppose that the `Fraction` class defines a conversion function named `operator float`. The following command is legal:

```
>? (float)myFraction
```

You can also call functions that return an object or that declare local objects.

You cannot call the **new** or **delete** operators. The command

```
? pDate = new Date(2,3,1985)
```

is illegal and CodeView displays an error message.

Overloading

You can call overloaded functions as long as there exists an exact match or a match that does not require a conversion involving the construction of an object. For example, if the `calc` function takes a `Fraction` object as a parameter, and the `Fraction` class defines a single-argument constructor that accepts an integer, the following command results in an error:

```
>? calc( 23 )
```

Even though a legal conversion exists to convert the integer into the `Fraction` object that the `calc` function expects, such a conversion involves the creation of an object and is not supported.

Operator Functions

Operator functions for a class can be invoked implicitly or explicitly. For example, suppose that `myFraction` and `yourFraction` are instances of a class that defines **operator+**. You can display the sum of those two objects using expression syntax:

```
>? myFraction + yourFraction
```

You can also use the functional notation to call an operator function:

```
>? myFraction.operator+( yourFraction )
```

If an operator function is defined as a friend, you can call it implicitly using the same syntax as for a member function, or you can invoke it explicitly, as follows:

```
>? operator+( myFraction, yourFraction )
```

Note that operator functions, like ordinary functions, cannot be called with arguments that require a conversion involving the construction of an object.

11.5 Debugging Assembly Language

The Microsoft Macro Assembler (MASM) versions 5.0 and later provide type and size information for CodeView. With this information, CodeView can correctly evaluate expressions derived from assembly code (except for arrays, which require a different syntax, as discussed later in this section).

You can use either the C or C++ expression evaluators for debugging assembly language; CodeView does not have an assembly-language expression evaluator. The C and C++ expression evaluators provide special operators to simulate essential MASM operations.

You cannot always specify an expression in CodeView exactly as it would appear in assembly-language source code. You have to write an equivalent CodeView expression. This section describes the CodeView equivalents for MASM expressions.

Memory Operators

A memory operator is a unary operator that returns the result of a direct memory operation. The memory operators are **BY**, **WO**, and **DW**. The C and C++ expression evaluators add the memory operators to the operators in the C and C++ languages. The memory operators are used mainly to debug assembly-language code.

Syntax `{BY | WO | DW} address`

The **BY** operator returns a short integer that contains the first byte at *address*. This operator simulates **BYTE PTR**.

The **WO** operator returns a short integer that contains the value of the word (two bytes) at *address*. This operator simulates the Microsoft Macro Assembler **WORD PTR** operation. The **DW** operator returns a long integer that contains the value of the first four bytes at *address*. This operator simulates **DWORD PTR**.

The examples that follow use the Display Expression (?) command, which is described on page 477. The *x* format specifier used in some of these examples causes the result to be displayed in hexadecimal.

Examples

The following example displays the first byte at the address of the variable `sum`.

```
>? BY sum
101
```

The following example displays the byte pointed to by the BP register with a displacement of 6.

```
>? BY bp+6,x
0042
```

The following example displays the first word at the address of the variable `new_set`.

```
>? W0 new_set
13120
```

The following example displays the word pointed to by the stack pointer (the last word pushed onto the stack). Because the stack pointer (SP) offset register is used with no segment address, the stack segment (SS) register is assumed.

```
>? W0 sp,x
2F38
```

The following example displays the doubleword at the address of `sum`.

```
>? DW sum
132120365
```

The following example displays the doubleword pointed to by the SI register. Because the SI index register is used without specifying a segment address, the DS register is assumed.

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

Register Indirection

The C expression evaluator does not recognize brackets ([]) to indicate a memory location pointed to by a register. Instead, you use the **BY**, **WO**, and **DW** operators to reference the corresponding byte, word, or doubleword values.

MASM Expression	CodeView Equivalent
BYTE PTR [bx]	BY bx
WORD PTR [bp]	WO bp
DWORD PTR [bp]	DW bp

Register Indirection with Displacement

To perform based, indexed, or based-indexed indirection with a displacement, use the **BY**, **WO**, and **DW** operators with addition.

MASM Expression	CodeView Equivalent
BYTE PTR [di+6]	BY di+6
BYTE PTR Test[bx]	BY &Test+bx
WORD PTR [si][bp+6]	WO si+bp+6
DWORD PTR [bx][si]	DW bx+si

Address of a Variable

Use the C address-of operator (&) instead of the MASM **OFFSET** operator.

MASM Expression	CodeView Equivalent
OFFSET Var	&Var

PTR Operator

Use type casts or the **BY**, **WO**, and **DW** operators with the address-of operator (&) to replace the assembly-language **PTR** operator.

MASM Expression	CodeView Equivalents
BYTE PTR Var	BY &Var *(unsigned char*)&Var
WORD PTR Var	WO &Var *(unsigned *)&Var
DWORD PTR Var	DW &Var *(unsigned long*)&Var

Strings

Add the string format specifier `,s` after the variable name.

MASM Expression	CodeView Equivalent
String	String,s

Because C strings end with a null (ASCII 0) character, CodeView displays all characters from the first byte of the variable up to the next null byte in memory when you request a string display. If you intend to debug an assembly-language program, and you want to view strings in the Watch window or with the Display Expression (?) command, you should delimit string variables with a null character. You can also view null-terminated or unterminated strings in a Memory window or with the Memory Dump ASCII (MDA) command.

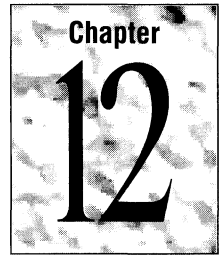
Array and Structure Elements

Prefix an array name with the address-of operator (&) and add the desired offset. The offset can be an expression, number, register name, or variable.

The following examples (using byte, word, and doubleword arrays) show how to do this.

MASM Expression	CodeView Equivalents
String[12]	BY &String+12 *(&String+12)
aWords[bx+di]	WO &aWords+bx+di *(unsigned*)(&aWords+bx+di)
aDWords[bx+4]	DW &aDWords+bx+4 *(unsigned long*)(&aDWords+bx+4)

CodeView Reference



This chapter describes the CodeView Command-window command format, explains the common items in CodeView expressions, and summarizes all Command-window commands in a convenient table. The final section describes each command in detail. The nonalphabetic commands appear at the end of the chapter.

12.1 Command-Window Command Format

Syntax *command* [[*arguments*]] [; *command* [[*arguments*]]]

Parameters *command*
A command name. The *command* is not case sensitive; any combination of uppercase and lowercase letters can be used.

arguments
Expressions that represent values or addresses used by the command. Source-level expressions used as arguments may or may not be case sensitive, depending on the current expression evaluator. The first argument can be placed immediately after *command* with no space separating the two fields.

If a command takes more than one argument, you must separate the arguments with spaces.

Remarks Additional commands may be specified on the same line. A semicolon (;) must separate each command from the next.

12.2 CodeView Expression Reference

When debugging, you use a few common elements in arguments to CodeView commands that are independent of the source language or the current expression evaluator. You often refer to line numbers in source files and, less often, to lines

in text files. You also specify registers and addresses. Some CodeView commands such as Memory Fill (**MF**) accept address ranges.

This section presents the ways to specify line numbers, refer to objects in memory, and use values stored in the processor registers. It also describes the syntax for memory ranges. Moreover, the context operator, which you use to specify the point at which to begin searching for a symbol, is summarized. For detailed information on the context operator and CodeView expressions, see Chapter 11.

Line Numbers

Syntax

`[[context]]@linenumber`
`[[context]].linenumber`

Description

Line numbers are useful for source-level debugging. They correspond to the lines in source-code files. In source mode, a program is displayed with each line numbered sequentially. The CodeView debugger allows you to use these numbers to access parts of a program.

The memory address of the code corresponding to a source-line number is specified as:

`@linenumber`

The actual file line number is:

`[[context]].linenumber`

CodeView assumes that the source line is in the current source file. To specify a source line in a different file, you must specify the line's context using the context operator:

`{,file}@linenumber`

CodeView displays an error message if *file* does not exist or no source line exists for *linenumber*.

Examples

The following example uses the View Source (**VS**) command to display code starting at source line 100. Since no file is indicated, the current source file is assumed.

```
>VS @100
```

This next example uses **VS** to display source code starting at line 301 of the file DEMO.C.

```
>VS {,demo.c}.301
```

Registers

Syntax `[[@]]register`

A register name represents the current value stored in the register. Table 12.1 summarizes the register names known to the CodeView debugger.

Table 12.1 Register Names

Register Type	Register 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
16-bit high word*	TH
16-bit low word*	TL
Quoting*	PQ
32-bit general purpose†	EAX, EBX, ECX, EDX
32-bit pointer†	ESP, EBP
32-bit index†	ESI, EDI

* Available only when debugging p-code

† Available only when 386 option turned on

To force a symbol to represent a register, prefix the symbol with an at sign (@). For example, to make `AX` represent a register rather than a variable, use `@AX`.

Addresses

Syntax `[[context]]@linenumber`
`[[context]][[segment:]]offset`
`register:offset`
`[[context]]function`
`[[context]]symbol`

Description If only an *offset* is specified, the *segment* is determined by the command in which the address appears. Commands that refer to data (Memory Dump, Memory Enter)

use the segment in the DS register. Commands that refer to code (Assemble, Breakpoint Set, Go, Unassemble, and View Source) use the segment in the CS register.

The Display Expression (?) and Add Watch Expression (W?) commands interpret numeric arguments as constants rather than as offsets. However, if you cast the argument to a pointer, as in

```
W? (char *)0
```

the argument is treated as an offset from DS.

Address Ranges

Syntax

```
start end  
start L count
```

Description

An address range is a pair of memory addresses that specify the higher and lower boundaries of contiguous memory. You can specify a range in two ways:

- Give the starting and ending addresses:

```
start end
```

The range covers *start* to *end*, inclusively. If you don't supply an ending address, CodeView assumes the default range. Each command has its own default range; the most common default range is 128 bytes.

- Give the starting address and the number of objects you want included in the range:

```
start L count
```

This type of range is called an “object range.” The starting address is the address of the first object in the list, and *count* specifies the number of objects in the range. The way the size of an object is measured depends on the command. For example, the Memory Dump Bytes (MDB) command has byte objects, the Memory Dump Words (MDW) command has words, the Unassemble (U) command has instructions, and so on.

Examples

This example 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 Memory Dump Bytes command) is assumed.

```
MDB buffer
```

The following example dumps 21 bytes starting at `buffer` and ending at `buffer+20` (the point 20 bytes beyond `buffer`).

```
MDB buffer buffer+20
```

The following example uses an object range to dump a range of memory. The `L` indicates that the range is an object range, and `20` indicates the number of objects in the range.

```
MDB buffer L 20
```

Here, each object has a size of 1 byte since that is the size of objects dumped by the Memory Dump Bytes (**MDB**) command.

Context Operator ({})

Syntax { *[[function]]* , *[[module]]* , *[[dllexe]]* } *[[object]]*

Parameters

function

The name of a function or procedure in the program. Case is significant for case-sensitive languages.

module

The name of a source file. If the file is not in the current directory, you must specify the path.

dllexe

The full path of a dynamic-link library (DLL) in the program or the program's .EXE file.

object

A variable name, line number, or expression.

Description

The context operator specifies the exact starting point to search for a symbol or line. You apply it the same way as a type cast is applied in C. When you do not use the context operator, the current context (CS:IP) is used.

You can omit *function*, *module*, or *dll*, but all leading commas must be given. Trailing commas can be omitted. If a name contains a comma, the name must be enclosed in parentheses.

For complete information on the context operator, see “The Context Operator” on page 406.

Example

This example displays the value of the variable `Pos`, which is local to the function `make_box` defined in the source file `BOXDRAW.C`.

```
? {make_box,C:\PROJ\boxdraw.c}Pos
```

12.3 CodeView Command Overview

Table 12.2 summarizes the CodeView Command-window commands. The next section describes each command in detail.

Table 12.2 CodeView Command Summary

Command	Name	Description
A	Assemble	Inserts assembly-language instructions
BC	Breakpoint Clear	Clears one or more breakpoints
BD	Breakpoint Disable	Disables one or more breakpoints
BE	Breakpoint Enable	Enables one or more breakpoints
BL	Breakpoint List	Lists all breakpoints
BP	Breakpoint Set	Sets a breakpoint
E	Animate	Executes the program in slow motion
G	Go	Executes the program
H	Help	Provides Help information
I	Port Input	Reads a byte from a hardware port
K	Stack Trace	Displays active routines K command
L	Restart	Restarts the program
MC	Memory Compare	Compares two blocks of memory byte by byte
MD	Memory Dump	Displays sections of memory in the Command window in various formats
ME	Memory Enter	Modifies memory
MF	Memory Fill	Fills a block of memory
MM	Memory Move	Copies one block of memory to another
MS	Memory Search	Scans memory for specified byte values
N	Radix	Changes current radix for entering arguments and displaying values
O	Options	Views or sets options
O	Port Output	Outputs a byte to a hardware port
P	Program Step	Executes the current line and steps over functions

Table 12.2 *(continued)*

Command	Name	Description
Q	Quit	Terminates CodeView
R	Register	Displays the values of registers and flags and optionally changes them
T	Trace	Executes the current line and traces into functions
T	Trace Speed	Specifies speed for the Animate command
U	Unassemble	Displays assembly-language instructions
USE	Use Language	Specifies the active expression evaluator
VM	View Memory	Displays sections of memory in a Memory window in various formats
VS	View Source	Displays source code in varying formats in a Source window
W?	Add Watch	Sets an expression to be watched
WC	Delete Watch	Deletes one or more watch expressions
WDG	Windows Display Global Heap	Displays memory objects in the global heap
WDL	Windows Display Local Heap	Displays memory objects in the local heap
WDM	Windows Display Modules	Displays a list of the applications and DLL modules known by Windows
WGH	Windows Dereference Global Handle	Dereference a global handle
WKA	Windows Kill Application	Terminates the current task by simulating a fatal error
WL	List Watch	Lists current watch expressions
WLH	Windows Dereference Local Handle	Dereference a local handle
X	Examine Symbols	Displays the addresses and types of symbols
!	Shell Escape	Runs an MS-DOS shell
"	Pause	Interrupts execution of redirected commands and waits for keystroke
#	Tab Set	Sets number of spaces for each tab character
*	Comment	Displays explanatory text during redirection
.	Current Location	Displays the current location
/	Search	Searches for a regular expression in the source
7	8087	Shows the values of the 8087 or emulator registers and flags
:	Delay	Delays execution of redirected commands

Table 12.2 (continued)

Command	Name	Description
<	Redirect Input	Reads input from specified device
>	Redirect Output	Sends output to specified device
=	Redirect Input and Output	Sends output and reads input from specified device
?	Display Expression	Evaluates and displays expressions or symbols
??	Quick Watch	Displays variables and data structures in a dialog box
@	Redraw	Redraws the screen
\	Screen Exchange	Exchanges the CodeView and output screens

12.4 CodeView Command Reference

The rest of this chapter is an alphabetical reference to all CodeView Command-window commands. Nonalphabetic commands such as the Pause (") command are listed after the alphabetic reference.

A (Assemble)

Syntax A [[*address*]]

Parameter *address*
 Begins assembly at this address. If *address* is not given, assembly begins at the current assembly address (see below).

Description The Assemble (A) command assembles 8086-family (8086/87/88, 80186/286, 80287/387, and 80286/386/486 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 80386/486 protected-mode mnemonics. In addition, the CodeView debugger can assemble 80286 instructions that use the 32-bit 386/486 registers.

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 or the address pointed to by CS:IP. 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 Animate), the assembly address is reset to the current address.

Entering Instructions

Use the following procedure to assemble instructions:

1. Type the Assemble (**A**) command in the command window and press **ENTER**. CodeView displays the assembly address and waits for you to enter a new instruction.
2. Type a mnemonic and press **ENTER**. CodeView assembles the instruction into memory and displays the next available address. If an instruction you enter contains a syntax error, CodeView displays the message:

```
^ Syntax error
```

Then CodeView redisplay the current assembly address and waits for you to enter a correct instruction. The caret (^) in the message points to the first character that CodeView could not interpret.

3. Continue entering new instructions until you have assembled all the instructions you want.
4. Press **ENTER** without entering any mnemonic to conclude assembly and return to the CodeView prompt.

Remarks

Consider the following principles when you enter instruction mnemonics:

- The far-return mnemonic is **RETF**.
- 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.
- CodeView 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. The **NEAR** prefix can be abbreviated to **NE**, but the **FAR** prefix cannot be abbreviated.
- CodeView 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**.
- CodeView cannot determine whether an operand refers to a memory location or to an immediate operand. CodeView uses the convention that operands enclosed in brackets ([]) refer to memory.
- CodeView supports all forms of indirect register instructions.

- All instruction-name synonyms are supported. If you assemble instructions and then examine them with the Unassemble (U) command, CodeView may show synonymous instructions, rather than the ones you have assembled.
- Do not assemble and execute 8087/287 instructions if your system is not equipped with a math coprocessor chip.

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 Restart command. The original code is reloaded, possibly writing over parts of memory that you have changed.

Example

This example places two new instructions in a program, replacing any instructions already there.

```
>a 0x47:0xb3e
0001:0B3E  mov  ax,bx
0001:0B40  mov  si,0x9ce
0001:0043
```

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.

BC (Breakpoint Clear)

Syntax **BC** [*list* | *start-end* | *]

Parameters

list

List of breakpoints to be removed, with breakpoint numbers separated by spaces. A number identifies each breakpoint. You can use the Breakpoint List (BL) command to display currently set breakpoints and their numbers.

start-end

Range of breakpoints to clear. The command clears breakpoints numbered from *start* to *end*, inclusive.

*

Removes all currently set breakpoints.

Description

The Breakpoint Clear (BC) command permanently removes one or more previously set breakpoints.

Mouse and Keyboard

In addition to typing the **BC** command, you can clear breakpoints with the following shortcuts:

- From the Data menu, choose Edit Breakpoints.
- Double-click the line containing the breakpoint.
- Using the keyboard, move to the line containing the breakpoint, and press F9.

Examples

The following example removes breakpoints 0, 4, and 8:

```
>BC 0 4 8
```

The following example removes all breakpoints:

```
>BC *
```

The following example removes breakpoints 4, 5, 6, and 7:

```
>BC 4-7
```

BD (Breakpoint Disable)

Syntax

BD *[[list | start-end | *]]*

Parameters

list

List of breakpoints to be disabled, with breakpoint numbers separated by spaces. A number identifies each breakpoint. You can use the Breakpoint List (**BL**) command to display currently set breakpoints and their numbers.

start-end

Range of breakpoints to disable. The command disables breakpoints numbered from *start* to *end*, inclusive.

*

Disables all currently set breakpoints.

Description

The Breakpoint Disable (**BD**) command temporarily disables one or more existing breakpoints. The breakpoints are not deleted; they can be restored at any time using the Breakpoint Enable (**BE**) command.

A disabled breakpoint can be cleared using the Breakpoint Clear (**BC**) command.

In the Source window, enabled breakpoints are highlighted. However, the highlighting disappears once the breakpoint is disabled.

Mouse and Keyboard

As an alternative to typing the **BD** command, choose Edit Breakpoints from the Data menu. There is no keyboard shortcut.

Examples

The following example temporarily disables breakpoints 0, 4, and 8:

```
>BD 0 4 8
```

The following example temporarily disables all breakpoints:

```
>BD *
```

The following example disables breakpoints 4, 5, 6, and 7:

```
>BD 4-7
```

BE (Breakpoint Enable)

Syntax

BE [*list* | *start-end* | *]

Parameters

list

List of breakpoints to be enabled, with breakpoint numbers separated by spaces. A number identifies each breakpoint. You can use the Breakpoint List (**BL**) command to display currently set breakpoints and their numbers.

start-end

Range of breakpoints to enable. The command enables breakpoints numbered from *start* to *end*, inclusive.

*

Enables all currently disabled breakpoints.

Description

The Breakpoint Enable (**BE**) command enables breakpoints that have been temporarily disabled with the Breakpoint Disable (**BD**) command.

Mouse and Keyboard

In addition to typing the **BE** command, you can also enable breakpoints from the Data menu by choosing Edit Breakpoints. There is no keyboard shortcut.

Examples

The following example reenables breakpoints 0, 4, and 8:

```
>BE 0 4 8
```

The following example enables all disabled breakpoints:

```
>BE *
```

The following example enables breakpoints 4, 5, 6, and 7:

```
>BE 4-7
```

BL (Breakpoint List)

Syntax

BL

Description

The Breakpoint List (**BL**) command lists current information about all breakpoints.

For each breakpoint, the command displays the following:

- The breakpoint number.
- The breakpoint status, where “E” is for enabled, “D” is for disabled, and “V” is for “virtual.” A virtual breakpoint is a breakpoint set in an overlay or a DLL that is not currently loaded. A virtual breakpoint may be enabled or disabled.
- The address, function, file, and line number where the breakpoint is set.
- The expression, pass count, and break commands, if set.

Mouse and Keyboard

In addition to typing the **BL** command, you can also list breakpoints from the Data menu by choosing Edit Breakpoints. There is no keyboard shortcut.

BP (Breakpoint Set)

Syntax

BP [[*address*]] [[=*expression* [/R*range*]] | [/?*expression*]] [/P*passcount*] [/C"*commands*"] [/M*msgname*/*msgclass* [/D]]]

Parameters

address

An expression giving the address at which to set the breakpoint. If omitted, sets a breakpoint on the current line, unless =*expression* is also specified.

=*expression*

Breaks program execution when the value of *expression* changes. If *address* is given, the expression is checked for changes only at that address. The expression is usually the name of a variable.

/R*range*

Watches all addresses in the given range for changes. The range is determined by multiplying *range* with the size of *expression*.

?expression

Breaks program execution when *expression* becomes true (nonzero). If *address* is given, the breakpoint *expression* is evaluated only at that address. You cannot specify both *=expression* and *?expression* in the same breakpoint. Also, you cannot have more than one local context in *expression*. If the *expression* contains spaces, it must be enclosed in double quotation marks ("*expression*").

/Ppasscount

Specifies the first time the breakpoint is to be taken. For example, if *passcount* is 5, the breakpoint will be ignored the first four times it is encountered and taken the fifth time. From that point on, the breakpoint is always taken until the program is restarted.

/C"commands"

A list of command-window commands to be executed when the breakpoint is encountered. The *commands* must be enclosed in double quotation marks (" ") and separated by semicolons (;).

/Mmsgname

(CVW only) Breaks program execution whenever the specified message is received. When */D* is also specified, the message received is displayed, but the breakpoint is not taken.

/Mmsgclass

(CVW only) Breaks program execution whenever a message belonging to one of the specified classes is received. When */D* is also specified, the message received is displayed but the breakpoint is not taken. Can be one or more of the following:

Message Class	Type of Windows Message
m	Mouse
w	Window management
n	Input
s	System
i	Initialization
c	Clipboard
d	DDE
z	Nonclient

Description

The Breakpoint Set (**BP**) command creates a breakpoint at a specified address. Whenever a breakpoint is encountered during program execution, the program halts and waits for a new command.

You can set breakpoints at source lines, functions, explicit addresses, or labels in any module of a program. If no arguments are given, **BP** sets a breakpoint at the current line.

Windows Breakpoints

In CodeView for Windows (CVW) use of the */M* options requires that *address* be the name or address of a window function (“winproc”).

When the */D* option is specified, CVW displays each message in the Command window as it is sent to the application’s window function. The message is displayed in the following format:

```
HWND:wh wParam:wp lParam:lp msg:msgnum msgname
```

where *wh* is the window handle, *wp* is the message’s word-sized parameter, *lp* is the message’s long-sized parameter, *msgnum* is the message number, and *msgname* is the name of the message. The following is a typical display:

```
HWND:1c00 wParam:0000 lParam:000000 msg:000F WM_PAINT
```

Windows breakpoints appear in the list of breakpoints and may be enabled, disabled, and cleared with the usual CodeView breakpoint commands.

Breakpoint Options

For any breakpoint, you can also specify:

- A pass count to tell CodeView how many times to pass over the breakpoint.
- Commands to be executed after the program reaches the breakpoint.

Breakpoints are numbered, beginning with the number 0. Each new breakpoint is assigned the next available number. Breakpoints remain in memory until you explicitly delete them. Breakpoints are saved in the CURRENT.STS file when you exit CodeView and are restored the next time you debug the program.

Types of Breakpoints

You can set breakpoints to break execution when any of the following conditions are true:

- The program reaches a given source line, function, label, or address.
- An expression becomes true (nonzero). The CodeView expression evaluator evaluates this type of expression based on the the currently visible function.
- The value of an expression or memory range changes. CodeView references this type of expression by memory location. As a result, the original value of the expression is checked no matter which function is currently visible.

Mouse and Keyboard

In addition to typing the **BP** command, you can also set a breakpoint with the following shortcuts:

- From the Data menu, choose Set Breakpoint.
- Double-click a source line.
- Move the cursor to a source line, and press F9.

Examples

Command	Action
<code>BP @47</code>	Set a breakpoint at line 47 of the currently executing module.
<code>BP 0x23f0:3c84</code>	Set a breakpoint at address 23F0:3C84.
<code>BP =curr_sum</code>	Halt execution whenever the value in <code>curr_sum</code> changes.
<code>BP =myint /R8</code>	Halt execution whenever a change occurs in the range of eight integers that begins at <code>myint</code> . If <code>myint</code> is a 2-byte value, the range is 16 bytes in size.
<code>BP @47 =int_array[0] /R20</code>	Set a breakpoint at line 47 of the currently executing module. The breakpoint will be taken at that line if any 20 elements of the array <code>int_array</code> changes. Since <code>int_array</code> is a 2-byte value, the range is 40 bytes in size.
<code>BP {,mymod}@47 ?myptr==0</code>	Set a breakpoint at line 47 of the module <code>mymod</code> . The breakpoint is taken only if <code>myptr</code> is zero.
<code>BP stats /P10 /C"?counter+=1"</code>	Set a breakpoint at the address of the function <code>stats</code> but ignore the breakpoint the first nine times the function is executed. On the tenth and later call to <code>stats</code> , stop execution, and use the Display Expression (?) command to increment the value of <code>counter</code> . If <code>counter</code> is set to 0 when the breakpoint is set, <code>counter</code> can be used to count the number of times the breakpoint is taken.

E (Animate)

Syntax **E**

Description The Animate (**E**) command traces through a program one step at a time, with a user-selectable pause between each step, beginning at the current instruction. In

the Source mode, CodeView pauses after each line of source text. In the Mixed or Assembly-only mode, CodeView pauses after each instruction. The Animate command allows you to see how execution proceeds in your program.

You can set the time the command pauses with the Trace Speed (T) command or by choosing Trace Speed from the Options menu.

Mouse and Keyboard

To begin animating, you can also choose Animate from the Run menu. There is no keyboard shortcut.

G (Go)

Syntax

G [*address*]

Parameters

address

Address at which to stop execution.

Description

The Go (G) command starts execution at the current address. If *address* is given, CodeView executes the program until it reaches that address. If the specified address is never reached, the program executes until it terminates. If no address is given, CodeView executes the program until it terminates, until it reaches a break-point, or until you interrupt execution.

When CodeView reaches the end of the program in MS-DOS, it displays a message with the format:

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

The *number* is the program's return value (also known as the "exit" or "errorlevel" code). This is the value in the AX register at the time your program terminates. For example, the C function call

```
exit(1);
```

places 1 in the AX register and terminates the program.

Mouse and Keyboard

In addition to typing the G command, you can start execution using the following shortcuts:

- Click the <F5=Go> button in the status bar.
- Press F5.

To execute up to a certain location, you can use the following shortcuts:

- Click the right mouse button on the source line.
- Move the cursor to the source line and press F7.

Example

The following example executes up to the label `panic_exit` in the **main** function. Because labels are always local to a procedure, you must specify the context (procedure or function name) if the label is not in the current function.

```
>G {main}panic_exit
```

H (Help)

Syntax

H [*topic*]

Parameter

topic

Provides help on *topic*, which can be a command-window command. If no *topic* is given, the table of contents is displayed.

Description

The Help (**H**) command displays help information in a separate window. You can get help on CodeView commands, CodeView error messages, and any other topic within the Microsoft Advisor Help system.

Mouse and Keyboard

In addition to typing the **H** command, you can get help using the following short-cuts:

- With the right mouse button, click the keyword to display the corresponding Help topic. This method works in all CodeView windows except the Source, Memory, and 8087 windows.
 - Move the cursor to a topic and press F1 to display the corresponding Help topic.
 - Choose one of the commands on the Help menu.
-

I (Port Input)

Syntax

I *port*

Parameter

port

A 16-bit port address.

Description	<p>The Port Input (I) command reads and displays a byte from a specified hardware port. The specified port can be any 16-bit address. CodeView displays the byte read in the Command window.</p> <p>This command is often used in conjunction with the Port Output (O) command. Use this command to write and debug hardware-specific programs in Assembly mode.</p> <p>Note This command may affect the status of the hardware using the port.</p>
Example	<p>The following example reads the input port numbered 2F8 and displays the result, E8. You can enter the port address using any radix, but the result is always displayed in current radix.</p> <pre>>I 2F8 ;* hexadecimal radix assumed E8</pre>

K (Stack Trace)

Syntax **K**

Description	<p>The Stack Trace (K) command displays functions that have been called during program execution, including their arguments in the Command window. It also displays the address of the instruction that will be executed when control returns to each function.</p> <p>Output from the Stack Trace command gives you the following information:</p> <ul style="list-style-type: none">▪ Functions listed in the reverse order in which they were called.▪ Arguments to each function, listed in parentheses.▪ The address or line number of the next instruction to be executed when control returns to that function. <p>Thus, the current function is listed first, and the address of the next instruction to be executed is the current CS:IP address. At the bottom is the main function of your program and the address of the next instruction to be executed when execution returns to the main function.</p> <p>For each function, the command shows argument values in the current radix in parentheses after the function name.</p>
--------------------	---

You can use the address displayed for each line of the stack trace as an argument to the View Source (**VS**) or Unassemble (**U**) commands to see the code at the point where each function is called.

Mouse and Keyboard

In addition to typing the (**K**) command, you can use the Calls menu to see the stack trace.

Remarks

The term “stack trace” is used because as each function is called, its address and arguments are stored on or pushed onto the program stack. CodeView traces through the program stack to find out which functions were called. With C programs, the function **main** is always at the bottom of the stack.

The Stack Trace (**K**) command does not display anything until the program executes the beginning of the main function. The main function sets up the stack trace through frame pointers (the BP register), which CodeView uses to locate parameters, local variables, and return addresses.

If the main module is written in assembly language, the program must execute at least to the beginning of the first procedure called. In addition, your procedures must follow the standard Microsoft calling conventions.

Example

The following example shows the functions executed in a program so far, where hexadecimal is the current radix under CodeView:

```
>K
convert(0x3:0x17FC,1,2) address 1:ada
make_header(0x3:0x17FC) address 1:314
main(4,0x3:0x181E) address 1:c98
>
```

Here, `convert` is the currently executing function, at address ADA. It is passed three parameters: a pointer and two integers. When it returns control to `make_header`, the program is executing at address 314. When `make_header` returns, the program is executing at address C98.

L (Restart)

Syntax

L [*arguments*]

Parameter

arguments

New arguments to the program. No other CodeView commands may be specified after the Restart command. They are interpreted as additional program arguments.

Description	<p>The Restart (L) command resets execution to the beginning of the program and optionally sets a new program command line.</p> <p>After you issue the Restart command:</p> <ul style="list-style-type: none">▪ The program's variables are reinitialized.▪ The program's instructions are reset. Any modifications you may have made to the code with the Assemble (A), Memory Enter (ME), Memory Fill (MF), or View Memory (VM) commands are lost.▪ Any existing breakpoints or watch statements are retained. The pass counts for all breakpoints are reset. <p>Used alone, the Restart command keeps the previous command-line arguments specified for your program. You can change the command-line arguments using the Restart command followed by any new arguments to your program.</p> <p>You can use Restart any time execution has stopped: at any kind of breakpoint, while single-stepping, or when execution is complete.</p>
Mouse and Keyboard	<p>In addition to typing the L command, you can also restart from the Run menu by choosing Restart. To set a new command line and restart the program, choose Set Runtime Arguments from the Run menu. There is no keyboard shortcut.</p>
Remarks	<p>The Restart command does not reset system resources, such as open files or video mode, and does not free allocated system objects. If the application redefines interrupts, the system may no longer work correctly.</p>

MC (Memory Compare)

Syntax	<i>MC range address</i>
Parameters	<i>range</i> Range of first block of memory. <i>address</i> Starting address of second block of memory.
Description	<p>The Memory Compare (MC) command compares the bytes in a given range of memory with the corresponding bytes beginning at another address. If one or more pairs of corresponding bytes do not match, the command displays each pair of mismatched bytes.</p>

You can enter arguments to the Memory Compare (**MC**) command in any radix, but the output of the command is always in hexadecimal.

Examples

The following example compares the block of memory from 100 to 1FF with the block from 300 to 3FF. CodeView reports that the third and ninth bytes differ in the two ranges.

```
>MC 100 1FF 300 ;* hexadecimal radix assumed
004E:0102 0A 00 004E:0302
004E:0108 0A 01 004E:0308
>
```

The following example compares the 100 bytes starting at the address of `arr1[0]` with the 100 bytes starting at the address of `arr2[0]`.

```
>MC arr1[0] L 100 arr2[0] ;* decimal radix assumed
>
```

Because CodeView produced no output, the first 100 bytes of each array are identical.

MD (Memory Dump)

Syntax `MD[[format]] [[addressrange]]`

Parameters

format

Specifies the format to dump data. The format can be one of the following:

Specifier	Format
A	ASCII characters
B	Byte (hexadecimal)
C	Code (instructions)
I	Integer (2-byte decimal)
IU	Integer unsigned (2-byte decimal)
IX	Integer hex (2-byte hexadecimal)
L	Long (4-byte decimal)
LU	Long unsigned (4-byte decimal)
LX	Long hex (4-byte hexadecimal)
R	Real (4-byte float)
RL	Real long (8-byte float)
RT	Real ten-byte (10-byte float)

If *format* is not given, the Memory Dump command defaults to the format last used. If never used before, it defaults to an 8-bit dump.

address

Starting address of memory to be dumped. This can be any expression that evaluates to an address. The amount of memory dumped depends on the format specified. If *address* is omitted, the Memory Dump command defaults to the byte immediately following the last byte in the previous dump command. If the Memory Dump command was never used before, it defaults to DS:0000.

range

Range of memory to be dumped. Maximum range is 32K.

Description

The Memory Dump (**MD**) command displays the contents of memory in the command window, using the format you specify. This command can be used with the Redirection commands to send the contents of memory to another device. Use the View Memory (**VM**) command to display the contents of memory in a separate window.

The Memory Dump Code (**MDC**) command is like the Unassemble (**U**) command, except that **MDC** displays instructions in the Command window instead of the active Source window. Although you normally specify a range with the L character, you can also use the I character with **MDC** to specify a range of instructions rather than bytes.

Examples

The following example displays 12 instructions starting from the address at line number 32 in the source code:

```
>mdc .32 I 12
```

The following example displays the byte values in the range between DS:0 and DS:1B. The data segment is assumed when no segment is given. ASCII characters are shown on the right.

```
>mdb 0x0 0x1b
0087:0000 00 00 00 00 00 00 00 00 4D 53 20 52 75 6E .....MS Run
0087:000E 2D 54 69 6D 65 20 4C 69 62 72 61 72 79 20 -Time Library
>
```

The following example displays seven elements of `float_array` as four-byte real values:

```
>mdr float_array[0]
0087:0D56 DC 0F 49 40 +3.141593E+000
0087:0D5A A0 17 CE 3F +1.610096E+000
0087:0D5E 66 66 5B C2 -5.485000E+001
0087:0D62 00 00 C0 3F +1.500000E+000
0087:0D66 FF FF 1F 41 +9.999999E+000
0087:0D6A 00 00 00 00 +0.000000E+000
0087:0D6E 00 00 00 00 +0.000000E+000
>
```

ME (Memory Enter)

Syntax `MEtype address [[list]]`

Parameters

type

Specifies the type of the data to be entered into memory.

Specifier	Type
A	ASCII characters
B	Byte (hexadecimal)
I	Integer (2-byte decimal)
IU	Integer unsigned (2-byte decimal)
IX	Integer hex (2-byte hexadecimal)
L	Long (4-byte decimal)
LU	Long unsigned (4-byte decimal)
LX	Long hex (4-byte hexadecimal)
R	Real (4-byte float)
RL	Real long (8-byte float)
RT	Real ten-byte (10-byte float)

If no *type* is given, the command defaults to the last type used by a Memory Enter (**ME**), a Memory Dump (**MD**), or a View Memory (**VM**) command. If no such commands were used, it defaults to byte-sized data.

address

Indicates where the data will be entered. If no segment is given in the address, the data segment (**DS**) is assumed.

list

List of data to enter into memory at *address*. These expressions must evaluate to data of the size specified by *type*. If *list* is not given, CodeView prompts for new values.

Description

The Memory Enter (**ME**) command enters one or more byte values into memory at a given address.

The command may include a list of expressions separated by spaces. The expressions are evaluated and entered in the current radix. If an invalid value appears in the list, CodeView refuses to enter the invalid value and ignores remaining values. If no list is given, CodeView prompts for new values.

Because it can modify any part of your program's memory, the Memory Enter command can change your program's instructions. The Assemble (**A**) command, however, is better suited to that purpose.

Mouse and Keyboard

There is no keyboard shortcut to enter items into memory. You can use the Memory window, however, to alter items in memory.

Entering Values

If you do not give a list of expressions in a Memory Enter (**ME**) command, CodeView prompts for a new value at the address you specify by displaying the address and its current value. At this point, you can do one of the following:

- Replace the value by typing a new value.
- Skip to the next value by pressing the **SPACEBAR**. Once you have skipped to the next value, you can change its value or skip again. CodeView will automatically prompt with new addresses as necessary.
- Return to the preceding value by typing a backslash (****). When you return to the preceding value, CodeView starts a new display line and prompts with the address and current value.
- Stop entering values and return to the command prompt by pressing **ENTER**.

Example

The following example replaces the byte at DS:256 (DS:0100 hexadecimal) with 66 (42 hexadecimal).

```
>MEB 256
3DA5:0100 41 A. 66
>
```

MF (Memory Fill)

Syntax

MF *range list*

Parameters

range

Specifies the range of memory to be filled.

list

List of byte values used to fill *range*.

Description

The Memory Fill (**MF**) command fills the addresses in the specified range with the byte values specified in the argument list. You can enter byte values using any radix.

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.

The Memory Fill command provides an efficient way to fill up a block of memory with any values you specify. You can use it to initialize large data areas, such as arrays or structures. Because it can modify any part of your program's memory, the Memory Fill command can change your program's instructions. However, the Assemble (**A**) command is better suited to that purpose.

Examples

The following example fills 255 (100 hexadecimal) bytes of memory starting at DS:0100 with the value 0; hexadecimal radix is assumed. This command could be used to reinitialize the program's data without having to restart the program.

```
>MF 100 L 100 0  
>
```

This next example fills the 100 (64 hexadecimal) bytes starting at `table` with the following hexadecimal byte values: 42, 79, 74. These three values are repeated (42, 79, 74, 42, 79, 74, ...) until all 100 bytes are filled; hexadecimal radix is assumed.

```
>MF table L 64 42 79 74  
>
```

MM (Memory Move)

Syntax

MM *range address*

Parameters

range

Specifies the range of memory to copy.

address

Destination address to copy the *range*.

Description	<p>The Memory Move (MM) command copies all the values in one block of memory directly to another block of memory of the same size. All data in the source block is guaranteed to be copied completely over the destination block, even if the two blocks overlap.</p> <p>When the source is at a higher address than the destination, the Move Memory command copies data starting at the source block's lowest address. When the source is at a lower address, the Memory Move command copies data beginning at the source block's highest address.</p> <p>You use the Memory Move command to program in Assembly mode (to copy function fragments, for example) or to copy large amounts of data.</p>
Examples	<p>In the following example, the block of memory to copy begins with the first element of <code>array1</code> and is <code>array_size</code> bytes long. It is copied directly to a block of the same size, beginning at the address of the first element of <code>array2</code>.</p> <pre>>MM array1[0] L array_size array2[0] ></pre>

MS (Memory Search)

Syntax	<i>MS range list</i>
Parameters	<p><i>range</i> The range of memory to search.</p> <p><i>list</i> A list of byte values separated by spaces or commas or an ASCII string delimited by quotation marks.</p>
Description	<p>The Memory Search (MS) command scans a range of memory for specific byte values. Use this command to test for the presence of certain values within a range of data.</p> <p>You can specify any number of byte values to the Memory Search command. Unless the list is an ASCII string, each byte value must be separated by a space or a comma.</p> <p>If the list contains more than one byte value, the Memory Search command looks for a series of bytes that precisely match the order and value of bytes in the list. If</p>

the command finds such a series of bytes, it displays the beginning address of that series.

Examples

The following example displays the address of each memory location containing the string `error`. The command searched the first 1,500 bytes at the address specified by the variable `buffer`. CodeView found the string at three addresses.

```
>MS buffer L 1500 "error"
2BBA:0404
2BBA:05E3
2BBA:0604
>
```

The following example displays the address of each memory location that contains the byte value 0A in the range DS:0100 to DS:0200; hexadecimal is assumed to be the default radix. CodeView found the value at two addresses.

```
>MS DS:100 200 A ;* hexadecimal radix assumed
3CBA:0132
3CBA:01C2
>
```

N (Radix)

Syntax

N *[[radix]]*

Parameter

radix

New radix while running CodeView. Can be 8 (octal), 10 (decimal), or 16 (hexadecimal). If omitted, the command displays the current radix.

Description

The Radix (N) command changes the current radix for entering arguments and displaying the values of expressions. The new radix number can be 8 (octal), 10 (decimal), or 16 (hexadecimal). Binary and other radices are not allowed. With no arguments, the command displays the current operating radix.

Note Changing the radix does not convert the l-value of displayed expressions, only the r-value.

When you start up CodeView, the default radix is 10 (decimal), unless your main program is written with the Microsoft Macro Assembler (MASM). In this case, the default radix is 16 (hexadecimal).

Remarks

The following conditions are not affected by the Radix command:

- The radix for entering a new radix is always decimal.
- Format specifiers given with the Display Expression (?) command override the current radix.
- Addresses are always shown in hexadecimal.
- In Assembly mode, all values are shown in hexadecimal.
- The display radix for the Memory Dump (MD) and Breakpoint Set (BP) commands is always hexadecimal if the size is bytes, words, or doublewords; it is always decimal if the size is integers, unsigned integers, short reals, long reals, or 10-byte reals.
- The input radix for the Memory Enter (ME) command's prompt is always hexadecimal if the size is bytes, words, or doublewords; it is 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.
- The register display is always in hexadecimal.

Example

The following example shows the decimal equivalents of the number 14 in octal and in hexadecimal.

```
>N8
>? 14,i
12
>N16
>? 14,i
20
>
```

Here, the Display Expression (?) command uses the *i* format specifier, which prints a number in decimal regardless of the current radix.

O (Options)

Syntax

```
O[[option[[+|-]] ]]
OL[[ [scope]][[+|-]] ]]
```

Parameters

option

Character indicating the option to be turned on or off.

Specifier	Option
A	Show Status Bar
B	Bytes Coded
C	Case Sense
F	Flip/Swap
H	Horizontal Scroll Bar
L	Show Address
N	Native Mode
S	Symbols
3	386
V	Vertical Scroll Bar

scope

For the **OL** command, you can specify a scope of variables to display in Local window using one or more of the following:

Specifier	Scope
L	Lexical
F	Function
M	Module
E	Executable
G	Global
*	All of the above

+

Turns option(s) on.

-

Turns option(s) off.

Description

The Options (**O**) command allows you to view or set the state of the following CodeView options:

Letter	Option	Display
A	Show Status Bar	If on, the status bar appears at the bottom of the screen. If off, the bottommost line becomes part of the window area.
B	Bytes Coded	If on, instruction addresses and machine code are displayed for assembly instructions.
C	Case Sense	If on, symbols are case-sensitive; if off, they are not.
F	Flip/Swap	If on, CodeView flips the program and output screens as the program executes. If off, no screen flipping is performed.

Letter	Option	Display
H	Horizontal Scroll Bar	If on, windows have a horizontal scroll bar.
L	Show Address	If on, addresses relative to BP for all local variables are displayed in the Local window.
N	Native Mode	If on, instructions are displayed in the native processor format. If off, p-code instructions are displayed.
S	Symbols	If on, symbols in assembly instructions appear in symbolic form. If off, they appear as addresses.
3	386	If on, registers appear in wide 80386 format, and you can assemble and execute instructions that reference 32-bit registers and memory.
V	Vertical Scroll Bar	If on, windows have a vertical scroll bar.

The Local window always displays variables local to the current routine. You can specify a scope of additional variables to display in the Local window with **OL** form of the Options command. Using **OL** with no options displays the current scope setting for the Local window.

The **O** form of the command (all options) takes no arguments; it displays the state of all options. The other forms of the command (**OF**, **OB**, **OC**, **OS**, **OL**, **O3**, **OA**, **ON**, **OH**, and **OV**) can be used as follows:

- With no arguments. The state of the option is displayed.
- With the + or – argument. The + argument turns the option on; the – argument turns the option off.

Mouse and Keyboard

As an alternative to typing the **O** command, you can view and set options on the Options menu.

Remarks

Use the Options (**O**) command to set options when you first start CodeView. You can set these options in the following ways:

- Give one or more **O** commands with the /C option on the CodeView command line or include a similar command line in the CodeView response file.
- Give one or more **O** commands as the **Autostart** entry in the TOOLS.INI file.

Example

In the following example, the **O** command is used to display current option settings. Then, the **O3** and **OF** commands are used to display and set options for 386 mode and for screen flip/swap mode. Finally, the **OL** command turns on symbol addresses in the Local window and displays not only local variables but global variables as well.

```
>0
Flip/Swap On
Bytes Coded On
Case Sense On
Show Symbol Address On
Symbols Off
Vertical scroll bar On
Horizontal scroll bar On
Status bar On
>03
386 Off
>03+
386 On
>0F
Flip/Swap On
>0F-
Flip/Swap Off
>0LG+
```

O (Port Output)

Syntax *O port byte*

Parameters *port*
 A 16-bit port address.

byte
 Byte to send to *port*.

Description The Port Output (**O**) command sends a byte value to a hardware port. You use this command to debug a program that interacts directly with hardware.

The Port Output command is often used with the Port Input (**I**) command.

Example In the following example, the byte value 4F hexadecimal is sent to output port 2F8.

```
>0 2F8 4F ;* hexadecimal radix assumed
>
```

P (Program Step)

Syntax **P** [*count*]

Parameters *count*
Repeat stepping *count* times.

Description The Program Step (**P**) command executes the current line (in Source mode) or instruction (in Mixed or Assembly mode), stepping over functions. To trace into functions, use the Trace (**T**) command. If a value for *count* is specified, CodeView steps through *count* lines or instructions. If not, only the current line or instruction is executed.

In Source mode, if the current source line contains a function call, CodeView executes the entire function and is ready to execute the line after the call.

In Mixed or Assembly Mode, if the current instruction is CALL, INT, or REP, CodeView executes the entire procedure, interrupt, or repeated string sequence.

Mouse and Keyboard In addition to typing the **P** command, you can step through a program using the following shortcuts:

- Click the <F10=Step> button in the status bar to step once.
- Press F10 to step once.

Q (Quit)

Syntax **Q**

Description The Quit (**Q**) command terminates CodeView and returns control to the environment from which CodeView was invoked: Programmer's WorkBench (PWB), Windows, or the operating system.

CodeView always saves state information on exit.

Mouse and Keyboard As an alternative to typing the **Q** command, choose Exit from the File menu. There is no keyboard shortcut.

R (Register)

Syntax **R** [*register* [[**[[=]]***expression*]]]

Parameters *register*

Change the contents of the given register. If omitted, displays the values of all registers and flags and the current machine instruction.

[[=]]*expression*

Assign the value of the expression to the specified register. The equal sign (=) is optional; a space has the same effect.

Description

The Register (**R**) command displays and changes the values in the CPU registers. To display register contents without changing them, type the Register (**R**) command without any arguments. This form of the command shows the current values of all registers and flags. Flags are shown symbolically. It also shows the current instruction at the address given by CS:IP.

If an operand of the instruction contains memory expressions or immediate data, CodeView evaluates the operand and indicates 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.

Changing Registers

You can use the **R** command to change the values in CPU registers. Also, you can change the bits in the flag register symbolically without having to compute a value of the register.

Changing Register Values

To change the value in a register:

1. Type the command letter **R** followed by the name of a register. The register name can be any of the following: AX, BX, CX, DX, CS, DS, SS, ES, SP, BP, SI, DI, IP, or FL (for flags). If you have a 80386/486-based machine and the 386 option is turned on, the register name can be one of the 32-bit register names: EAX, EBX, ECX, EDX, ESP, EBP, ESI, or EDI.

Note CodeView allows you to load different execution models which may specify a certain set of registers. For example, the valid registers in the p-code model are DS, SS, CS, IP, SP, BP, PQ, TH, and TL.

2. CodeView displays the current value of the register and prompts for a new value with a colon (:).
 - If you only want to examine the value, press ENTER.
 - If you want to change the value, type an expression (in the current radix) for the value and press ENTER.

- As an alternative, you can use the Display Expression (?) command to change the value in a register:

?register=expression

Changing Flag Values

To change a flag value:

1. Type the command letter **R** followed by the letters FL.
2. The command displays the value of each flag as a two-letter name. At the end of the list of values, the command prompts for new flags with a dash (-).
3. Type the new values after the dash for the flags you wish to change, then press ENTER.
 - You can enter flag values in any order. If you do not enter a new value for a flag, it remains unchanged.
 - If you do not want to change any flags, press ENTER.

Note If you enter an illegal flag name, CodeView displays an error message. The flags preceding the error are changed; flags at and following the error are not changed.

The flag values are:

Flag	Set Symbol	Clear Symbol
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

Mouse and Keyboard

As an alternative to typing the **R** command, you can use the Register window to display CPU values. To change CPU values with the Register window, type over the old values.

Example In the following example, the **R** command displays the current registers and CPU flags. Then the **R** command changes the value in the AX register.

```
>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
0047:014F 8B5E06          lea    di,[BP+06] ss:ff38=299E
>R AX
AX 0005
:3
>
```

T (Trace)

Syntax **T** [*count*]

Parameters *count*

Repeat tracing *count* times. If omitted, trace once.

Description The Trace (**T**) command executes the current line (in Source mode) or instruction (in Assembly or Mixed mode), tracing into functions or assembly-language CALL instructions. Use the Program Step (**P**) command to execute function calls without tracing into them.

In Source mode, the Trace command traces into functions whose source code is available and executes through those functions whose source is unavailable.

In Assembly or Mixed mode, CodeView always traces into functions. If the current instruction is CALL or INT, CodeView executes the first instruction of the procedure or interrupt. If the current instruction is REP, CodeView executes one iteration of the repeated string sequence.

CodeView executes MS-DOS function calls without tracing into them. CodeView can trace through BIOS calls in Assembly or Mixed mode.

Since the Trace command uses the hardware trace mode of the 8086 family of processors, you can also trace instructions stored in read-only memory (ROM). However, the Program Step command does not work in ROM; in this case, it has the same effect as the Go (**G**) command.

Mouse and Keyboard

In addition to typing the **T** command, you can trace once with the following shortcuts:

- Click the <F8=Trace> button in the status bar.
- Press F8.

T (Trace Speed)

Syntax **T**{**S**|**M**|**F**}

Parameter {**S**|**M**|**F**}

Specifies the trace speed for the Animate (**E**) command. You can specify the following speeds:

Specifier	Speed
S	Slow (1/2 second between steps)
M	Medium (1/4 second between steps; default)
F	Fast (no wait between steps)

Description The Trace Speed command controls the speed at which CodeView executes a program with the Animate (**E**) command.

Mouse and Keyboard

In addition to typing the **TS**, **TM**, or **TF** commands, you can also set the trace speed from the Options menu. There is no keyboard shortcut.

U (Unassemble)

Syntax **U** [*context*][*address*]

Parameters *context*
 Any legal context operator.

address
 Shows unassembled instructions starting at this address. If omitted, unassemble at the current CS:IP address.

Description The Unassemble (**U**) command displays assembly-language code beginning at the specified address in the active Source window. If you omit an address, the

command uses the current CS:IP address. The command changes the Source window to Assembly mode.

Setting the Source window display mode to Assembly and giving the Unassemble command with no arguments causes the code to scroll to the next page of assembly-language instructions.

Note If you specify an address that is within an instruction or within program data, CodeView will still attempt to disassemble and display instructions. Instructions that CodeView cannot disassemble are shown as ???.

Mouse and Keyboard

As an alternative to typing the U command, you can display assembly-language instructions using the following shortcuts:

- Click the <F3=Src1 Fmt> or <F3=Src2 Fmt> buttons until the active Source window is in Assembly mode.
- Press F3 until the Source window is in Assembly mode.
- From the Options menu, choose Source Window. Then set the display mode to Assembly.

Note that with these shortcuts, you cannot specify an address to start showing unassembled instructions.

Example

The following example sets the mode of the Source window to Assembly and displays assembly-language instructions beginning at address 0x7:0x11.

```
>U 0x7:0x11
>
```

USE (Use Language)

Syntax

USE *evaluator*

Parameter

evaluator

Selects the specified expression evaluator. If omitted, the command displays the currently selected expression evaluator. You can specify **AUTO** for the evaluator. With this setting, CodeView selects the appropriate expression evaluator based on the extension of the source file.

Description

The **USE** command specifies which expression evaluator CodeView is to use while debugging.

Mouse and Keyboard

As an alternative to typing the **USE** command, choose the Language command from the Options menu. There is no keyboard shortcut.

Remarks

When you switch expression evaluators, CodeView displays expressions in the Local and Watch windows with the nearest equivalent type in the new language. If the new language does not have an equivalent type, the results are unpredictable.

VM (View Memory)

Syntax

VM[[*window*]] [[*type*]] [[*address*]] [[*options*]]

Parameters

window

Specifies the memory window to display or change (1 or 2). If a value for *window* is omitted, the command defaults to the active Memory window or Memory window 1 if no Memory windows are open.

type

Specifies the data-type format of the window's display.

Value	Format
A	ASCII characters
B	Byte (hexadecimal)
I	Integer (2-byte decimal)
IU	Integer unsigned (2-byte decimal)
IX	Integer hex (2-byte hexadecimal)
L	Long (4-byte decimal)
LU	Long unsigned (4-byte decimal)
LX	Long hex (4-byte hexadecimal)
R	Real (4-byte float)
RL	Real long (8-byte float)
RT	Real ten-byte (10-byte float)

If *format* is omitted, the command defaults to the last type used by a View Memory (**VM**) command or to byte-display format if the **VM** command was never used.

address

Starting address of memory to display or any expression that evaluates to an address. If *address* is omitted, the command defaults to the current address in the active Memory window or DS:00 if no Memory windows are open.

options

Specifies how to display and update the Memory window's contents.

/R[[+|-]]

Raw data display

Option	Description
+	CodeView displays formatted data along with the corresponding bytes in hexadecimal format.
– (default)	CodeView displays only formatted data.

/L[[+|-]]

Live expression

Option	Description
+	Dynamic: CodeView evaluates <i>address</i> at each step and adjusts the Memory window accordingly.
– (default)	Static: CodeView evaluates <i>address</i> only when the command is entered.

/F[[*| *length*]]

Fixed-width data display

Option	Description
* (default)	CodeView displays as many items as will fit in the window.
<i>length</i>	CodeView displays a fixed number of items on each line. Must be in the range 1–125.

Description

The View Memory (**VM**) command displays the contents of memory in a Memory window using the type and format you specify. The Memory window is updated whenever you execute a command. You can modify memory in the window directly by typing over the displayed memory.

If you enter the **VM** command with no arguments and no Memory windows are open, CodeView opens Memory window 1 in the default display format (variable-width byte display at a static address). If you enter the **VM** command with no arguments and at least one Memory window is open, CodeView displays the current settings for the Memory windows in the Command window.

You can directly modify memory using the Memory window. Type over the values displayed in the active Memory window.

To display the contents of memory in the Command window, use the Memory Dump (**MD**) command.

Mouse and Keyboard

In addition to typing the **VM** command, you can open and manipulate Memory windows with the following shortcuts:

- To open a Memory window from the Windows menu, choose Memory 1 or Memory 2.
- To set display format and enter expressions for a Memory window, choose Memory Window from the Options menu.

You can cycle through the display formats with the following shortcuts:

- Click the <SH+F3=Mem1 Fmt> or <SH+F3=Mem2 Fmt> buttons in the status bar.
- Press SHIFT+F3 to cycle forwards.
- Press CTRL+SHIFT+F3 to cycle backwards.
- When the cursor is in the Memory window, press CTRL+O to display the Memory Window Options dialog box.

Examples

The following example opens Memory window 2 and displays memory in integer format plus the raw bytes that make up the integers, beginning at the address of the variable `myint`.

```
>VM2I /R myint
```

The following example specifies ASCII format for the current Memory window. The memory displayed begins at the string referred to by element `i` of the array `argv`. The expression is live, so the display is updated as `i` changes.

```
>VMA /L *argv[i]
```

VS (View Source)**Syntax**

```
VS[[window]] [[format]] [[address]] [[/option[[+|-]] ]]
```

Parameters

window

Specifies the Source window (1 or 2) to open or make active.

format

Specifies the way to display source code as one of the following:

Specifier	Format
+	Display source lines from the source file
-	Display assembly-language instructions
&	Display both source lines and assembly-language instructions

address

Address or line number at which to start displaying source code. The address must fall within the executable portion of your program.

[/option[+|-]] ...

Zero or more source display options. The option can be any of the following specifiers:

Specifier Option

a	Address When turned on (/a[+]), displays the address of each instruction. When turned off (/a-), does not display addresses.
b	Bytes coded When turned on (/b[+]), displays the hexadecimal form of the instructions. When turned off (/b-), does not display the encoded bytes.
c	Case of disassembly When turned on (/c[+]), displays instruction mnemonics and registers in uppercase. When turned off (/c-), displays instruction mnemonics and registers in lowercase.
l	Line-oriented display When turned on (/l[+]), displays mixed source and assembly in source-line order. When turned off (/l-), displays mixed source and assembly in instruction-code order.
s	Symbols in disassembly When turned on (/s[+]), symbols in instructions appear in symbolic form. If turned off (/s-), they appear as addresses.
t	Track current location (CS:IP) When turned on (/t[+]), the Source window follows the thread of execution (CS:IP). When turned off (/t-), the Source window does not automatically scroll to follow the current location.

Description

CodeView can display two Source windows at the same time. At least one source window must always be open. You can type the **VS 1** or **VS 2** command to make Source window 1 or 2 active. If the Source window you request is not open, CodeView opens it and makes it active.

The Source windows can show code in a number of display modes:

Source

CodeView displays the lines from your program's source files.

Assembly

CodeView displays the assembly instructions that make up your program.

Mixed

CodeView displays each line of your program's source file, followed by the assembly instructions for that line. This ordering can be reversed by turning the Line-Oriented Display option off (/l-).

Source and Mixed modes are available only if the executable file contains debugging information.

Note Programs that do not contain debugging information are always displayed in Assembly mode.

In the Source and Mixed modes, tracing into a function for which no source lines are available, such as a library function, switches the Source window to Assembly mode. Once program execution returns to an area where source lines are available again, CodeView automatically switches back to Source or Mixed mode.

If you specify a line number or an address with the **VS** command, CodeView draws the Source window so that the source line corresponding to the given address appears in the middle of the Source window. If the address is in another file, CodeView loads that file into the Source window. If you specify an address for which there is no corresponding source text (in your program's data, for example), CodeView will respond with an error message.

You can scroll the contents of the active Source window down a page by typing the **VS** command with no arguments. You can also use the Source window scroll bars.

Mouse and Keyboard

To make a Source window active or to open a Source window:

- Click anywhere in an open Source window to make it active.
- Press **ALT+3** or **ALT+4** to activate or open Source window 1 or 2.
- From the Windows menu, choose Source 1 or Source 2.

To change the source display mode:

- Click the **<F3=Src1 Fmt>** or **<F3=Src2 Fmt>** buttons in the status bar to cycle through the three modes.
- Press **F3** to cycle forward.
- Press **CTRL+F3** to cycle backward.
- From the Options menu, choose Source Window to open the Source Window Options dialog box. Under Display Mode, select one of the option buttons.
- When the cursor is in the Source window, press **CTRL+O** to display the Source Window Options dialog box.

Examples

The following example opens Source window 2 in the mixed mode. The display will start at the function `toss_token`.

```
>VS 2 & toss_token
```

The next example changes the display format in Source window 2 to source lines only.

```
>VS 2 +
```

W? (Add Watch Expression)

Syntax

W? *expression* [, *format*]

Parameters

expression

Expression to add to the Watch window.

format

A CodeView format specifier that indicates the format in which *expression* is displayed.

Description

The Add Watch Expression (**W?**) command displays one or more specified values in the Watch window. Watch expressions allow you to watch how a variable changes as your program executes. CodeView updates the Watch window each time the value of the watch expression changes during program execution.

The Watch window shows variables in the default format for their types. To display a watch expression in a different format, type a comma after the expression, followed by a CodeView format specifier. You can also cast the expression to the format you want to use.

CodeView always evaluates watch expressions according to the current radix and reevaluates watch expressions if the radix changes.

For relational expressions, the Watch window shows 0 if the expression is false and 1 if the expression is true.

Mouse and Keyboard

As an alternative to typing the **W?** command, choose the Add Watch command from the Data menu. There is no keyboard shortcut.

Examples

Command	Action
W? n	Display the value of the variable <i>n</i> in the Watch window.
W? high * 100	Display the value of 100 times the variable <i>high</i> in the Watch window.
W? (char *) 0	Display the byte at DS:0. Because 0 is explicitly cast to a pointer type, CodeView treats it as an offset rather than a constant.

WC (Delete Watch Expression)

Syntax *WC number**

Parameters *number*
 Deletes the watch expression with this number.

*
 Deletes all watch expressions.

Description The Delete Watch Expression (**WC**) command removes a watch expression from the Watch window.

When you set a watch expression, CodeView automatically assigns it a number, starting with 0 for the first watch expression in the window. Use the List Watch (**WL**) command to view the numbers of current watch statements.

Mouse and Keyboard In addition to typing the **WC** command, you can use the following shortcuts to delete watch expressions:

- From the Data menu, choose Delete Watch.
- Select the Watch window, move the cursor to the watch expression, and press CTRL+Y.

Examples The following example deletes watch expression 2 from the Watch window:

```
>WC 2
```

The following example deletes all watch expressions from the Watch window:

```
>WC *
```

WDG (Windows Display Global Heap)

Syntax **WDG** [*ghandle*]

Parameter *ghandle*
A handle to a global memory object. The **WDG** command displays the five memory objects in the global heap, starting at the specified object. The *ghandle* must be a valid handle to an object allocated on the global heap. If *ghandle* is not specified, **WDG** displays the entire global heap.

Description Global memory objects are displayed in the order in which Windows manages them, which is typically not in ascending handle order. The output from the **WDG** command has the following format:

Format *handle address size PDB locks type owner*

Any field may not be present if that field is not defined for the block.

Field	Description
<i>handle</i>	Value of the global memory block handle.
<i>address</i>	Address of the global memory block.
<i>size</i>	Size of the block in bytes.
PDB	Block owner. If present, indicates that that task's Process Descriptor Block is the owner of the block.
<i>locks</i>	Count of locks on the block.
<i>type</i>	The memory-block type.
<i>owner</i>	The block owner's module name.

WDL (Windows Display Local Heap)

Syntax **WDL**

The output from the **WDL** command has the following format:

Format *handle address size flags locks type heaptypes blocktype*

Any field may not be present if that field is not defined for the block.

Field	Description
<i>handle</i>	Value of the global memory block handle
<i>address</i>	Address of the block
<i>size</i>	Size of the block in bytes
<i>flags</i>	The block's flags
<i>locks</i>	Count of locks on the block
<i>type</i>	The type of the handle
<i>heaptype</i>	The type of heap the block resides in
<i>blocktype</i>	The block's type

WDM (Windows Display Modules)

Syntax **WDM**

Description The **WDM** command displays a list of all DLL and application modules loaded by Windows. Each line of the display has the following format:

Format *handle refcount module path*

Field	Description
<i>handle</i>	The module handle
<i>refcount</i>	The number of times the module has been loaded
<i>module</i>	The name of the module
<i>path</i>	The path of the module's executable file

WGH (Windows Dereference Global Handle)

Syntax **WGH** *handle*

Parameter *handle*
 Global memory handle of memory object to convert.

Description To convert a global memory handle to a pointer, use the **WGH** command. **WGH** converts a global memory handle into a near or a far pointer. Use **WLH** to convert local memory handles.

The **WDG** and **WDL** commands convert the handle into a pointer and display the value of the pointer in *segment:offset* format. You can then use that value to access the memory.

In a Windows program, the **GlobalLock** function is used to convert memory handles into near or far pointers. You may know the handle of the memory object, but you might not know what near or far address it refers to unless you are debugging in an area where the program has just called **GlobalLock**.

You use the **WGH** command at any time to find out what the pointer addresses are for global memory handles.

Example

The following example is used to display a string in Window's global heap. First, the following code sets up the string:

```
HANDLE hGlobalMem;
LPSTR lpstr;

hGlobalMem = GlobalAlloc( GMEM_MOVEABLE, 10L )
lpstr = GlobalLock( hGlobalMem );
lstrcpy( lpstr, "ABCDEF" );
GlobalUnlock( hGlobalMem );
```

You can display the contents of the string with the following sequence of commands:

```
>wgh hGlobalMem
0192:6E30
>? *(char far*) 0x0192:0x6E30,s
```

WKA (Windows Kill Application)

Syntax

WKA

Description

The Windows Kill Application (**WKA**) command terminates the current task by simulating a fatal error.

There may be times when you want to halt your program immediately. You can force an immediate interrupt of a CVW session by pressing CTRL+ALT+SYSREQ. You then have the opportunity to change debugging options, such as setting break-points and modifying variables. To resume continuous execution, press F5; to single-step, press F10.

You should take care when you interrupt the CVW session. For example, if you interrupt the session while Windows code or other system code is executing, using the Step or Trace functions produces unpredictable results. When you interrupt the CVW session, it is usually safest to set breakpoints in your code and then resume continuous execution rather than using Step or Trace.

If the current code is in your application, you can safely use the **WKA** command without affecting other tasks. However, the **WKA** command does not perform all the cleanup tasks associated with the normal termination of a Windows application.

For example, global objects created during the execution of the program but not destroyed before you terminated the program remain allocated in the global heap. This reduces the amount of memory available during the rest of the Windows session. For this reason, you should use the **WKA** command to terminate the application only if you cannot terminate it normally.

For more information on using the Windows Kill Application (**WKA**) command, see Chapter 10.

WL (List Watch Expressions)

Syntax	WL
Description	The List Watch Expressions (WL) command lists all currently set watch expressions along with their numbers and values.
Mouse and Keyboard	As an alternative to typing the WL command, you can use the Watch window to view the current watch expressions.
Example	The following example displays watch expressions and their values: <pre>>WL 0) code : 17 1) (float)letters/words : 4.777778 2) lines==11 : 0 ></pre> <p>In the example, three watch expressions are set:</p>

1. The variable `code`, which is 17.
 2. The arithmetic expression `letters` divided by `words` as a floating point number, currently 4.777778
 3. The conditional expression `lines==11`, currently false (zero).
-

WLH (Windows Dereference Local Handle)

Syntax `WLH handle`

Parameter `handle`
Local memory handle of memory object to convert.

Description To convert a local memory handle to a pointer, use the Dereference Local Handle (**WLH**) command. **WLH** converts a local memory handle into a near or a far pointer. Use **WGH** to convert global memory handles.

The **WDG** and **WDL** commands convert the handle into a pointer and display the value of the pointer in *segment:offset* format. You can then use that value to access the memory.

In a Windows program, the **LocalLock** function is used to convert memory handles into near or far pointers. You may know the handle of the memory object, but you might not know what near or far address it refers to unless you are debugging in an area where the program has just called **LocalLock**.

You use the **WLH** command to find out at any time what the pointer addresses are for local memory handles.

Example The following example uses **WLH** to refer to an array during a debugging session. First, the following code sets up the array:

```
{
HANDLE      hLocalMem;
int near *  pnArray;
hLocalMem = LocalAlloc( LMEM_MOVEABLE, 100 );
pnArray = LocalLock( hLocalMem );

/* load values into the array */

LocalUnlock( hLocalMem );
. . .
```

Now, after setting a breakpoint immediately after the call to **LocalLock**, the following command displays the array location:

```
>mdw pnArray
```

Outside of this fragment, though, you cannot rely on the value of the `pnArray` variable since the actual data in the memory object may move. Therefore, use the following sequence to display the correct array location:

```
>w!h hLocalMem
0192:100A
>mdw 0192:100A
```

X (Examine Symbols)

Syntax `Xscope [[context]][[regex]]`

Parameters

scope

Specifies the scope in which to search for symbols. Can be one or more of the following:

Specifier	Scope
L	Lexical
F	Function
M	Module
E	Executable
P	Public
G	Global
*	All of the above

context

Specifies context under which to search with the context operator.

regex

Specifies a CodeView regular expression.

Description

The Examine Symbols (**X**) command displays the names and addresses of symbols and the names of modules defined within a program. You can specify the scope in which to search and a regular expression against which to match symbols. You can further specify a context using the context operator.

For more information on regular expressions, see Appendix A.

Examples

The following example shows all the symbols and their addresses in the current lexical scope. The command uses the regular expression `.*` to match any symbol.

```
>XL .*
```

The following example displays all symbols and their addresses in the program that start with `s_`:

```
>XE s_.*
```

! (Shell Escape)

Syntax

```
![[ [!]command]]
```

Parameter

command

Executes the given program or operating-system command without leaving CodeView. Use the second exclamation point to return to CodeView immediately after completing *command*.

Description

The Shell Escape (!) command (CV only) allows you to exit from the CodeView debugger to an MS-DOS shell. You can execute MS-DOS commands or programs from within the debugger, or you can exit from CodeView to MS-DOS while retaining your current debugging context.

If you want to exit to MS-DOS and execute several commands or programs, enter the Shell Escape command with no arguments (!). After the MS-DOS screen appears, you can run internal system commands or programs. When you are ready to return to CodeView, 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 an internal system command from within CodeView, enter the Shell Escape command followed by the name of the command or program you want to execute, as in:

```
!command
```

The output screen appears, and CodeView executes the command or program. When the output from the command or program is finished, 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. To suppress this prompt and

return directly to CodeView after the command is executed, use two exclamation points (!!) for the Shell Escape command.

The Shell Escape command works by executing a second copy of COMMAND.COM.

Mouse and Keyboard

In addition to typing the ! command, you can also invoke a command shell from the File menu.

Remarks

Opening a shell requires a significant amount of free memory since the following are all resident in memory:

- CodeView
- The debugging information
- The system's command processor
- The program being debugged

If your machine does not have enough memory, an error message appears. Even if there is enough memory to start a new shell, there may not be enough memory left to execute large programs from the shell.

In order for you to use the Shell Escape commands, the executable file being debugged must release unneeded memory. Programs created with Microsoft compilers release memory during startup.

Side effects of commands executed while in a shell, such as a change in the working directory, may not be seen when you return to CodeView.

Example

In the following example, the shell command **DIR** is executed with the argument `A:*.OBJ`. The directory listing will be followed by the prompt that asks you to press any key:

```
!DIR A:*.OBJ
```

In the following example, the **COPY** command is executed and control returns to CodeView. No prompt appears.

```
!!copy output.txt d:\backup
```

" (Pause)

Syntax "

Description The Pause (") command interrupts the 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 The following example is from a text file redirected to 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
"
```

(Tab Set)

Syntax #*number*

Parameter *number*
Number of characters for new tab stops. The valid range for *number* is 1–19.

Description 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 can specify values in the range 1–19.

You may 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 contains no tab characters.

* (Comment)

Syntax **comment*

Description 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. Use this command in combination with the redirection commands when you are:

- Saving a commented session.
- Writing a commented session that will be redirected to the debugger.

Example In the following example, the user is sending a copy of a CodeView session to the file OUTPUT.TXT. Comments are added to explain the purpose of the command. The text file will contain commands, comments, and command output.

```
> T>OUTPUT.TXT
> * Dump first 20 bytes of screen buffer
> MDB 0xB800:0 L 20
B800:0000 54 17 6F 17 20 17 72 17 65 17 74 17 75 17 72 17
B800:0010 6E 17 20 17
> >CON
```

. (Current Location)

Syntax .

Description The Current Location (.) command displays the source line or assembly-language instruction corresponding to the current program location. It puts the current program location in the center of the active Source window.

Use this command if you have scrolled the current source line or assembly instruction out of the active Source window.

The Current Location (.) command is equivalent to the command:

```
VS .
```

/ (Search)

Syntax / [[*regexp*]]

Parameter *regexp*
 Searches for the first line containing this regular expression. If omitted, the command searches for the next occurrence of the last regular expression given.

Description The Search (/) command searches for a regular expression in a source file.

“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 MS-DOS method of using wild card characters in filenames.

CodeView’s regular expressions use a subset of the UNIX syntax supported by the Programmer’s WorkBench (PWB). For complete information on regular expressions in PWB and CodeView, see Appendix A.

When you enter the Search command with a regular expression, CodeView searches the source file for the first line containing the expression. If you do not give a regular expression, CodeView searches for the next occurrence of the last regular expression specified.

Even if you do not understand regular expressions, you can still use the Search command with plain text strings, since text strings are the simplest form of regular expressions. For example, you can enter

```
>/ COUNT
```

to search for the word `COUNT` in the source file.

To find strings containing a special regular expression character (`.\^$*[]`), you must precede the character with a backslash (`\`); this cancels their special meanings. For example, use the command:

```
>/ x\*y
```

to find the string `x*y`.

In Source windows, CodeView starts searching at the current cursor position and places the cursor at the line containing the regular expression. The search wraps to the beginning of the file if necessary.

Mouse and Keyboard

In addition to typing the / command, you can also search for regular expressions by choosing Find from the Search menu.

Remarks When you search for the next occurrence of a regular expression, CodeView searches to the end of the file, then wraps around and begins again at the start of the file. This search can have unexpected results if the expression occurs only once. For example, when you give the command repeatedly, there is no activity on the screen. Actually, CodeView is repeatedly wrapping around and finding the same expression each time.

The Case Sensitivity command on the Options menu and the Options Case Sense (OC) command affect regular expression searches.

If you want to find a label in your source code, you can also use the View Source (VS) command.

7 (8087)

Syntax 7

Description The 8087 (7) command dumps the contents of the math processor registers. If you do not have an 8087 or equivalent math processor chip, this command dumps the contents of the software-emulated registers.

Example The following example shows and describes the output from the 7 command:

```
cControl 037F (Closure=Projective Round=nearest, Precision=64-bit )
                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 op-code=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
```

Here, the lowercase *c* that precedes several lines of the output indicates that the coprocessor is in use. If this command had been used with an emulated coprocessor, an *e* would precede the lines. The following is a line-by-line description of the output from the 7 command:

Line 1 This line shows the value in the control register, 037F. The rest of the line interprets the bits represented by the number in the control register:

- The closure method, which can be projective or affine.
- The rounding method, which can be nearest (even), down, up, or chop (truncate to zero).
- The precision, which can be 64, 53, or 24 bits.

Line 2 This line lists the status of the exception-mask bits, described in the following table:

Name	Description
IEM	Interrupt enable
PM	Precision
UM	Underflow
OM	Overflow
ZM	Zero divide
DM	Denormalized operand
IM	Invalid operation

Line 3 This line lists the value in the status register (6004 hexadecimal), the condition code (1000 binary), and the top of stack register (4 decimal). It then lists the exception flags, described in the following table:

Flag	Meaning
PE	Precision
UE	Underflow
OE	Overflow
ZE	Zero divide
DE	Denormalized operand
IE	Invalid operation

Line 4 This line lists the 20-bit address of the tag register, the offset of the instruction, the offset of the operand, and the offset of the op-code, all in hexadecimal. When using software-emulated coprocessor routines, this line does not appear.

Lines 5–9 The rest of the output from the 8087 command lists the contents of the stack register. In this example, ST(3) contains the value infinity, ST(2) contains a value that cannot be interpreted as any number, ST(1) contains a real number, and ST(0) contains zero.

: (Delay)

Syntax :

Description 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 In the following example, the Delay (:) command is used to slow execution of the redirected file into CodeView.

```
: ;* That was a short delay...  
:::: ;* That was a longer delay...
```

< (Redirect CodeView Input)

Syntax <*device*

Parameter *device*
 Device or file from which to read commands.

Description The Redirect Input (<) command causes CodeView to read all subsequent command input from a file or device.

Example The following example redirects command input from the file INFILE.TXT to CodeView. Use this method to run “scripts” of CodeView commands that you have prepared in advance.

```
> <INFILE.TXT
```

You can also start up CodeView with redirected input by typing the following at the operating-system prompt:

```
CV /C"<infile.txt"
```

> (Redirect CodeView Output)

Syntax `[[T]]>[>] device`

Parameter *device*
Device or file to which to write output.

Description The Redirect Output (>) command causes CodeView to write all subsequent command output to a device, such as another terminal, a printer, or a file. The term “output” includes not only 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. If you do not give a **T**, CodeView echoes only commands that you enter. Use the **T** option if you are redirecting output to a file to see output from the commands that you are typing.

Note If you are redirecting output to another terminal, you may not want to see the output on the CodeView terminal.

If you specify an existing file, CodeView truncates the file and then starts writing output. To preserve the contents of the file, use a second greater-than symbol (>>), which appends output to the file.

Example In the following example, output is redirected to the device designated as COM1 (for example, a remote terminal). Enter this command when you are debugging a graphics program and you want CodeView commands to be displayed on a remote terminal while the program display appears on the originating terminal.

```
> >COM1
```

In the following example, output is redirected to the file OUTFILE.TXT. Use this command to keep a permanent record of a CodeView session.

```
> T>OUTFILE.TXT
```

```
  . . .  
> >CON  
  . . .
```

Note 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, use the command >CON to return output to the terminal.

= (Redirect CodeView Input and Output)

Syntax = *device*

Parameter *device*
Device to which to redirect input and output. Specify >CON for the CodeView Command window.

Description The Redirect Input and Output (=) command causes the CodeView debugger to read all subsequent command input from the device and write all subsequent output to the device. You cannot redirect both input and output to a file.

To reset the input and output for CodeView after you have entered one of the other redirection commands, use the command:

>= con

? (Display Expression)

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

Parameters *expression*
The expression to display. This can be any valid CodeView expression.

,format
A CodeView format specifier that indicates the format in which to display *expression*.

Description The Display Expression (?) command displays the value of a CodeView expression. The simplest form of expression is a symbol representing a single variable or function. An expression may also call functions that are part of the executable file.

The Display Expression command can also set values. For example, with the C or C++ expression evaluator, you can increment the variable *n* by using an assignment expression:

? n=n+1

The command displays the value after incrementing *n*.

You can specify the format in which the values of expressions are displayed by the Display Expression command. After the expression, type a comma, followed by a CodeView format specifier.

Example

The following example 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, then in 4-byte hexadecimal, and then in octal.

```
>? amount
500
>? amount,x
01f4
>? amount,lx
000001f4
>? amount,o
764
>
```

?? (Quick Watch)

Syntax *?? symbol*

Parameter *symbol*
 Displays the given variable, array, or structure in the Quick Watch dialog box.

Description The Quick Watch (??) command displays the value of any selected expression in the Quick Watch dialog box. You can use Quick Watch to quickly check the value of a variable or structure and expand or contract items in a structure.

Expanding/Contracting Items

The Quick Watch dialog box allows you to:

- Expand or contract nested structures and arrays.
- View variables, structures, or arrays addressed by pointers.
- Add any structure or array to the Watch window.

Expandable items appear with a plus sign (+) in the Quick Watch dialog box. Once expanded, an item appears with a minus sign (-).

Expanding an item has the following effects:

Item	Action
Nested structure	Expands the structure so that the dialog box displays each member of the nested structure.
Pointer	Dereferences the pointer; that is, displays the data that the pointer addresses.
Array	Expands the array so that the dialog box displays each element of the array.

Contracting an item reverses the effects of expanding described above.

Note You can add any expression in the Quick Watch dialog box to the Watch window by choosing the Add Watch button.

Mouse and Keyboard

After opening the Quick Watch dialog, you can expand or contract an item using the following methods:

- Double-click the left mouse button on the item.
- Select the item, then choose the Expand/Contract button at the bottom of the dialog box.
- Use the arrow keys to select the item, and press ENTER.

@ (Redraw)

Syntax @

Description The Redraw (@) command redraws the CodeView screen. Use this command if the output of the program being debugged disturbs the CodeView display.

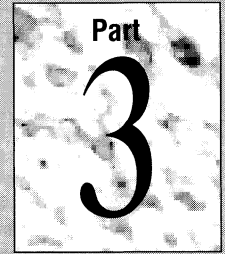
\ (Screen Exchange)

Syntax \

Description The Screen Exchange (\) command allows you to switch temporarily from the debugging screen to the output screen. The CodeView debugger uses either screen flipping or screen swapping to store the output and debugging screens.

To return to the CodeView screen, press any key.

Compiling and Linking



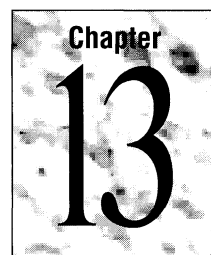
Chapter 13	CL Command Reference	485
14	Linking Object Files with LINK	561
15	Creating Overlaid DOS Programs	597
16	Creating Module-Definition Files	607
17	Using EXEHDR	629

Compiling and Linking

Microsoft C/C++ provides you with the tools to compile and link your programs. You can run these tools on the DOS command line, or you can use them in the form built into the PWB environment.

Chapter 13 describes the command-line syntax and options for the CL compiler. Chapter 14 covers LINK command-line syntax and options. Chapter 15 describes how to overlay a DOS program. Chapter 16 explains the contents and use of module-definition files. Chapter 17 describes how to use EXEHDR to examine the file header of a program.

CL Command Reference



This chapter describes the CL command, which you use to compile and link C and C++ programs. It explains the rules for giving input on the CL command line and describes the CL options in alphabetical order.

The chapter provides reference material for programmers who are familiar with the Microsoft C Compiler in general and the CL command in particular.

13.1 The CL Command Line

The CL command line has the following format:

```
CL [[option...]] file... [[option]file]... [[lib...]] [/link link-opt...]
```

The following list describes input to the CL command:

Entry	Meaning
<i>option</i>	One or more CL options; see “CL Options” on page 488 for descriptions.
<i>file</i>	The name of one or more source files, object files, or libraries. You must specify at least one filename. CL compiles source files and passes the names of the object files and libraries to the linker.
<i>lib</i>	One or more library names. CL passes the names of these libraries to the linker for processing.
<i>link-opt</i>	One or more of the linker options described in Chapter 14, “Linking Object Files with LINK.” The CL command passes these options to the linker for processing.

You can specify any number of options, filenames, and library names, as long as the length of the command line does not exceed the limit dictated by the operating system.

A filename can contain any combination of uppercase and lowercase letters, hyphens, underscores, and a period. Any filename can include a full or partial path.

A full path includes a drive name of the form D: and one or more directory names separated by backslashes (\). A partial path omits the drive name, which CL assumes to be the current drive. If you don't specify any path, CL assumes the file is in the current directory.

CL determines how to process each file depending on its filename extension, as follows:

Extension	Processing
.C	CL assumes the file is a C source file and compiles it.
.CXX, .CPP	CL assumes the file is a C++ source file and compiles it.
.OBJ	CL assumes the file is an object file and passes its name to the linker.
.LIB	CL assumes the file is a library and passes its name to the linker. The linker links this library with the object files CL created from source files and the object files given on the command line.
.ASM	CL assumes the file is an assembly file and invokes MASM (ML.EXE) to assemble it.
.DEF	CL assumes the file is a definition file and passes its name to the linker.
Any other extension or no extension	CL assumes the file is an object file and passes its name to the linker.

13.2 How the CL Command Works

The CL command uses the following procedure to create an executable file from one or more C source files:

1. CL compiles each source file, creating an object file for each. CL automatically includes the options listed in the CL environment variable. In each object file, CL places the name of the appropriate standard combined library. The library's name reflects both the memory model and the floating-point math package used to compile the program. See "/A Options" on page 488 for more information on the library names.
2. CL invokes the linker, passing the names of the object files it has created plus the name of any object files or libraries given on the command line. CL automatically includes the options listed in the LINK environment variable. If you use /link to specify linker options on the CL command line, these options apply as well. If conflicts occur, options that follow /link override those in the LINK environment variable.

3. The linker links the object files and libraries named by CL to create a single executable file.

Before it creates the executable file, the linker resolves “external references” in the object files. An external reference is a function call in one object file that refers to a function defined in another object file or in a library. To resolve an external reference, the linker searches for the called function in the following locations in the following order:

- a. The object files passed by CL
- b. The libraries given on the CL command line, if any
- c. The libraries named in the object files

Example

Assume that you are compiling three C source files: MAIN.C, MOD1.C, and MOD2.C. Each file includes a call to a function defined in a different file:

- MAIN.C calls the function named `func1` in MOD1.C and the function named `func2` in MOD2.C.
- MOD1.C calls the standard library functions **`printf`** and **`scanf`**.
- MOD2.C calls graphics functions named **`myline`** and **`mycircle`**, which are defined in a library named MYGRAPH.LIB.

First, compile with a command line of the following form:

```
CL MAIN.C MOD1.C MOD2.C MYGRAPH.LIB
```

CL first compiles the C source files and creates the object files MAIN.OBJ, MOD1.OBJ, and MOD2.OBJ. The compiler places the name of the standard library SLIBCE.LIB in each object file.

Next, CL passes the names of the C source files to the linker. Finally, the linker resolves the external references as follows:

1. In MAIN.OBJ, the reference to the function `func1` is resolved using the definition in MOD1.OBJ; the reference to the function `func2` is resolved using the definition in MOD2.OBJ.
2. In MOD1.OBJ, the references to **`printf`** and **`scanf`** are resolved using the definitions in SLIBCE.LIB. The linker uses this library because it finds the library name within MOD1.OBJ.
3. In MOD2.OBJ, the references to `myline` and `mycircle` are resolved using the definitions in MYGRAPH.LIB.

13.3 CL Options

Options to the CL command consist of either a forward slash (/) or a dash (-) followed by one or more letters. Certain CL options take arguments; in some of these options, a space is required between the option and the argument; in others, no space is allowed. The spacing rules for the options are given in their descriptions.

Note CL options (except for the /HELP option) are case sensitive. For example, /C and /c are different options.

Except for /f, /qc, and /Oq, options can appear anywhere on the CL command line—these three must appear first. With two exceptions (/c, /Fe), each CL option applies to the files that follow it on the command line and does not affect files preceding it on the command line.

You can also define CL options in the CL environment variable; these options are used every time you invoke CL. See “Specifying Options with the CL Environment Variable” on page 557.

The remainder of this section describes all the CL options in alphabetical order. If an option can take one or more arguments, its format is shown under an “Option” heading before its description.

/A Options (Memory Models)

Every program’s code and data are stored in blocks called “segments.” The memory model of the program determines the organization of these segments. (See Chapter 16 for more information on segments.) The memory model also determines what kind of executable file is generated. All models except tiny produce an .EXE file. The tiny model produces a .COM file. Any version of CL that targets a 32-bit system offers only the small memory model (/AS). CL offers the memory-model options described in Table 13.1.

Table 13.1 Memory Models

CL Option	Memory Model	Data Segments	Code Segments	Long Form
/AT	Tiny	One segment for both data and code	One segment for both data and code	none
/AS	Small	One	One	/Asnd
/AM	Medium	One	One code segment per module	/AInd
/AC	Compact	Multiple data segments; data items must be smaller than 64K	One	/Asfd

Table 13.1 (continued)

CL Option	Memory Model	Data Segments	Code Segments	Long Form
/AL	Large	Multiple data segments; data items must be smaller than 64K	One code segment per module	/Alfd
/AH	Huge	Multiple data segments; arrays can be larger than 64K	One code segment per module	/Alhd

By default, the compiler uses the small memory model.

For programs that target 16-bit processors, memory models with multiple code segments can accommodate larger programs than can memory models with one code segment. Memory models with multiple data segments can accommodate more data-intensive programs than can memory models with one data segment. Programs with multiple code or data segments, however, are slower than programs with a single code or data segment. A program destined to be a Windows DLL can use only a single data segment.

For 32-bit target code, memory is no longer partitioned into hardware-defined 64K segments. The compiler still partitions each program into one code segment and one data segment; the size of the segments is limited only by the amount of memory addressable by a 32-bit pointer—about 4 gigabytes.

It is more efficient to compile with the smallest possible memory model and use the `__near`, `__far`, `__huge`, and `__based` keywords to override the default addressing conventions for any data items or functions that cannot be accommodated in that model. For more information, see Chapter 4 in the *Programming Techniques* manual.

CL also supports customized memory models, in which different features from standard memory models are combined. You specify a customized memory model with the `/Astring` option, where *string* is composed of three letters that specify the code pointer size, the data pointer size, and the stack and data segment setup, respectively. All three letters must be present, but they can appear in any order. The allowable letters appear in Table 13.2.

Table 13.2 Customized Memory Model Codes

Group	Code	Description
Code pointers	s	Small
	l	Large
Data pointers	n	Near
	f	Far
	h	Huge
Segment setup	d	SS == DS
	u	SS != DS; DS loaded for each function entry
	w	SS != DS; DS not loaded at function entry

As examples, the customized representations of the standard memory models appear in the Long Form column of Table 13.1.

The segment setup codes can also be given as separate options when used to modify a standard memory model. For example, the options `/ASu` specify the small model and force DS to be loaded at function entry.

The memory-model and math options used to compile the program determine the library the linker searches to resolve external references. The library name is `mLIBCf.LIB`, where the memory-model option determines *m*: S for small (default) or tiny model, M for medium model, C for compact model, or L for large or huge model, and the math option determines *f*: E for emulator (default), A for alternate math, or 7 for 8087/80287. For example, the following specifies a small-model library with coprocessor support:

```
MLIBC7.LIB
```

For more information on the math options, see “/FP Options” on page 508.

/batch (Compile in Batch Mode)

The `/batch` option assumes that CL is being executed from a batch file. Usually, if CL cannot find one of the compiler pass files (for instance, C1.EXE or C2.EXE) or one of the utilities (such as LINK.EXE or CVPACK.EXE), it displays a prompt requesting the appropriate full-path filename. If you specify the `/batch` option, CL simply terminates compilation with an error.

/Bm (Increasing Compiler Capacity)

Option `/Bmmemavailable`

Use the `/Bm` option to increase the amount of memory that is available to the compiler during its second pass. Use of this option is necessary only if you encounter

the “Function too large to optimize” error message. The default memory limit that the compiler works within is 2048K. The *memavailable* argument accepts the new amount (in kilobytes) of memory; thus a *memavailable* value of 4096 makes 4 megabytes of memory available. A space between */Bm* and *memavailable* is optional.

/c (Compile Without Linking)

The */c* option instructs CL to compile all C source files given on the command line, creating object files; it does not link the object files. When you specify the */c* option, CL does not produce an executable file. Regardless of its position on the command line, this option applies to all source files on the command line.

Example

```
CL /c FIRST.C SECOND.C THIRD.OBJ
```

This example compiles *FIRST.C*, creating the object file *FIRST.OBJ*, and then compiles *SECOND.C*, creating the object file *SECOND.OBJ*. No processing is performed with *THIRD.OBJ* because CL skips the linking step.

/C (Preserve Comments During Preprocessing)

The */C* (for “comment”) option preserves comments during preprocessing with the */E*, */P*, or */EP* options. If you do not specify the */C* option, the preprocessor does not pass source-file comments to its output file.

This option is valid only if the */E*, */P*, or */EP* option is also used. These CL options are described later in this chapter.

Example

```
CL /P /C SAMPLE.C
```

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

/D (Define Constants and Macros)

Option */D**identifier*[[=*number*]]

Use the */D* option to define constants or macros for your source file.

The *identifier* is the name of the constant or macro. It can be defined as a string or as a number. No space can separate `/D` and the *identifier*. Enclose the *string* in quotes if it includes spaces.

Note that with Microsoft C/C++, you can substitute a number sign (`#`) for an equal sign (`=`) when setting the *identifier* to *string|number*. From either the DOS command line or from a batch file, you cannot set an environment variable, such as `CL`, to a string that contains an equal sign. Environment variables do, however, accept the number sign. Now that the `CL` driver allows the substitution of `"#"` for `"="`, you can use the `CL` environment variable to define preprocessor constants:

```
SET CL="/DTEST#0"
```

If you omit both the equal sign and the string or number, the identifier is assumed to be defined, and its value is set to 1. For example, entering `/DSET` defines a macro named `SET` with a value of 1. Note that the *identifier* argument is case sensitive. For example, the preceding `/D` option would have no effect on a constant named `set` that is defined in the source file.

Use the `/D` option in combination with either the `#if` or `#ifdef` directive to compile source files conditionally.

You can replace a keyword, identifier, or a numeric constant with no text in a source file. To do so, use the `/D` option with a keyword, identifier, or a numeric constant and append an equal sign followed by a space. For example, Use the following command to remove all occurrences of `RELEASE` from `TEST.C`:

```
CL /DRELEASE= TEST.C
```

Similarly, use the following command to remove all occurrences of the keyword `__far` in `TEST.C`:

```
CL /D__far= TEST.C
```

Defining macros and constants with the `/D` option has the same effect as using a `#define` preprocessor directive at the beginning of your source file. The identifier is defined until either an `#undef` directive in the source file removes the definition or until the compiler reaches the end of the file.

If an identifier defined in a `/D` option is also defined within the source file, `CL` uses the definition on the command line until it encounters the redefinition of the identifier in the source file.

Example

```
#if !defined(RELEASE)
    __nheapchk();
#endif
```

This code fragment calls a function to check the near heap unless the constant `RELEASE` is defined. While developing the program, you can leave `RELEASE` undefined and perform heap checking to find bugs. Assuming the program name is `BIG.C`, you would compile with the following command:

```
CL BIG.C
```

After you have found all of the bugs in the program, you can define `RELEASE` in a `/D` option so the program runs faster, as follows:

```
CL /DRELEASE BIG.C
```

/E (Copy Preprocessor Output to Standard Output)

The `/E` option preprocesses the C source file and copies preprocessor output to the standard output device (usually your terminal). The `/E` option adds **#line** directives to the output. The **#line** directives are placed at the beginning and end of each included file and around lines removed by preprocessor directives that specify conditional compilation. You can use the `/EP` option, discussed below, to suppress the addition of **#line** to the output.

The output that the `/E` option generates is identical to the original source file except that all preprocessor directives are carried out, macro expansions are performed, and comments are removed. You usually use the `/E` option with the `/C` option (see “`/c (Compile Without Linking)`” on page 491), which preserves comments in the preprocessed output. You can use DOS redirection to save the output in a disk file.

Use this option when you want to resubmit the preprocessed listing for compilation. The **#line** directives renumber the lines of the preprocessed file so that errors generated during later stages of processing refer to the line numbers of the original source file rather than to the preprocessed file.

The `/E` option suppresses compilation. It also suppresses production of the alternate output files that the `/Fa`, `/Fc`, `/Fm`, or `/Fo` options generate.

Example

```
CL /E /C ADD.C > PREADD.C
```

This command creates a preprocessed file with inserted **#line** directives from the source file `ADD.C`. The output is redirected to the file `PREADD.C`.

Note Precompiled headers do not work with the `/E` option.

/EP (Copy Preprocessor Output to Standard Output)

The `/EP` option is similar to the `/E` option: it preprocesses the C source file and copies preprocessor output to the standard output device. Unlike the `/E` option, however, the `/EP` option does not add **#line** directives to the output. You can use the `/EP` option with the `/P` option, discussed later in this chapter, to suppress adding **#line** directives to the output.

Preprocessed output is identical to the original source file except that all preprocessor directives are carried out, macro expansions are performed, and comments are removed. You can use the `/EP` option with the `/C` option (see “/c (Compile Without Linking)” on page 491), which preserves comments in the preprocessed output.

The `/EP` option suppresses compilation; CL does not produce an object file or a map file, even if you specify the `/Fo` or `/Fm` option on the CL command line. Also, it will not produce the output files specified by the `/Fa` or the `/Fc` options.

Example

```
CL /EP /C ADD.C
```

This command creates a preprocessed file from the source file `ADD.C`. It preserves comments but does not insert **#line** directives. The output appears on the screen.

/F (Set Stack Size)

Option */F number*

The `/F` option sets the program stack size to *number* bytes, where *number* is a hexadecimal number in the range 0001 to FFFF. A space is required between the option and *number*. Decimal and octal numbers are not allowed. If you don't specify this option, a stack size of 2K is used by default.

You may want to increase the stack size if your program gets stack-overflow diagnostic messages. If your program uses little stack space, you may want to decrease the size of your program by reducing the stack size. In general, if you modify the stack size, do not use the `/Gs` option to suppress stack checking until you are sure the new stack size is large enough.

/f (Fast Compile)

Use the `/f` option to compile source files without any default optimizations. It generates the `_FAST` preprocessor constant. Programs compiled with `/f` are

slower and larger but compile in less time than the optimizing compiler requires; this option is useful during the development process. If used, the /f option must be the first option on the command line. The Fast Compile option does not support initialized static huge data.

Note that with Microsoft C/C++, the /f option supersedes the /qc option.

/Fo, /Fe, /Fs, /Fa, /Fl, /Fc, /Fm, /Fp, /Fr, /FR (Set Alternate Output Files)

Option

/Fo filename

/Fe filename

/Fs [[filename]]

/Fa [[filename]]

/Fl [[filename]]

/Fc [[filename]]

/Fm [[filename]]

/Fp [[filename]]

/Fr [[filename]]

/FR [[filename]]

Use these options to specify alternate names for output files. Table 13.3 summarizes the purpose of each option. This section begins with information that is true for all of the /F options and ends with information specific to each individual option.

Table 13.3 Optional File Types

Option	File Type	Default File Name	Default Extension
/Fo	Object	Base name of source file plus .OBJ	.OBJ
/Fe	Executable	Base name of source file plus .EXE	.EXE
/Fs	Source listing	Base name of source file plus .LST	.LST
/Fa	Assembly listing	Base name of source file plus .ASM	.ASM
/Fl	Assembler- and machine-code listing	Base name of source file plus .COD	.COD
/Fc	Source-, assembler-, and machine-code listing	Base name of source file plus .COD	.COD
/Fm	Linker map	Base name of source file plus .MAP	.MAP
/Fp	Precompiled header	Base name of source file plus .PCH	.PCH

Table 13.3 (continued)

Option	File Type	Default File Name	Default Extension
/Fr	PWB Source Browser database—ignores local variables	Base name of source file plus .SBR	.SBR
/FR	PWB Source Browser database	Base name of source file plus .SBR	.SBR

Path Specifications and Extensions

If used, the *filename* argument must follow the option with no intervening space. It can be a file specification, a drive name, or a path specification. If *filename* is a drive name or path specification, the CL command creates the specified file or files in the given location; the default name is the base name of the first file plus the default extension. If the *filename* argument is a path without a filename (i.e., a directory), end the path with a backslash (\) or CL cannot differentiate the path from a filename.

You can give any name and extension you want for *filename*. If you give a filename without an extension, CL automatically appends the default extension.

Since you can process more than one file at a time with the CL command, the order in which you give listing options and the kind of argument you give for each option affect the result. Only one kind of object or assembly listing can be produced for each source file. The following list summarizes the effects of each option with each type of argument.

Argument	Effect
Filename	Creates a listing for the next source file on the command line; uses default extension if no extension is given
Drive Name or Path	Creates listings in the given location for every source file listed after the option on the command line; uses default names
None	Creates listings in the current directory for every source file listed after the option on the command line; uses default names

Interactions Between Options

The following list summarizes the interactions between the options.

Option	Effect
/Fc	Overrides /Fa and /Fl
/Fa and /Fl	/Fl overrides /Fa
/Fa and /Fs	Produces combined listing
/Fm and /c	Linking is suppressed; /Fm is ignored
/Fe and /c	Linking is suppressed; /Fe is ignored
/Fs and /Fl	Produces combined source-, assembler-, and machine-code listing

DOS Device Names

You can append the MS-DOS device names AUX, CON, PRN, and NUL to the alternate output-file options and direct the resulting listing files to your terminal or printer. There can be no space between the option and the device name. For instance, the following command line generates assembler code for TEST.C and directs the output to the console:

```
CL /FaCON TEST.C
```

The following list summarizes the result when a device name is appended to an option:

Device Name	Result
AUX	Sends the listing file to an auxiliary device
CON	Sends the listing file to the console
PRN	Sends the listing file to a printer
NUL	No file is created

Note Do not append a colon (:) to the device names when you use them as arguments to the listing options. For example, use CON instead of CON: and PRN instead of PRN:.

Examples

```
CL /FsHELLO.SRC /FcHELLO.CMB HELLO.CPP
```

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

```
CL /FsHELLO.SRC /FsHELLO.LST /FcHELLO.CMB HELLO.CXX
```

The second example produces a source listing called HELLO.LST rather than HELLO.SRC, since the name associated with the rightmost option determines the resulting filename. This example also produces a combined source-code,

assembler-code, and machine-code listing file called HELLO.CMB. The object file has the default name HELLO.OBJ.

```
CL /FsPRN HELLO.CXX
```

In this example, CL sends a source listing to the printer.

/Fo (Rename Object File)

Option

/Fofilename

By default, CL gives each object file it creates the base name of the corresponding source file plus the extension .OBJ. The /Fo option lets you give different names to object files or create them in a different directory. No space is allowed between the option and *filename*.

If you are compiling more than one source file, you can use the /Fo option with each source file to rename the corresponding object file. You can also use the /Fo option with a directory name to place all of the object files in a different directory. Note that the /Fo option requires a *filename* argument.

You can give any name and extension you want for *filename*. However, it is recommended that you use the conventional .OBJ extension since the linker and the LIB library manager use .OBJ as the default extension when processing object files.

Examples

```
CL /FoB:\OBJECT\ THIS.C
```

In this example, CL compiles the source file THIS.C and gives the resulting object file the name THIS.OBJ by default. The directory specification B:\OBJECT\ tells CL to create THIS.OBJ in the directory named \OBJECT on drive B.

```
CL /c /Fo\ASM\ THIS.C THAT.C /Fo\SRC\NEWTHOSE.OBJ THOSE.C
```

In this example, the /c option tells CL to run the compiler and not the linker. The first /Fo option tells the compiler to generate two object files, THIS.OBJ and THAT.OBJ, from THIS.C and THAT.C and place them in the \ASM directory. The second /Fo option tells the compiler to create the object file NEWTHOSE.OBJ (generated from THOSE.C) in the \SRC directory. Note that any path appended to an option must end with a backslash (\ASM must be \ASM\).

/Fe (Rename Executable File)

Option

*/Fe**filename*

By default, CL names the executable file with the base name of the first file (source or object) on the command line plus the extension .EXE. The /Fe option lets you give the executable file a different name or create it in a different directory. Note that the /Fe option requires a *filename* argument.

Because CL creates only one executable file, you can type the /Fe option anywhere on the command line. If you enter more than one /Fe option, CL gives the executable file the name specified in the last /Fe option on the command line.

The /Fe option applies only in the linking stage. If you specify the /c option to suppress linking, /Fe has no effect.

Examples

```
CL /FeC:\BIN\PROCESS *.C
```

This example compiles and links all source files with the extension .C in the current directory. The resulting executable file is named PROCESS.EXE and is created in the directory C:\BIN.

```
CL /FeC:\BIN\ *.C
```

This example is similar to the first example except that the executable file is given the same base name as the first file compiled instead of being named PROCESS.EXE. The executable file is created in the directory C:\BIN.

/Fs (Create Source-File Listing)

Option

/Fs[[*filename*]]

The /Fs option produces a file that contains an annotated listing of the source file. This file lists and numbers every line in the source file, including lines of code, comment lines, and blank lines. The line numbers begin with 1. The file also lists the local and global symbols and classifies them by name, class, type, size, and offset. If the symbol is a register variable, the file notes the register used. Finally, the file lists the size of the data segments and whether any errors were detected. You can use the /Sl, /Sp, /St, and /Ss options (described in this chapter) with the /Fs option to specify a source listing's line width and page length and to provide a title and subtitle.

Note that the /Fs option is not supported by the Fast Compile option (/f).

A fragment of a sample source-file listing follows:

```

.
.
.
83     if( result == NULL )
84         printf( " Value %u not found\n", key );
85     else
86         printf( " Value %u found in element %u\n",
87             key, result - array + 1 );
88 }

main Local Symbols

Name                Class  Type                Size  Offset  Register

elements. . . . . auto                -0004
key . . . . . auto                -0002
i . . . . . auto                 ***      si
result. . . . . auto                 ***      di
.
.
.
cmpe Local Symbols

Name                Class  Type                Size  Offset  Register

key . . . . . param                0004
tableentry. . . . . param            0006

Global Symbols

Name                Class  Type                Size  Offset

array . . . . . common  struct/array      2000  ***
bsearch . . . . . extern  near function     ***   ***
cmpe . . . . . global  near function     ***   01cc
cmpgle . . . . . global  near function     ***   0196
lfind . . . . . extern  near function     ***   ***
main . . . . . global  near function     ***   0000
printf . . . . . extern  near function     ***   ***
qsort . . . . . extern  near function     ***   ***
rand . . . . . extern  near function     ***   ***
srand . . . . . extern  near function     ***   ***
time . . . . . extern  near function     ***   ***

Code size = 01ee (494)
Data size = 011a (282)
Bss size = 0000 (0)

No errors detected

```

Examples

```
CL /FsQSORT QSORT.C
```

This example compiles and links the source file QSORT.C and generates a source-listing file named QSORT.LST.

/Fa (Create Assembly-File Listing)

Option

```
/Fa[[filename]]
```

The /Fa option translates your C or C++ source code to assembly language. Unless you name the resulting file with the optional *filename* argument, it is given the base name of the C or C++ source file specified on the CL command line with an .ASM extension.

A fragment of a sample assembly-language listing follows:

```
.
.
.
_main PROC NEAR
; Line 28
    push    bp
    mov     bp,sp
    mov     ax,8
    call    __aNchkstk
;   i = -4
;   result = -8
;   elements = -2
;   key = -6
; Line 29
    mov     WORD PTR [bp-2],1000    ;elements
    mov     WORD PTR [bp-6],500    ;key
; Line 32
    xor     ax,ax
    push    ax
    call    _time
    add     sp,2
    push    ax
    call    _srand
    add     sp,2
; Line 35
    mov     ax,OFFSET DGROUP:$SG330
    push    ax
    call    _printf
    add     sp,2
; Line 36
    mov     WORD PTR [bp-4],0    ;i
```

```
$FC332:
; Line 37
    call    _rand
    mov cx,1000
    cwd
    idiv   cx
    inc dx
    mov bx,WORD PTR [bp-4] ;i
    shl bx,1
    mov WORD PTR _array[bx],dx
    inc WORD PTR [bp-4] ;i
    cmp WORD PTR [bp-4],cx ;i
    jb  $FC332
.
.
.
```

Example

```
CL /FaQSORT QSORT.C
```

This example compiles and links the source file QSORT.C and generates an assembly-listing file named QSORT.ASM.

/FI (Combined Assembly- and Machine-Code Listing)

Option

```
/FI[[filename]]
```

The /FI option translates your C or C++ source code to a combined assembly- and machine-code listing. Unless you name the resulting file with the optional *filename* argument, it is given the base name of the C or C++ source file specified on the CL command line with a .COD extension.

A fragment of a sample assembly- and machine-code listing follows:

```
.
.
.

_main PROC NEAR
; Line 28
    *** 000000 55          push    bp
    *** 000001 8b ec      mov bp,sp
    *** 000003 b8 08 00    mov ax,8
    *** 000006 e8 00 00    call   __aNchkstk
;   i = -4
;   result = -8
;   elements = -2
;   key = -6
```

```

; Line 29
*** 000009 c7 46 fe e8 03      mov WORD PTR [bp-2],1000      ;elements
*** 00000e c7 46 fa f4 01      mov WORD PTR [bp-6],500 ;key
; Line 32
*** 000013 33 c0                xor ax,ax
*** 000015 50                push ax
*** 000016 e8 00 00      call _time
*** 000019 83 c4 02      add sp,2
*** 00001c 50                push ax
*** 00001d e8 00 00      call _srand
*** 000020 83 c4 02      add sp,2
; Line 35
*** 000023 b8 00 00      mov ax,OFFSET DGROUP:$G330
*** 000026 50                push ax
*** 000027 e8 00 00      call _printf
*** 00002a 83 c4 02      add sp,2
; Line 36
*** 00002d c7 46 fc 00 00      mov WORD PTR [bp-4],0      ;i
                        $FC332:
; Line 37
*** 000032 e8 00 00      call _rand
*** 000035 b9 e8 03      mov cx,1000
*** 000038 99                cwd
*** 000039 f7 f9                idiv cx
*** 00003b 42                inc dx
*** 00003c 8b 5e fc      mov bx,WORD PTR [bp-4] ;i
*** 00003f d1 e3                shl bx,1
*** 000041 89 97 00 00      mov WORD PTR _array[bx],dx
*** 000045 ff 46 fc      inc WORD PTR [bp-4] ;i
*** 000048 39 4e fc      cmp WORD PTR [bp-4],cx ;i
*** 00004b 72 e5                jb $FC332
.
.
.

```

/Fc (Combined Source-, Assembly-, and Machine-Code Listing)

Option

/Fc[[*filename*]]

The **/Fc** option translates your C or C++ source code to a combined source-code, assembly-code, and machine-code listing. The file numbers and lists every line contained in your C or C++ source file and follows each line of source code with the code the compiler generates. Unless you name the resulting file with the optional *filename* argument, it is given the base name of the C or C++ source file specified on the CL command line with a .COD extension.

Following is a fragment of a combined source-code, assembly-code, and machine-code listing:

```

.
.
.
_main PROC NEAR
; *** /* QSORT.C illustrates randomizing, sorting, and searching. Functions
; *** * illustrated include:
; *** * srand rand qsort
; *** * _lfind _lsearch bsearch
; *** *
; *** * The _lsearch function is not specifically shown in the program, but
; *** * its use is the same as _lfind except that, if it does not find the
; *** * element, it inserts it at the end of the array rather than failing.
; *** */
; ***
; *** #include <search.h>
; *** #include <stdlib.h>
; *** #include <string.h>
; *** #include <stdio.h>
; *** #include <time.h>
; ***
; *** #define ASIZE 1000
; *** unsigned array[ASIZE];
; ***
; *** /* Macro to get a random integer within a specified range */
; *** #define getrandom( min, max ) ((rand() % (int)((max)+1) - (min))) +
(min))
; ***
; *** /* Must be declared before call */
; *** int __cdecl cmpgle( unsigned *elem1, unsigned *elem2 );
; *** int __cdecl cmpe( unsigned *key, unsigned *tableentry );
; ***
; *** void main()
; *** {
; Line 28
; *** 000000 55 push bp
; *** 000001 8b ec mov bp,sp
; *** 000003 b8 08 00 mov ax,8
; *** 000006 e8 00 00 call __aNchkstk
; i = -4
; result = -8
; elements = -2
; key = -6
; *** unsigned i, *result, elements = ASIZE, key = ASIZE / 2;
; Line 29
; *** 000009 c7 46 fe e8 03 mov WORD PTR [bp-2],1000 ;elements
; *** 00000e c7 46 fa f4 01 mov WORD PTR [bp-6],500 ;key
; ***
; *** /* Seed the random number generator with current time. */
; *** srand( (unsigned)time( NULL ) );

```

```

; Line 32
*** 000013 33 c0          xor ax,ax
*** 000015 50           push  ax
*** 000016 e8 00 00        call  _time
*** 000019 83 c4 02        add  sp,2
*** 00001c 50           push  ax
*** 00001d e8 00 00        call  _srand
*** 000020 83 c4 02        add  sp,2
.
.
.

```

Example

```
CL /FcQSORT QSORT.C
```

This example compiles and links the source file QSORT.C and also generates a combined source-code, assembly-code, and machine-code listing file named QSORT.COD.

/Fm (Create Map File)

Option

```
/Fm[[filename]]
```

The /Fm option produces a map file. The map file contains a list of segments in order of their appearance within the load module.

A fragment of a sample map file follows:

Start	Stop	Length	Name	Class
00000H	01E9FH	01EA0H	_TEXT	CODE
01EA0H	01EA0H	00000H	C_ETEXT	ENDCODE
.

The information in the `Start` and `Stop` columns shows the 20-bit address (in hexadecimal) of each segment, relative to the beginning of the load module. The load module begins at location zero. The `Length` column gives the length of the segment in bytes. The `Name` column gives the name of the segment, and the `Class` column gives information about the segment type. The starting address and name of each group appear after the list of segments. A sample group listing follows:

Origin	Group
01EA:0	DGROUP

In this example, **DGROUP** is the name of the data group. **DGROUP** is used for all near data (that is, all data not explicitly or implicitly placed in their own data segment) in Microsoft C/C++ programs.

The following map file contains two lists of global symbols: the first list is sorted in ASCII-character order by symbol name, and the second is sorted by symbol address. The notation `Abs` appears next to the names of absolute symbols (symbols containing 16-bit constant values that are not associated with program addresses).

```

Address          Publics by Name

01EA:0096        STKHQQ
0000:1D86        _brkctl
01EA:04B0        _edata
01EA:0910        _end
.
.
.
01EA:00EC        __abrkp
01EA:009C        __abrktb
01EA:00EC        __abrktbe
0000:9876 Abs    __acrtmsg
0000:9876 Abs    __acrtused
.
.
.
01EA:0240        ___argc
01EA:0242        ___argv

Address          Publics by Value

0000:0010        _main
0000:0047        _htoi
0000:00DA        _exp16
0000:0113        __chkstk
0000:0129        __astart
0000:01C5        __cintDIV
.
.
.

```

Global symbols in a map file usually have one or more leading underscores because the compiler adds an underscore to the beginning of variable names. Many of the global symbols that appear in the map file are symbols used internally by the compiler and the standard libraries.

The addresses of the external symbols show the location of the symbol relative to zero (the beginning of the load module).

Following the lists of symbols, the map file gives the program entry point, as shown in the following example:

```
Program entry point at 0000:0129
```

/Fp (Specify Precompiled Header Filename)

Option

/FPfilename

Option (/Fp)

Use the /Fp option to specify the name of the desired .PCH file in cases where the .PCH filename specified with the /Yc option is different than the filename of the associated include file or source file. For example, if you want to create a precompiled header file for a debugging version of your program, you can specify a command such as:

```
CL /DDEBUG /YcPROG /FcDPROG PROG.CPP
```

This command creates a precompilation of all header files up to and including PROG.H, and stores it in a file called DPROG.PCH. If you need a release version in parallel, change the compilation command to:

```
CL /YcPROG /FpRPROG PROG.CPP
```

This command creates a separate precompilation of the header files up to and including PROG.H and stores it in RPROG.PCH.

For more information on precompiled headers, see Chapter 2 in the *Programming Techniques* manual. The related options /Yc, /Yd, and /Yu are described later in this chapter.

/FR, /Fr (Generate PWB Browser Files from CL)

Option

/FR[[filename]]

/Fr[[filename]]

These options are part of the process you must follow to use the PWB Source Browser. Both options create a file with an .SBR extension; only one option is required. The BSCMAKE utility uses an .SBR file to generate a database file with a .BSC extension. This is the file used by the Source Browser.

An .SBR file contains symbolic information about your program. The /FR option generates complete symbolic information in the .SBR file. The /Fr option generates symbolic information without information on local variables. Both /FR and /Fr use the SBRPACK utility to compact the .SBR file by removing unreferenced definitions. As the BSCMAKE utility uses .SBR files to produce output for the PWB Source Browser, a smaller .SBR file gives BSCMAKE greater effective capacity and can also increase the speed of producing .BSC files. Smaller .SBR files save disk space.

By default, both options generate a filename that uses the source file's base name and appends an .SBR extension. Use the *filename* argument to provide a name other than the default; an .SBR extension is required. No space is allowed between either the /FR or the /Fr option and the *filename* argument.

The /Zn option, described later in this chapter, suppresses the CL driver's call to SBRPACK. The /Zn option is required if you want to use BSCMAKE's /lu option.

For information about using the PWB Source Browser to look for code in a project, to see where functions are invoked or where variables and types are used, or to generate call trees and cross-reference tables, see Chapter 21, "Browser Utilities."

Example

```
CL /FRNEWSORT QSORT.C
```

This example compiles and links the source file QSORT.C and generates a file named NEWSORT.SBR. The SBRPACK utility then compacts the .SBR file, and the BSCMAKE utility uses it to generate a .BSC file that you can examine with the PWB Source Browser.

/FP Options (Select Floating-Point-Math Package)

The /FPa, /FPc, /FPc87, /FPi, and /FPi87 options specify how your program handles floating-point-math operations. The following table summarizes the different options:

Table 13.4 Floating-Point Options

	Generated Code	Default Library	Link-Time Possibilities	Comments
/FPa	Function calls to <i>mLIBCA.LIB</i>	<i>mLIBCA.LIB</i>	<i>mLIBCE.LIB</i> <i>mLIBC7.LIB</i>	(4, 5)
/FPc	Function calls to <i>mLIBCE.LIB</i>	<i>mLIBCE.LIB</i>	<i>mLIBCA.LIB</i> <i>mLIBC7.LIB</i>	(5)

Table 13.4 (continued)

	Generated Code	Default Library	Link-Time Possibilities	Comments
<i>/FPc87</i>	Function calls to <i>mLIBC7.LIB</i>	<i>mmIBC7.LIB</i>	<i>mLIBCA.LIB</i> <i>mLIBCE.LIB</i>	(1, 5)
<i>/FPi</i>	Software interrupts	<i>mLIBCE.LIB</i>	<i>mLIBCE.LIB</i> <i>mLIBC7.LIB</i>	(2)
<i>/FPi87</i>	Inline 80x87 instructions	<i>mLIBC7.LIB</i>	<i>mLIBCE.LIB</i>	(1, 3)

1 Requires a math coprocessor if program is linked with *mLIBC7.LIB*.

2 The executable file interacts with the *NO87* environment variable allowing run-time selection of using either the emulator library or a math coprocessor.

3 Produces the smallest, fastest code if a coprocessor is available.

4 Produces the fastest code if no coprocessor is available.

5 Libraries that are compiled with this option can later be linked with any Microsoft floating-point library.

***/FPa* (Alternate Math Package)**

This option generates floating-point calls and selects the alternate math library for the current memory model (*mLIBCA.LIB*). Calls to this library provide the fastest and smallest code if you do not have an 80x87 coprocessor. This library provides full IEEE 64-bit precision using doubles and 32-bit precision using floats. Long doubles are not supported.

At link time, you can specify an emulator library (*mLIBCE.LIB*) or an 80x87 library (*mLIBC7.LIB*). These libraries provide 80-bit precision.

Note that neither the Fast Compile option (*/f*) nor the p-code option (*/Oq*) support the */FPa* option.

***/FPc* (Coprocessor)**

The */FPc* option generates floating-point calls to the emulator library and places the name of an emulator library (*mLIBCE.LIB*) in the object file.

At link time, you can specify an 80x87 library (*mLIBC7.LIB*) or alternate math library (*mLIBCA.LIB*) instead. The */FPc* option gives you flexibility in your choice of libraries for linking.

This option is recommended if you compile modules that:

- Perform floating-point operations (when you plan to include these modules in a library).
- Link with libraries other than the libraries provided with this compiler.

Note that certain optimizations are not performed when `/FPc` is used. This may reduce the efficiency of your code. The `/FPc` option is not supported with either the Fast Compile option (`/f`) or the p-code option (`/Oq`).

`/FPc87 (80x87 Calls)`

This option generates function calls to routines in the 80x87 library (`mLIBC7.LIB`) in order to perform the corresponding 80x87 instructions.

You must have an 80x87 coprocessor installed to run programs compiled with the `/FPc87` option and linked with an 80x87 library. Otherwise, the program fails and an error message is displayed.

If an 80x87 coprocessor is unavailable, you can specify an emulator library (`mLIBCE.LIB`) or the appropriate alternate math library (`mLIBCA.LIB`) at link time.

Note that certain optimizations are not performed when `/FPc87` is used. This may reduce the efficiency of your code. The `/FPc87` option is not supported with either the Fast Compile option (`/f`) or the p-code option (`/Oq`).

`/FPi (Emulator)`

Use the `/FPi` option if you do not know whether a math coprocessor will be available at run time. Programs compiled with `/FPi` work as follows:

- If a coprocessor is present at run time, the program uses the coprocessor.
- If no coprocessor is present or if the `NO87` environment variable has been set, the program uses the emulator.

The `/FPi` option generates inline instructions for an 80x87 coprocessor and places the name of the emulator library (`mLIBCE.LIB`) in the object file. At link time, you can specify an 80x87 library (`mLIBC7.LIB`) instead. If you do not choose a floating-point option, CL uses the `/FPi` option by default.

This option works whether or not a coprocessor is present because the compiler does not generate “true” inline 80x87 instructions. Instead, it generates software interrupts to library code. The library code, in turn, adds fixups to the interrupts to select either the emulator or the coprocessor, depending on whether a coprocessor is present.

/FPi87 (Coprocessor)

The `/FPi87` option includes the name of an 80x87 library (`mLIBC7.LIB`) in the object file. At link time, you can override this option and specify an emulator library (`mLIBCE.LIB`) instead so that the program runs on computers without coprocessors.

If you use the `/FPi87` option and link with `mLIBC7.LIB`, an 8087 or 80287 coprocessor must be present at run time; otherwise, the program fails and the following error message is displayed:

```
run-time error R6002
- floating point not loaded
```

If you compile with `/FPi87` and link with `mLIBCE.LIB`, you can set the `NO87` environment variable to suppress the use of the coprocessor.

Compiling with the `/FPi87` option results in the smallest, fastest programs possible for handling floating-point arithmetic.

Library Considerations for Floating-Point Options

You may want to use libraries in addition to the default library for the floating-point option you have chosen on the CL command line. For example, you may want to create your own libraries or object files, then link them at a later time with object files that you have compiled using different CL options.

You must be sure that you use only one standard combined C library when you link. You can control which library the linker uses in one of two ways:

- Make sure the first object file passed to the linker has the name of the desired library. For example, if you want to use an 80x87 library, give the `/FPi87` option before the first source-file name on the CL command line. You can also give the name of an object file compiled with `/FPi87` as the first filename on the command line. All floating-point calls in this object file refer to the 80x87 library.
- Give the `/NOD` (no default-library search) option after the `/link` option on the CL command line. Then specify the name of the library you want to use on the CL command line. The `/NOD` option overrides the library names embedded in the object files. Because the linker searches libraries given on the command line before it searches libraries named in object files, all floating-point calls refer to the libraries you specify.

Another complication might arise if you create your own libraries: usually, each module in the library you create contains a standard-library name, and the linker tries to search the standard libraries named in the modules when it links with your library.

The safest course, especially when you are distributing libraries to others, is to use the `/Zl` option when you compile the object files that make up your libraries. The `/Zl` option tells the compiler not to put library names in the object files. Later, when you link other object files with your library, the standard library used for linking depends only on the floating-point and memory-model options used to compile those object files.

Examples

```
CL CALC.C ANOTHER SUM
```

In this example, no floating-point option is given, so `CL` compiles the source file `CALC.C` with the default floating-point option, `/FPi`. The `/FPi` option generates inline instructions and selects the small-model-emulator combined library (`SLIBCE.LIB`), which is the default.

```
CL /FPi87 CALC.C ANOTHER.OBJ SUM.OBJ SLIBCE.LIB /link /NOD
```

In this example, `CL` compiles `CALC.C` with the `/FPi87` option, which selects the `SLIBC7.LIB` library. The `/link` option, however, overrides the default library specification: the `/NOD` option suppresses the search for the default library, and the emulator library (`SLIBCE.LIB`) is specified. `LINK` uses `SLIBCE.LIB` when it creates the resulting executable file, `CALC.EXE`.

Compatibility Between Floating-Point Options

Each time you compile a source file, you can specify a floating-point option. When you link two or more source files to produce an executable program file, you are responsible for ensuring that floating-point operations are handled in a consistent way.

Example

```
CL /AM CALC.C ANOTHER SUM /link MLIBC7.LIB /NOD
```

This example compiles the program `CALC.C` with the medium-model option (`/AM`). Because no floating-point option is specified, the default (`/FPi`) is used. The `/FPi` option generates inline 80x87 instructions and specifies the emulator library `MLIBCE.LIB` in the object file. The `/link` field specifies the `/NOD` option and the name of the medium-model 80x87 library, `MLIBC7.LIB`. Specifying the 80x87 library forces the program to use an 8087 coprocessor; the program fails if a coprocessor is not present.

The NO87 Environment Variable

Programs compiled with the /FPi option automatically use an 80x87 coprocessor at run time if one is installed. You can override this and force the use of the emulator instead by setting an environment variable named NO87.

If NO87 is set to any value when the program is executed, the program uses the emulator even if a coprocessor is present. When this occurs, the NO87 setting is displayed on the standard output device as a message. The message is displayed only if a coprocessor is present and its use is suppressed; if no coprocessor is present, no message appears. If you want to force use of the emulator but don't want a message to appear, set NO87 equal to one or more spaces. The variable is still considered to be defined.

Note that the presence or absence of the NO87 definition determines whether use of the coprocessor is suppressed. The actual value of the NO87 setting is used only for the message.

The NO87 variable takes effect with any program linked with an emulator library (*mLIBCE.LIB*). It has no effect on programs linked with 8087/80287 libraries (*mLIBC7.LIB*).

Examples

```
SET NO87=Use of coprocessor suppressed
```

This example causes the message *Use of coprocessor suppressed* to appear when a program that would use an 80x87 coprocessor is executed on a computer that has such a coprocessor.

```
SET NO87=space
```

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

Standard Combined Libraries

Table 13.5 shows each combination of memory-model and floating-point options and the corresponding library name that CL embeds in the object file.

Table 13.5 CL Options and Default Libraries

Floating-Point Option	Memory-Model Option	Default Library
/FPi87	/AT or /AS	SLIBC7.LIB
	/AM	MLIBC7.LIB
	/AC	CLIBC7.LIB
	/AL or /AH	LLIBC7.LIB
/FPi	/AT or /AS	SLIBCE.LIB
	/AM	MLIBCE.LIB
	/AC	CLIBCE.LIB
	/AL or /AH	LLIBCE.LIB

/G0, /G1, /G2, /G3, /G4 (Generate Processor-Specific Instructions)

If you are writing programs for a machine with an 8086/8088, 80186/80188, 80286, 80386, or 80486 processor, you can use the /G0, /G1, /G2, /G3, or /G4 option, respectively, to enable the instruction set for those processors. When you use /G2 and /G3 options, the compiler automatically defines the appropriate **M_I286** or **M_I386** identifier.

Although it is sometimes advantageous to enable the appropriate instruction set, you may not always want to do so. If you have an 80286 processor, for example, but you want your program to be able to run on an 8086/8088, do not compile with the /G2 option.

The /G0 option enables the instruction set for the 8086/8088 processor. You do not have to specify this option explicitly because CL uses the 8086/8088 instruction set by default. Programs compiled with /G0 also run on machines with 80186/80188, 80286, 80386, and 80486 processors but do not take advantage of any processor-specific instructions. When you specify the /G0 option, the compiler automatically defines the identifier **M_I8086**.

If your program includes inline assembler code that uses a mnemonic instruction supported only by the 80186/87, 80286/87, 80386/87, or 80486 processors, you must compile with the /G1, /G2, /G3, or the /G4 option, respectively; compiling with /G0 results in an error. Note that you cannot use 80186, 80286, 80386, or the 80486 mnemonics as labels, even if you are compiling for an 8086/8088.

These options apply to all filenames that follow on the command line until another /G0, /G1, /G2, /G3, or /G4 option appears.

Note The /G0, /G1, and /G2 options work only with the 16-bit compiler; the /G3 and /G4 options work only with the 32-bit compiler.

Command-line drivers that produce 16-bit code (CL3216) generate warnings if they encounter either the /G3 or the /G4 option.

/GA, /GD (Optimize Entry/Exit Code for Protected-Mode Windows)

For protected-mode Windows applications, use the /GA option to optimize the entry/exit code of all far functions explicitly marked as `__export`.

For Windows dynamic-link libraries (DLLs) designed for protected mode, use the /GD option to optimize the entry/exit code of all far functions explicitly marked as `__export`.

When used instead of either the /GW or the /Gw option, both the /GA option and the /GD option save 10 bytes and 7 instructions for each function call. For /GA, use of `__export` adds an additional 6 bytes and 4 instructions to a function call; /GD with `__export` adds an additional 6 bytes and 7 instructions to a function call. The code generated in all four cases is smaller than that generated by /Gw or /GW. Note that in all four cases, you can also use the Generate 80286 Code option (/G2) to save an additional 4 bytes.

Both the /GA and /GD options define the `_WINDOWS` constant, and the /GD option defines the `_WINDLL` constant. The /GA option specifies use of both `mLIBC/W.LIB` and the Windows API library. The /GD option specifies use of both `mDLLC/W.LIB` and the Windows API library. You cannot specify the /Gw, /GW, or /Gq options with either the /GA or the /GD options.

/GE (Customize Windows Entry/Exit Code)

Option /GE[[*string*]]

The /GE option gives you control over the entry/exit code that the compiler produces for windows functions. It can only be used with the /GA and the /GD options. The *string* argument, which cannot be separated from /GE by a space, is one or more letters, with no intervening spaces, from the following table:

Letter	Optimizing Procedure
/GEr	Generates real-mode entry/exit code: /Gw code for functions marked as <code>__export</code> and /GW code for all other far functions. The /GEr option cannot be combined with any other /GE option.
/GEm	Specify generation of code to mark the far frame.
/GEf	Treat all far functions as if they were explicitly marked as <code>__export</code> .
/GEa, /GEd, /GEs	Load DS from AX (a), DGROUP (d), or SS (s). Specify only one letter.
/GEe	Force emission of linker EXPDEF records.

/Gc, /Gd (Use FORTRAN/Pascal or C Calling Convention)

The `__fortran`, `__pascal`, and `__cdecl` keywords and the /Gc and /Gd options allow you to control the function-calling and naming conventions so that your C programs can call and be called by functions that are written in FORTRAN or Pascal.

Because functions in C programs can take a variable number of arguments, C must handle function calls differently from languages such as Pascal and FORTRAN. Pascal and FORTRAN usually push actual parameters to a function in left-to-right order so that the last argument in the list is the last one pushed onto the stack. In contrast, because C functions do not always know the number of actual parameters, they must push their arguments from right to left, so that the first argument in the list is the last one pushed.

In C programs, the calling function must remove the arguments from the stack. In Pascal and FORTRAN programs, the called function must remove the arguments. If the code for removing arguments is in the called function (as in Pascal and FORTRAN), it appears only once; if it is in the calling function (as in C), it appears every time there is a function call. Because a typical program has more function calls than functions, the Pascal/FORTRAN method results in slightly smaller, more efficient programs. The compiler can generate the Pascal/FORTRAN calling convention in several ways.

Using `__pascal` and `__fortran` Keywords

You can use the `__pascal` and `__fortran` keywords with functions or pointers to functions to specify a function that uses the Pascal/FORTRAN calling convention. In the following example, `sort` is declared as a function using the alternative calling convention:

```
short pascal sort(char *, char *);
```

The `__pascal` and `__fortran` keywords can be used interchangeably. Use them when you want to use the left-to-right calling sequence for selected functions only.

/Gc (Use the Pascal/FORTRAN Calling Convention)

If you use the `/Gc` option, the entire module is compiled using the Pascal/FORTRAN calling convention. You might use this method to make it possible to call all the functions in a C module from another language or to gain the performance and size improvement provided by this calling convention.

When you use `/Gc` to compile a module, the compiler assumes that all functions called from that module use the Pascal/FORTRAN calling convention, even if the functions are defined outside that module. Therefore, using `/Gc` would usually mean that you cannot call or define functions that take variable numbers of parameters and that you cannot call functions such as the C library functions that use the C calling sequence. In addition, if you compile with the `/Gc` option, either you must declare the **main** function in the source program with the `__cdecl` keyword, or you must change the startup routine so that it uses the correct naming and calling conventions when calling **main**.

/Gd (Use the C Calling Convention)

The `/Gd` option has the same effect as the `__cdecl` keyword. It specifies that the entire module should use the C calling convention. This option is on by default.

The `__cdecl` keyword in C is the inverse of the `__fortran` and `__pascal` keywords. When applied to a function or function pointer, `__cdecl` indicates that the associated function is to be called using the usual C calling convention. This allows you to write C programs that take advantage of the more efficient Pascal/FORTRAN calling convention while still having access to the entire C library, other C objects, and even user-defined functions that accept variable-length argument lists. The `__cdecl` keyword takes precedence over the `/Gc` option.

For convenience, the `__cdecl` keyword has already been applied to run-time-library function declarations in the include files distributed with the compiler. Therefore, your C programs can call the library functions freely, no matter which calling conventions you compile with. Make sure to use the appropriate include file for each library function the program calls.

Naming Conventions

Use of the `__pascal` and `__fortran` keywords or the `/Gc` option also affects the naming convention for the associated item (or, in the case of `/Gc`, all items): the name is converted to uppercase letters, and the leading underscore that C usually prefixes is not added. The `__pascal` and `__fortran` keywords can be applied to data items and pointers, and also to functions; when applied to data items or

pointers, these keywords force the naming convention described previously for that item or pointer.

The **__fastcall** naming convention uses the function name preceded by an at sign (@). No case translation is done. When using this convention, make sure to use the standard include files. Otherwise, you get unresolved external references.

The **__pascal**, **__fortran**, **__fastcall**, and **__cdecl** keywords, like the **__based**, **__near**, **__far**, and **__huge** keywords, are disabled by use of the `/Za` option. If this option is given, these names are treated as ordinary identifiers, rather than keywords.

Examples

```
int __cdecl var_print(char*,...);
```

In this example, `var_print` is declared with a variable number of arguments using the usual right-to-left C function-calling convention and naming conventions. The **__cdecl** keyword overrides the left-to-right calling sequence set by the `/Gc` option if the option is used to compile the source file in which this declaration appears. If this file is compiled without the `/Gc` option, **__cdecl** has no effect since it produces the same result as the default C convention.

```
float __pascal root(number, root);
```

This example declares `root` to be a function returning a pointer to a value of type **float**. The function `root` uses the default calling sequence (left-to-right) and naming conventions for Microsoft FORTRAN and Pascal programs.

/Ge, /Gs (Turn Stack Checking On or Off)

The `/Ge` option, which applies to all source files that follow it on the command line, enables “stack probes.” A “stack probe” is a short routine called on entry to a function to verify that the program stack has enough room to allocate local variables required by the function. The stack-probe routine is called at every function-entry point. Ordinarily, the stack-probe routine generates a stack-overflow message if the required stack space is not available. When stack checking is turned off, the stack-probe routine is not called, and stack overflow can occur without being diagnosed (that is, no stack-overflow message is printed).

The compiler uses stack probes to guard against possible execution errors. These stack probes are used whenever the `/Ge` option (the default) is in effect. You can remove the stack probes by using either the `/Gs` option or the **check_stack** pragma, which reduces the size of a program and speeds up execution slightly. Note that the `/Gs` option and the **check_stack** pragma have no effect on standard C library routines; they affect only the functions you compile.

Use the `/Gs` option when you want to turn off stack checking for an entire module and you know the program does not exceed the available stack space. For example, stack probes may not be needed for programs that make very few function calls or that have only modest local-variable requirements. In the absence of the `/Gs` option, stack checking is on. Use the `/Gs` option with great care. Although it can make programs smaller and faster, it can also make the program unable to detect overflow of the program stack.

Use the `check_stack` pragma when you want to turn stack checking on or off only for selected routines, leaving the default (as determined by the presence or absence of the `/Gs` option) for the rest. When you want to turn off stack checking, put the following line before the definition of the function you don't want to check:

```
#pragma check_stack (off)
```

Note that the preceding line disables stack checking for all routines that follow it in the source file, not just the routines on the same line. To reinstate stack checking, insert the following line:

```
#pragma check_stack (on)
```

If you don't give an argument for the `check_stack` pragma, stack checking reverts to the behavior specified on the command line: disabled if the `/Gs` option is given or enabled if it is not. The interaction of the `check_stack` pragma with the `/Gs` option is summarized in Table 13.6.

Table 13.6 Using the `check_stack` Pragma

Syntax	Compiled with <code>/Gs</code> Option?	Action
<code>#pragma check_stack()</code>	Yes	Turns off stack checking for routines that follow
<code>#pragma check_stack()</code>	No	Turns on stack checking for routines that follow
<code>#pragma check_stack(on)</code>	Yes or no	Turns on stack checking for routines that follow
<code>#pragma check_stack(off)</code>	Yes or no	Turns off stack checking for routines that follow

Note For older versions of Microsoft C, the `check_stack` pragma had a different format: `check_stack(+)` enabled stack checking and `check_stack(-)` disabled stack checking. The Microsoft C/C++ compiler no longer accepts this format.

Example

```
CL /Gs FILE.C
```

This example optimizes the file FILE.C by removing stack probes with the /Gs option. If you want stack checking for only a few functions in FILE.C, you can use the `check_stack` pragma before and after the definitions of functions you want to check.

/Gr (Register Calling Convention)

Usually, your program passes parameters to functions on the stack. The /Gr option causes your program to pass parameters in registers instead. Typically, this calling convention decreases execution time, but it gives no advantage if you compile with the Fast Compile (/f) option. Therefore, use the /Gr option only for final compilations that require the full optimizing capabilities of Microsoft C/C++.

Passing parameters in registers is not appropriate for all functions. The /Gr option enables register passing for all eligible functions, and the `__fastcall` keyword enables it on a function-by-function basis. You cannot use the `__fastcall` keyword with the `__pascal`, `__fortran`, or `__cdecl` keywords.

Because the 80x86 processor has a limited number of registers, only the first three parameters are passed in registers; the remaining parameters are passed using the FORTRAN/Pascal calling convention (see the /Gc option).

Note that the compiler allocates different registers for variables declared as **register** and for passing arguments using the register calling convention. Passing arguments in registers does not conflict with any register variables that you may have declared.

Important Be careful when using the register calling convention for any function written in inline assembly language. Your use of registers in assembly language could conflict with the compiler's use of registers for storing parameters.

/Gn (Remove P-Code Native Entry Points)

The /Gn option lets you remove the native-code entry point from the beginning of a p-code function that does not require it, saving about four bytes for each function. This option must be used in conjunction with the /Oq option.

Native-code entry points are a short series of machine code instructions placed at the beginning of a function compiled into p-code. They are generated by the compiler in programs that mix p-code and machine code.

You can only use the /Gn option to remove the native-code entry point from functions called by other functions that you plan to compile into p-code.

You can control the removal of native-code entry points from within your source file using the **native_caller** pragma. This pragma takes “on” or “off” as an argument.

For example, to turn off native-code entry point generation for a p-code function, enter the following line prior to the beginning of the function:

```
#pragma native_caller (off)
```

Then turn the **native_caller** pragma back on after the end of the function by entering:

```
#pragma native_caller (on)
```

/Gp (Specifying Entry Tables)

Option */Gpnumber*

Use the */Gp* option to specify the maximum number of entry tables for your program. A space between */Gp* and *number* is optional. Like the other options for fine-tuning p-code described in Chapter 3 of the *Programming Techniques* manual, the */Gp* option must be used in conjunction with the */Oq* option.

An entry table is needed for every segment that contains a p-code function or a function called by a p-code function. One entry table can describe up to 256 such functions. If a segment contains more than that, the Make P-Code utility (MPC) creates additional entry tables.

Note The MPC utility is invoked automatically when you specify the */Oq* option on the CL command line.

Specify */Gpnumber* when you compile your source file. When the MPC utility processes the resulting .EXE file, it creates up to *number* entry tables. MPC returns an error if the program needs more than *number* entry tables.

If you do not specify the */Gp* option, *number* is assigned the default value of 255. In addition to the space that the actual entry tables take up, there is a four-byte overhead for each possible entry table.

/Gq (Real-Mode Windows Entry/Exit Code)

The */Gq* option is provided to maintain functional compatibility with earlier versions of the */GW* option. It generates entry/exit code for real-mode Windows functions not explicitly marked **__export**. The */Gq* option affects an entire

compilation module; use the `/Gw` option (described later in this chapter) instead of the `/Gq` option if a module contains functions marked as `__export`.

The entry/exit code that the `/Gq` option generates does not save the calling function's data segment (DS). Use the new `/GW` option (which does save DS) if DS might change during the time a function is on the call stack. The `/GW` option is described later in this chapter.

/Gt (Set Data Threshold)

Option `/Gt[[number]]`

The `/Gt` option causes all data items other than constant data whose size is greater than or equal to *number* bytes or that are assumed to be far (data items that are not initialized or are marked as **extern** are assumed to be far) to be allocated in a new data segment. For related information, see the `/Gx` option on page 523.

If you specify *number*, it must follow the `/Gt` option immediately with no intervening spaces. If you use `/Gt` without a number, the default threshold value is 256. If you don't use the `/Gt` option, the default threshold value is 32,767.

Use this option with programs that have more than 64K of initialized static and global data in small data items.

By default, the compiler allocates all static and global data items within the default data segment in the tiny, small, and medium memory models. In compact-, large-, and huge-model programs, only initialized static and global data items are assigned to the default data segment.

Note You can use the `/Gt` option only if you are creating a compact-, large-, or huge-model program because tiny-, small-, and medium-model programs have only one data segment.

/Gw, /GW (Generate Entry/Exit Code for Real-Mode Windows Functions)

Use the `/Gw` option when compiling real-mode Windows modules containing far functions marked as `__export`. The `/Gw` option instructs the compiler to generate entry/exit code sequences for real-mode Windows call-back functions.

Use the `/GW` option when compiling real-mode Windows modules containing only functions not marked as `__export`. The `/GW` option is similar to the `/Gw` option, but generates a more efficient entry sequence for real-mode windows functions that are not callback functions.

See the Microsoft Windows Software Development Kit for more information. Both options define the `_WINDOWS` constant, declared in the Windows version of `STDIO.H`.

Note The `/GW` option has been improved for Microsoft C/C++. Use the `/Gq` option, described earlier in this chapter, if you need the entry/exit code generated by previous versions of `/GW`.

The `/GA`, `/GD`, and `/GE` options, described earlier in this chapter, perform entry/exit code that will run only under standard- or enhanced-mode Windows. Use of these options can save up to 10 bytes and 7 instructions for each function call.

`/Gx (Assume That Data Is Near)`

Under the compact, large, or huge memory model, the compiler allocates initialized data items as near if they are smaller than or equal in size to the threshold value set by the `/Gt` option. The `/Gx` option extends this initialized data allocation rule to data that is uninitialized and data that is marked as **extern**.

Without the `/Gx` option, the compiler makes no assumptions about where the linker places uninitialized or external data. All references to those data items are done with far addressing, in case they are placed in a far segment.

Near data offers two benefits:

- The compiler can generate more efficient code to reference data it knows is near.
- You can achieve multiple instances of a single Windows application when all data is near.

The `/Gx` option works only if the memory-model specification for each individual data declaration (and its definition) is consistent across compilation modules. That is, an individual data item is either **__near** everywhere, **__far** everywhere, or its memory model is not specified anywhere.

To ensure that all data is near, mark unsized arrays as **__near**, make sure that no data is marked as **__far**, and that the data-size threshold set with the `/Gt` option does not force anything far. For more information on the data-size threshold, see the `/Gt` option earlier in this chapter.

The `/Gx` option does not affect pointers. Pointers remain far by default, and the dynamic allocation functions still return far pointers.

Use the `/Gx` option with either the `/AC`, the `/AL`, or the `/AH` option to modify the compact, large, or huge memory model, respectively. With `/Gx`, all three memory

models still offer multiple code and data segments. For more information on memory models, see Chapter 4 of the *Programming Techniques* manual. For more information on referencing declarations, see Chapter 3 in the *C Language Reference*.

Examples

```
CL /AL /Gx ONE.C TWO.C
```

This example compiles and links two modules, ONE.C and TWO.C, using the large memory model. Assume that ONE.C contains an external declaration of data (for example, **extern struct strr;**), and TWO.C defines **struct strr** as **__near**. In this case, specifying /Gx allows the compiler to safely generate more efficient code than would be possible if the compiler had to assume that **struct strr** might be far.

```
CL /AL /Gx /Gt8 ONE.C TWO.C
```

This example compiles and links two modules, ONE.C and TWO.C, using the large memory model. Because the /Gt option sets a data threshold size of 8 bytes, the compiler assumes that all data items smaller than 8 bytes and either uninitialized or marked as **extern** are near.

Note If you reference a data item with near addressing but declare it with **__far** in another module, your program will produce unpredictable results.

/Gy (Enable Function-Level Linking)

The /Gy option enables linking on a function-by-function basis by creating packaged functions. A packaged function is visible to the linker in the form of a COMDAT record. Packaged functions have several uses:

- You can exclude unreferenced packaged functions from the executable file by specifying the linker's /PACKF option. For more information on /PACKF, see page 589.
- You can place packaged functions in a specified order in the executable file by using a **FUNCTIONS** statement in a module-definition (.DEF) file. For more information on **FUNCTIONS**, see page 625.
- You can assign individual packaged functions to a specified segment by using a **FUNCTIONS** statement.
- You can place individual packaged functions in a specified overlay in a DOS program by using a **FUNCTIONS** statement. For more information on overlaid programs, see Chapter 15.

For C++, member functions are automatically packaged; other functions are not, and /Gy is required to compile them as packaged functions.

/H (Restricts Length of External Names)

Option */Hnumber*

The */H* option restricts the length of external (public) names. The *number* field accepts an integer specifying the maximum number of significant characters; the compiler considers only the first *number* characters of external names in the program. The program can contain external names longer than *number* characters, but the extra characters are ignored.

Without the */H* option, the default for C names is 32 characters; this includes any compiler-generated leading underscore (*_*) or at sign (*@*). (The compiler adds a leading underscore to names modified by *__cdecl* (default) calling conventions and a leading at sign to names modified by *__fastcall* calling conventions.) You can use the */H* option to restrict C names to values less than the 32-character default, or you can specify the acceptance of C names up to a limit of 247.

The */H* option is ignored for C++ names. Names from C++ programs have no added leading characters. C++; other decorated names have 247 significant characters.

You may find */H* useful when creating mixed-language or portable programs.

/HELP (List the Compiler Options)

Option */HELP*

/help

Calls the QuickHelp utility. If the QuickHelp program is not available, CL displays the most commonly used options to the standard output.

Unlike all other CL options, */HELP* is not case sensitive. Any combination of uppercase and lowercase letters is acceptable. For example, */hELp* is a valid form of this option. The option has no abbreviation.

/I (Search Directory for Include Files)

Option */Idirectory*

You can add to the list of directories searched for include files by using the */I* (for “include”) option. The space between */I* and *directory* is optional.

The search process for included files can involve three stages and begins with the **#include** directive. If the file specified to the **#include** directive contains a complete drive and path specification, that file is included without searching any directories. If the specified file is enclosed in double quotation marks, the directory of the file containing the **#include** directive is searched. If the current file is also an include file, the directory of the parent file is searched until the original source file's directory is searched.

The second stage uses the **/I** option. If the file is still not found or if it is specified to the **#include** directive in angle brackets, a directory specified by an **/I** command-line option is searched. To search more than one directory, give additional **/I** options on the CL command line—one **/I** option for each directory. Use a space to separate multiple **/I** options. Multiple directories are searched in order of their appearance on the command line. The directories are searched only until the specified include file is found.

The third and final stage involves the **INCLUDE** environment variable. If the file is not found in a directory specified by an **/I** option, a directory or path specified in the **INCLUDE** environment variable is used. This enables you to give a particular file special handling without changing the compiler environment you usually use. If the include file is not found, the compiler prints an error message and stops processing. When this occurs, you must restart compilation with a corrected directory specification.

Examples

```
CL /I \INCLUDE /I\MY\INCLUDE MAIN.C
```

In this example, CL looks for the include files requested by **MAIN.C** in the following order: first in the directory **\INCLUDE**, then in the directory **\MY\INCLUDE**, and finally in the directory or directories assigned to the **INCLUDE** environment variable.

```
CL /X /I \ALT\INCLUDE MAIN.C
```

In this example, the compiler looks for include files only in the directory **\ALT\INCLUDE**. The **/X** option, described later in this chapter, tells CL to consider the list of standard places empty; then the **/I** option specifies one directory to be searched.

/J (Change Default char Type)

The **/J** option changes the default **char** type from **signed** to **unsigned**. This option is useful when working with character data that will eventually be translated into a language other than English. If a **char** value is explicitly declared **signed**, the **/J** option does not affect it, and the value is sign-extended when widened to **int** type. The **char** type is zero-extended when widened to **int** type.

When you specify `/J`, the compiler automatically defines the identifier `_CHAR_UNSIGNED`, which is used with `#ifndef` in the `LIMITS.H` include file to define the range of the default `char` type.

Neither ANSI C nor C++ requires a specific implementation of the `char` type.

`/Ld, /Lw (Control Library Selection)`

Use the `/Ld` option to link with both `mDLLCfW.LIB` and with the Windows API library. This option is implied by the `/GD` option; you cannot use it with either the `/GA` or the `/Mq` option.

`/link (Linker-Control Options)`

Option `/link [[option]]`

This option passes to LINK the linker options, nondefault library names, and library search paths specified by the *option* argument. These options must appear after any source or object filenames and CL options. A space is required between `\link` and *option*.

The following CL options also affect the linker.

Option	Effect
<code>/F</code> <i>hexnum</i>	Resets stack size to specified hexadecimal number. This option is the same as passing the <code>STACK</code> option to LINK.
<code>/Fm</code>	Produces a map file.
<code>/FPa</code>	Specifies the alternate math library and places the name of the library for the current memory model in the object file.
<code>/FPc</code>	Generates floating-point calls to the emulator library and places the name of the library for the current memory model in the object file.
<code>/FPc87</code>	Generates function calls to the 80x87 library and places the name of the library for the current memory model in the object file.
<code>/FPi</code>	Uses the math emulator library.
<code>/FPi87</code>	Uses the coprocessor and places the name of the 80x87 library for the current memory model in the object file. You can specify an emulator library at link time with an object file that was linked with <code>/FPi87</code> so that the resulting program runs in a system without a coprocessor.
<code>/Ld</code>	Links with <code>mDLLCfW.LIB</code> and the Windows API library.
<code>/Lr</code>	Appends "r" (for real mode) to the default library name in the generated object files.
<code>/Lw</code>	Links with <code>mLIBCfW.LIB</code> and the Windows API library.

/Ln (Link Without C Run-Time Startup Code)

If you are using the tiny memory model (see the /AT option for CL), you will be creating a .COM file (see the /TINY option for LINK). Usually, CL tells LINK to link tiny-model programs with CRTCOM.LIB; this file contains startup code needed by any .COM program written in C. Programs written in assembly language do not need this code. Use the /Ln option to keep LINK from linking with this startup code.

/Lr (Real Mode Default Library)

The /Lr option appends “r” (for real mode) to the default library name in the generated object files.

/MA (Macro Assembler Options)

Option */MA[[option]]*

This option passes the specified *option* to the Microsoft Macro Assembler (MASM). The *option* must follow immediately after /MA, without an intervening space, slash (/), or hyphen (-).

For example, /MAZi passes the Zi option to MASM.

Files listed on the command line with the extension .ASM automatically cause MASM to be invoked. All MASM-supported options are accepted.

/Mq (QuickWin Support)

DOS programs compiled with the /Mq compiler option have a limited Windows user interface, including a standard menu bar, standard Help (for the QuickWin features), and a client (or application) window with a child (document) window for the C input/output streams, **stdin**, **stdout**, and **stderr**. The /Mq compiler option defines the **_WINDOWS** constant, declared in the Windows version of STDIO.H. For more information, see Chapter 8 in the *Programming Techniques* manual.

/ND, /NM /NQ, /NT, /NV (Name the Data or Code Segments)

Options */NDdatasegment*
 /NMmodulename
 /NQpcodesegment

/NTcodesegment

/NVvtablesegment

These options allow you to name or rename existing data and text segments, or to name a temporary p-code segment or a segment for far C++ virtual tables (v-tables). All options take an argument that names or renames the affected segment. The new name can include any combination of letters and digits. The space between the option and the name is optional.

The compiler places code and data into separate segments in the object file. Every segment in every object file has a name. The linker uses these names to determine which segments are combined during linking, and how the segments are ultimately grouped in the executable file.

The compiler usually creates the code and data segment names. The default names depend on the memory model chosen for the program. For example, in small-model programs the code segment is named **_TEXT** and the data segment is named **_DATA**.

Table 13.7 summarizes the naming conventions for code and data segments.

Table 13.7 Segment-Naming Conventions

Model	Code	Data
Tiny	_TEXT	_DATA
Small	_TEXT	_DATA
Medium	<i>name</i> _TEXT	_DATA
Compact	_TEXT	_DATA
Large	<i>name</i> _TEXT	_DATA
Huge	<i>name</i> _TEXT	_DATA

The */ND* option renames the default data segment of your code. This option is useful mainly for shared data segments. When compiled with */ND*, the program assumes that the data register (DS) contains the address of this new segment so that it can access the segment's contents using near pointers instead of far. In doing so, your program no longer assumes that the address in the stack segment register (SS) is the same as the address in the data register (DS). You must therefore use the **__loadds** modifier for function declarations or the */Au* segment setup option to ensure that DS is loaded on entry to a function.

Note that the C run-time system stores important data in its default data segment (**DGROUP**). To use the run-time system, DS must contain the address of this data segment. A call to the run-time system will fail if DS currently points to the data segment named with */ND*. A safer and a more flexible way to place data into a

special segment is to use the `__based` keyword with a segment variable in your C code instead of the `/ND` option on the command line.

The `/NM` and `/NT` options are similar; they rename the default code segment. The `/NM` option sets the name of a module used in naming the `_TEXT` segment of medium-, large-, or huge-model programs. It appends `_TEXT` to the specified module name to create a new code segment, `modulename_TEXT`. The `/NM` option is included in Microsoft C/C++ for compatibility with previous versions.

The `/NT` option gives the code segment the specified name. In general, you should not use the `/NT` option with the tiny, small and compact memory models. Doing so may cause overflow errors at link time.

The `/NQ` option sets the name of a temporary segment for the p-code compiler; the temporary segment is removed before the program is run. This option can only be used with the `/Oq` (p-code optimization) option. During its operation, the p-code compiler generates several temporary segments. If you encounter LINK error 1049 (“too many segments”), use `/NQ` to combine these temporary segments into one temporary segment.

The `/NV` option sets the name of a segment for far v-tables. All far v-tables in a C++ program are grouped in the specified segment.

/nologo (Suppress Display of Sign-On Banner)

The `/nologo` option suppresses the display of the sign-on banner when `CL` is invoked.

/O Options (Optimize Program)

Option

/O string

The `/O` options give you control over the optimizing procedures that the compiler performs. The *string* argument is one or more letters, with no intervening spaces, from the following table:

Letter	Optimizing Procedure
<code>/Oa</code>	Assume no aliasing
<code>/Ow</code>	Assume aliasing across function calls
<code>/Obn</code>	Control inline expansion, where <i>n</i> is a digit from 0 through 2
<code>/Oc</code>	Enable block-level common subexpression optimization (default)
<code>/Og</code>	Enable global-level common subexpression optimization
<code>/Od</code>	Turn off all optimization

Letter	Optimizing Procedure
/Oe	Ignore register keyword and allow compiler to perform global register allocation
/Of	Turn on p-code quoting (default)
/Of-	Turn off p-code quoting
/Oi	Generate intrinsic functions
/Ol	Enable loop optimization
/On	Turn off potentially unsafe loop optimizations
/Oz	Turn on potentially unsafe loop optimizations
/Oo	Turn on post code-generation optimizing (default)
/Oo-	Turn off post code-generation optimizing
/Op	Improve float consistency
/Oq	Turn on p-code optimization
/Or	Enable single exit point from functions (useful when debugging with CodeView)
/Os	Minimize executable file size
/O, /Ot	Minimize execution speed (default)
/Ov	Sort local variables by frequency of use, p-code only (default)
/Ov-	Sort local variables in the order that they occur, p-code only
/Ox	Maximize optimization

With the exception of the minus switches, /Of-, /Oo-, and /Ov-, the letters following the /O, including *bn*, can appear in a continuous series in any order. When there is a conflict, the compiler uses the last /O option given. Each option applies to all source files following it on the command line.

Note If you are using the debugger, you may want to set the /Od option. If you don't specify /Od, some forms of the compiler perform code-movement optimizations, possibly making it difficult for you to follow your program in the debugger.

/Oa and /Ow (Assuming No Aliasing)

An "alias" is a name used to refer to a memory location already referred to by a different name. Because a memory access requires more time than is required to access the CPU's registers, the compiler tries to store frequently used variables in registers. Aliasing, however, reduces the extent to which a compiler can keep variables in registers.

The /Oa option tells the compiler to ignore the possibility of multiple aliases for a memory location. In the list that follows, the term "reference" means read or write; that is, whether a variable is on the left side of an assignment statement or the right side, you are still referring to it. In addition, any function calls that use a variable as a parameter are references to that variable. When you tell the compiler to

assume that you are not doing aliasing, it expects that the following rules are being followed for any variable not declared as **volatile**:

- If a variable is used directly, no pointers are used to reference that variable.
- If a pointer is used to refer to a variable, that variable is not referred to directly.
- If a pointer is used to modify a memory location, no other pointers are used to access the same memory location.

Both the `/Oa` and `/Ow` options tell the compiler that you have not used aliases in your code. When you use `/Oa`, you specify that you will not be doing any aliasing (which allows the compiler to perform significant optimizations that might not otherwise have been possible), and that function calls are safe. When you use the `/Ow` option, you specify that aliasing might occur across function calls. Therefore, after each function call, pointer variables must be reloaded from memory. For more information on the `/Oa` and `/Ow` options and aliasing, see the *Programming Techniques* manual.

Aliasing bugs most frequently show up as corruption of data. If you find that global or local variables are being assigned seemingly random values, take the following steps to determine if you have a problem with optimization and aliasing:

- Compile the program with `/Od` (disable optimizations).
- If the program works when compiled with the `/Od` option, check your normal compile options for the `/Oa` option (assume no aliasing).
- If you were using the `/Oa` option, fix your compile options so that `/Oa` is not specified.

Note You can instruct the compiler to disable unsafe optimizations with code that does aliasing by using the **optimize** pragma with the **a** or **w** option.

`/Obn` (Control Inline Expansion)

Use the `/Ob` option to control the inline expansion of functions, where *n* is a digit from 0 though 2. The following table describes the action of each digit.

Table 13.8 Inline Expansion Control

Digit	Action
0	Disables inline expansion (default with <code>/Od</code>)
1	Only expand functions marked as inline or __inline or in a C++ member function defined within a class declaration (default without <code>/Od</code>)
2	Expand functions marked as inline or __inline and any other function that the compiler chooses (expansion occurs at compiler discretion)

/Oc and /Og (Enable Common Subexpression Optimization)

Use of either the `/Oc` or the `/Og` option allows the compiler to calculate the value of a common subexpression once. For example, `b + c` is common to the following code. If the values of `b` and `c` do not change between the three expressions, the compiler can calculate the value of `b + c` once, assign the calculation to a temporary variable, and substitute the variable as appropriate.

```
a = b + c;  
d = b + c;  
e = b + c;
```

When you use the `/Oc` option (default common subexpression optimization), the compiler examines short sections of code (basic blocks) for common subexpressions. When you use the `/Og` option (enable global common subexpression optimization), the compiler searches entire functions for common subexpressions. You can disable the default common subexpression optimization with the `/Od` option. For more information about common subexpression optimization, see Chapter 1 in the *Programming Techniques* manual.

Note You can enable or disable block-scope common subexpression optimization on a function-by-function basis using the **optimize** pragma with the **c** option. You can enable or disable global common subexpression optimization on a function-by-function basis using the **optimize** pragma with the **g** option.

/Od (Turn Off Optimization)

The `/Od` option tells the compiler to turn off all optimizations in the program, which speeds compilation. Use the `/Od` option when you compile with the `/Zi` option (described on page 553) to include debugging information. The `/Od` option does not reorganize code, making it easier to debug.

/Oe (Global Register Allocation)

The `/Oe` option instructs the compiler to ignore the **register** keyword and to allocate registers based on how often they are used. This allows the compiler to store frequently used variables and subexpressions in registers.

/Of (Turn On and Turn Off P-Code Quoting)

The `/Of` option (p-code only) enables the compiler to find duplicate sections of code and then create a single instance of that code. This process is known as “quoting.” With quoting enabled (`/Of`), the duplicate code is replaced with the p-code equivalent of a function call to produce smaller code than is produced when quoting is disabled (`/Of-`).

Quoting enabled (/Of) is the default. However, quoting makes code difficult to read; use the /Of- option to disable quoting while debugging. Quoting is especially useful for a final product.

/Oi (Generate Intrinsic Functions)

The /Oi option instructs the compiler to replace the following function calls with their inline forms:

Function	Target		Function	Target	
	16 bit	32 bit		16 bit	32 bit
_alloca	no	yes	_outpw	yes	yes
_disable	yes	yes	_rotl	yes	yes
_enable	yes	yes	_rotr	yes	yes
_fmemcmp	yes	no	_setjmp	no	yes
_fmemcpy	yes	no	_strset	yes	yes
_fmemset	yes	no	abs	yes	yes
_fstreat	yes	no	fabs	yes	yes
_fstrcmp	yes	no	labs	yes	yes
_fstrcpy	yes	no	memcmp	yes	yes
_fstrlen	yes	no	memcpy	yes	yes
_fstrset	yes	no	memset	yes	yes
_inp	yes	yes	strcat	yes	yes
_inpw	yes	yes	strcmp	yes	yes
_lrotl	yes	yes	strcpy	yes	yes
_lrotr	yes	yes	strlen	yes	yes
_outp	yes	yes			

Programs that use intrinsic functions are faster because they do not include the overhead associated with function calls. However, they may be larger due to the additional code generated.

Intrinsic versions of the **memset**, **memcpy**, and **memcmp** functions in compact- and large-model programs cannot handle huge arrays or huge pointers. To use huge arrays or huge pointers with these functions, you must compile your program with either the huge memory model from PWB or with the /AH option on the command line.

With /Oi, you cannot link to an alternate math library. Also, the following floating-point functions do not have true intrinsic forms; they do have versions that pass arguments directly to the floating-point chip instead of pushing them onto the usual normal argument stack:

Function	Target		Function	Target	
	16 bit	32 bit		16 bit	32 bit
acos	yes	yes	_acosl	yes	no
asin	yes	yes	_asinxl	yes	no
atan	yes	yes	_atanl	yes	no
atan2	yes	yes	_atan2l	yes	no
ceil	yes	no	_ceil	yes	no
cos	yes	yes	_cosl	yes	no
cosh	yes	yes	_coshl	yes	no
exp	yes	yes	_expl	yes	no
floor	yes	no	_floorl	yes	no
fmod	yes	yes	_fmodl	yes	no
log	yes	yes	_logl	yes	no
log10	yes	yes	_log10l	yes	no
pow	yes	yes	_powl	yes	no
sin	yes	yes	_sinl	yes	no
sinh	yes	yes	_sinhl	yes	no
sqrt	yes	yes	_sqrtl	yes	no
tan	yes	yes	_tanl	yes	no
tanh	yes	yes	_tanhl	yes	no

For more information on intrinsic functions, see Chapter 1 in the *Programming Techniques* manual.

/OI (Optimize Loops)

The /OI option enables a set of loop optimizations that move or rewrite code so that it executes more quickly. Because loops involve sections of code that are executed repeatedly, they are targets for optimization.

The /OI option removes invariant code. An optimal loop contains only expressions whose values change through each execution of the loop. Any subexpression whose value is constant should be evaluated before the body of the loop is executed. Unfortunately, these subexpressions are not always readily apparent. The optimizer can remove many of these expressions from the body of a loop at compile time. This example illustrates invariant code in a loop:

```
i = -100;
while( i < 0 )
{
    i += x + y;
}
```

In the preceding example, the expression `x + y` does not change in the loop body. Loop optimization removes this subexpression from the body of the loop so that it is only executed once, not every time the loop body is executed. The optimizer changes the code to the following:

```
i = -100;
t = x + y;
while( i < 0 )
{
    i += t;
}
```

Loop optimization is much more effective when the compiler can assume no aliasing. While you can use loop optimization without the `/Oa` or `/Ow` option, use `/Oa` to ensure that the most options possible are used.

Here is a code fragment that could have an aliasing problem:

```
i = -100;
while( i < 0 )
{
    i += x + y;
    *p = i;
}
```

If you do not specify the `/Oa` option for the preceding code, the compiler must assume that either `x` or `y` could be modified by the assignment to `*p`. Therefore, the compiler cannot assume the subexpression `x + y` is constant for each loop iteration. If you specify that you are not doing any aliasing (with the `/Oa` option), the compiler assumes that modifying `*p` cannot affect either `x` or `y`, and that the subexpression is indeed constant and can be removed from the loop, as in the previous example.

Note All loop optimizations specified by the `/Ol` option are safe optimizations. To enable aggressive loop optimizations, you must use the enable aggressive optimizations (`/Oz`) option. While the optimizations enabled by the combination of `/Ol` and `/Oz` are not safe for all cases, they do work properly for most programs.

`/Oo`, `/Oo-` (Enable and Disable Post-Code-Generation Optimizing)

The `/Oo` option allows the compiler to perform the following optimizations after code generation has occurred:

- Peepholes
- Code motion
- Dead code removal
- Unused locals removal

- Register reloading removal
- Exit sequence
- Branch shortening

The `/Oo-` option disables the optimizations. The following table shows the default state of the `/Oo` option and its state with the p-code option (`/Oq`) and the Fast Compile option (`/f`):

Compiler	Default
Optimizing	On
P-Code	On
Fast Compile	Off

`/On` (Turn Off Potentially Unsafe Loop Optimizations)

The `/On` option disables unsafe loop optimizations by preventing some constant expressions in a loop from being moved out of the loop. Since the `/Ol` option automatically implies `/On`, you do not need to specify `/On` explicitly; it is provided for compatibility with earlier versions of Microsoft C.

`/Oz` (Turn On Maximum Loop Optimization)

The `/Oz` option aggressively removes as many invariant expressions as possible from a loop. If you do not specify `/Oz`, the `/Ol` option removes only the invariant expressions guaranteed to be executed in every iteration of the loop. Do not use `/Oz` if you suspect that a loop optimization has caused an error.

`/Op` (Improve Float Consistency)

The primary use of the `/Op` option is to improve the consistency of tests for equality and inequality. Without the `/Op` option, the compiler usually uses coprocessor registers to hold the intermediate results of floating-point calculations. Such optimization increases program speed and decreases program size.

The coprocessor has 80-bit registers; the memory representation can be 32, 64, or 80 bits. Therefore, storing intermediate results in registers can provide a greater degree of precision than storing them in memory.

When you use the `/Op` option, the compiler loads data from memory prior to each floating-point operation and, if assignment occurs, writes the results back to memory upon completion. Loading the data prior to each operation guarantees that the data does not retain any significance greater than the capacity of its type.

A program compiled with `/Op` may be slower and larger than one compiled without `/Op`.

`/Oq` (Turns On P-Code Optimization)

The `/Oq` option optimizes your code for size by compiling the program into an alternate form called “p-code.” P-code produces much smaller programs than machine code, but your machine cannot execute them directly. Programs compiled into p-code are executed by a small run-time interpreter incorporated into your executable file. P-code does not support initialized static huge data.

You can also directly control aspects of p-code optimization. You can control which sections of the program get compiled into p-code using the `optimize` pragma and the `/Oq` option together.

For example,

```
#pragma optimize ("q", on)
// Functions compiled into p-code
#pragma optimize ("q", off)
// Remaining functions compiled into machine code
```

These pragmas are ignored if the program is not compiled with the `/Oq` option.

Since p-code runs more slowly than machine language, you may want to have the speed-critical sections of your program compiled into machine language.

Note The `/Oq` option is not compatible with any of the following options: `/FPa`, `/FPc`, `/FPc87`, `/Fs`, `/Gr`, `/Sl`, `/Sp`, `/Ss`, or `/St`.

`/Or` (Common Function-Exit Sequence)

This option causes all returns within a function to use a common exit sequence at the end of the function. This allows you to set a CodeView breakpoint on the closing brace `}` of the function rather than on every return statement within the function. This can be used, for example, to quickly get to the end of a function you wish to bypass.

Note The `/Or` option is useful only when using CodeView; do not use it when generating code for a finished product.

`/Os` (Minimize Executable File Size)

The `/Os` option minimizes the size of executable files; it produces smaller but possibly slower code. If you do not select this option, code is larger but may be faster.

/O and /Ot (Minimize Execution Time)

When you do not use any of the /O options, the CL command automatically optimizes for program execution speed. The /O and /Ot options have the same effect as this default.

Wherever the compiler has a choice between producing smaller (but perhaps slower) and larger (but perhaps faster) code, the compiler generates faster code. For example, when you specify the /Ot option, the compiler generates either library calls to the operating system or inline code to perform shift operations on long operands.

/Ov, /Ov- (Specify Type of Frame Sorting)

The compiler reduces the size of p-code programs by using 1-byte opcodes to reference local variables. These opcodes are frame-relative addresses, and only a limited number of them are available for each function. The optimizer uses one of two algorithms to determine which variables receive the available opcodes.

Option	Description
---------------	--------------------

/Ov	Sorts the local variables by frequency of use (default)
/Ov-	Sorts the local variables in the order that they occur (lexical order)

/Ox (Use Maximum Optimization)

The /Ox option is a shorthand way to combine optimizing options to produce the fastest possible program. Its effect is the same as using the following options on the same command line:

```
/Ob1cegilnot /Gs
```

Example

```
CL /Ox FILE.C
```

This command tells the compiler to use the optimizing compiler to compile FILE.C with the following options:

Option	Description
---------------	--------------------

/Ob1	Expand functions marked as inline or __inline or those in a C++ member function defined within a class declaration
/Oc	Enable local common subexpression elimination
/Oe	Allow global register allocation
/Og	Enable global common subexpression elimination
/Oi	Generate intrinsic functions

Option	Description
/Ol	Perform loop optimizations
/On	Turn off potentially unsafe loop optimizations
/Oo	Enable post code-generation optimizing
/Ot	Favor execution time over code size
/Gs	Remove stack probes

/P (Create Preprocessor-Output File)

The `/P` option writes preprocessor output to a file with the same base name as the source file but with the `.I` extension. This option adds **#line** directives to the output file. They are placed at the beginning and end of each included file and around lines removed by preprocessor directives that specify conditional compilation.

The preprocessed listing file is identical to the original source file except that all preprocessor directives are carried out and macro expansions are performed. You usually use the `/P` option with the `/C` option (discussed on page 491), which preserves comments in the preprocessed output.

The `/P` option suppresses compilation; `CL` does not produce an object file or listing, even if you specify the `/Fo` or `/Fm` option on the `CL` command line.

The `/P` option is similar to both the `/E` and the `/EP` options, which are described earlier in this chapter. Use of `/EP` with `/P` suppresses placement of **#line** directives in the output file.

Example

```
CL /P MAIN.C
```

This example creates the preprocessed file `MAIN.I` from the source file `MAIN.C`.

/qc (Quick Compile)

The Quick Compile (`/qc`) option is supported for compatibility with Microsoft C version 6.0. The Fast Compile (`/f`) option supersedes the `/qc` option in Microsoft C/C++. `CL` defines the `_QC` constant for the `/qc` option.

/Sl, /Sp, /Ss, /St (Source-Listing Format Options)

Option */Fs /Slnumber*

/Fs /Spnumber

/Fs /Ssstring

/Fs /Ststring

When combined with the */Fs* option, the */Sl*, */Sp*, */Ss*, and */St* options set the following source-listing attributes:

Option	Attribute
<i>/St</i>	Title; enclose argument in quotes if spaces or tabs are present.
<i>/Ss</i>	Subtitle; enclose argument in quotes if spaces or tabs are present.
<i>/Sl</i>	Line width; argument must be an integer between 79 and 132; default is 79.
<i>/Sp</i>	Page length; argument must be an integer between 15 and 255; default is 63.

An option applies to the arguments that follow it on the command line or until the next occurrence of the option. A space between */St*, */Ss*, */Sl*, or */Sp* and *number* is optional. A space is required between */Fs* and any of these four options.

If the source file compiles with no errors more serious than warning errors, the source listing includes tables of local symbols, global symbols, and parameter symbols for each function.

None of the */S* options are supported by the Fast Compile option (*/f*) or by the p-code option (*/Oq*).

/Tc, /Tp, Ta (Specify C, C++ Source File, or Assembly Language)

Options */Tcfilename*

/Tpfilename

/Tafilename

The `/Tc` option specifies that *filename* is a C source file, even if it doesn't have the extension `.C`. The `/Tp` option specifies that *filename* is a C++ source file, even if it doesn't have the extension `.CPP` or `.CXX`. The `/Ta` option specifies that *filename* is an assembly language file, even if it doesn't have the extension `.ASM`. You must have installed the Microsoft Macro Assembler in order to use the `/Ta` option. This option causes CL to invoke the Macro Assembler to assemble the file. For all three options, the space between the option and the *filename* argument is optional.

If this option does not appear, CL assumes that files with the `.C` extension are C source files, files with the `.CPP` or the `.CXX` extension are C++ source files, and files with the `.ASM` extension are assembly language files.

If you need to specify more than one source file with an extension other than the default (`.C`, `.CPP`, `.ASM`) you must specify each file with the appropriate command line option (`/Tc`, `/Tp` `/Ta`).

Example

In the following example, the CL command compiles the three source files `MAIN.C`, `TEST.PRG`, and `COLLATE.PRG`.

```
CL MAIN.C /Tc TEST.PRG /Tc COLLATE.PRG PRINT.PRG
```

Because the file `PRINT.PRG` is given without a `/Tc` option, CL treats it as an object file. Therefore, after compiling the three source files, CL links the object files `MAIN.OBJ`, `TEST.OBJ`, `COLLATE.OBJ`, and `PRINT.PRG`.

/U, /u (Remove Predefined Names)

Options

`/Uname`

`/u`

The `/U` (for “undefine”) option turns off the definition of the specified defined name. The `/u` option turns off every defined name. The `/U` and the `/u` options apply both to predefined names and to names that you define.

These names are useful in writing portable programs. For instance, they can be used with compiler directives to conditionally compile parts of a program, depending on the processor and operating system being used. The predefined identifiers and their meanings are listed in Table 13.9.

One or more spaces can separate `/U` and *name*. You can specify more than one `/U` option on the same command line.

Table 13.9 Predefined Names

Syntax	Purpose	When Defined
<code>_CHAR_UNSIGNED</code>	Specifies that the char type is unsigned by default.	When <code>/J</code> is given
<code>_DLL</code>	Specifies a DLL run-time library.	When <code>/MD</code> is given
<code>_FAST</code>	Specifies Fast Compile.	When <code>/f</code> is given
<code>_M_I86, _M_I86</code>	Specifies target machine as a member of the Intel family.	Always
<code>_M_I86mM, _M_I86mM</code>	Specifies memory model, where <i>m</i> is either T (tiny model), S (small model), C (compact model), M (medium model), L (large model), or H (huge model). If huge model is used, both <code>_M_I86LM</code> and <code>_M_I86HM</code> are defined.	Always
<code>_M_I8086, _M_I8086</code>	Specifies target machine as an 8086.	When <code>/G0</code> is given and by default
<code>_M_I286, _M_I286</code>	Specifies target machine as an 80286.	When <code>/G1</code> or <code>/G2</code> is given
<code>_M_I386, _M_I386</code>	Specifies target machine as an 80386.	When <code>/G3</code> is given
<code>_MSC_VER</code>	Specifies version of Microsoft C currently supported. Equal to 700.	Always
<code>_MSDOS, _MSDOS</code>	Specifies target operating system as MS-DOS.	Always
<code>_QC</code>	Specifies Quick Compile	When <code>/qc</code> is selected
<code>_PCODE</code>	Specifies p-code.	When <code>/Oq</code> is selected
<code>__STDC__</code>	Specifies full conformance with the ANSI C standard.	When <code>/Za</code> is selected
<code>_WINDLL</code>	Specifies protected-mode dynamic-linked library	When <code>/GD</code> is selected
<code>_WINDOWS</code>	Specifies protected-mode Windows	When <code>/GA, /GE, /Gn, /GW, /Mq,</code> and <code>/GD</code> are selected

Note If a predefined identifier has two forms, one with and one without an underscore, the command-line driver defines both if you specify the `/Ze` option (compile for Microsoft extensions). It defines only the leading underscore form if you specify the `/Za` option (compile for ANSI compatibility).

You can define 30 identifiers.

Example

```
CL /UMSDOS /UM_I86 WORK.C
```

This example removes the definitions of two predefined names. Note that the `/U` option must be given twice to do this.

`/V` (Set Version String)

Option `/Vstring`

The `/V` option embeds a text string in the object file. This string can label an object file with a version number or a copyright notice. If the specified string contains white-space characters, it must be enclosed in double quotation marks (" "). A backslash must precede any embedded double quotation marks. A space between `/V` and *string* is optional.

`/W, /w` (Set Warning Level)

Options `/W{0|1|2|3|4|X}`

`/w`

You can control the number of warning messages produced by the compiler by using the `/w`, `/W0`, `/W1`, `/W2`, `/W3`, `/W4`, or `/WX` option. A space between `/W` and 0, 1, 2, 3, 4, or X is optional. Compiler warning messages are any messages beginning with C4; see the *Comprehensive Index and Errors Reference* for a complete list of these messages.

Warnings indicate potential problems (rather than actual coding errors) with statements that may not compile as you intend.

The `/W` options affect only source files named on the command line; they do not apply to object files.

The following table describes the warning-level options. W1 warnings are the most serious and W4 warnings are the least serious:

Option	Action
<code>/w</code>	Turns off all warning messages. Use this option when you compile programs that deliberately include questionable statements. The <code>/w</code> option applies to the remainder of the command line or until the next occurrence of a <code>/w</code> option on the command line. <code>/W0</code> and <code>/W</code> are the same as <code>/w</code> .
<code>/W1</code>	Default. Displays severe warning messages.

Option	Action
/W2	Displays an intermediate level of warning messages. Level 2 includes warnings such as the following: <ul style="list-style-type: none">▪ Use of functions with no declared return type.▪ Failure to put return statements in functions with nonvoid return types.▪ Data conversions that would cause loss of data or precision.
/W3	Displays a less severe level of warning messages, including warnings about function calls that precede their function prototypes in the source code.
/W4	Displays the least severe level of warning messages, including warnings about the use of non-ANSI features and extended keywords.
/WX	Treats all warnings as errors. If there are any warning messages, an error message is emitted and compilation continues.

Note The descriptions of the warning messages in the *Comprehensive Index and Errors Reference* indicate the warning level that must be set (that is, the number for the appropriate /W option) for the message to appear.

Example

```
CL /W4 CRUNCH.C PRINT.C
```

This example enables all possible warning messages when the source files CRUNCH.C and PRINT.C are compiled. Microsoft C/C++ provides a pragma to control warning messages. For more information on **#pragma warning**, see Chapter 7, “Preprocessor Directives and Pragas,” in the *C Language Reference*.

/X (Ignore Standard Include Directory)

You can prevent the compiler from searching the standard places for include files by using the /X (for “exclude”) option. When CL sees the /X option, it does not search the current directory or any directories specified in the INCLUDE environment variable.

You can use this option with the /I option to define the location of include files that have the same names as include files found in other directories but that contain different definitions. See the /I option, described earlier in this chapter, for an example of /X used with /I.

/Fp (Specify Precompiled Header Filename)

Option */FPfilename*

Option (/Fp)

Use the /Fp option to specify the name of the desired .PCH file in cases where the .PCH filename specified with the /Yc option is different from the filename of the associated include file or source file. You can use the /Fp option with both the /Yc and the /Yu options. For example, if you want to create a precompiled header file for a debugging version of your program, you can specify a command such as:

```
CL /DDEBUG /YcPROG.H /FpDPROG.PCH PROG.CPP
```

This command creates a precompilation of all header files up to and including PROG.H, and stores it in a file called DPROG.PCH. If you need a release version in parallel, you simply change the compilation command to:

```
CL /YcPROG.H /FpRPROG.PCH PROG.CPP
```

This command creates a separate precompilation of the header files up to and including PROG.H and stores it in RPROG.PCH.

You can use the /Fp option with the /Yu option to specify the name of the .PCH file if the name is different from either the *filename* argument to /Yc or the base name of the source file:

```
CL /YuPROG.H /FpZPROG.PCH PROG.CPP
```

This command specifies a precompiled header file named ZPROG.PCH. The compiler uses the contents of ZPROG.H to restore the precompiled state of all header files up to and including PROG.H. The compiler then compiles the code that occurs after the PROG.H include statement.

For more information on precompiled headers, see Chapter 2 in the *Programming Techniques* manual. The related options /Yc, /Yd, and /Yu are described in the sections that follow.

/Yc, /Yd, /Yu (Precompiled Header Options)

With the /Yc, /Yd, /Yu, and /Fp options and the **hdrstop** pragma, Microsoft C/C++ provides the ability to precompile code. Precompilation, especially when used the fast compile (/f) option, can dramatically reduce compile time for code that is frequently compiled without modification.

You can precompile a stable body of code, which can be header files and all or part of source files—including inline code. Precompilation works with both C and C++.

This process saves the state of a compilation (including CodeView information) in a precompiled header file with a .PCH extension. In later compilations, the compiler simply restores the saved compilation state from a precompiled header file.

Creation or use of precompiled headers is governed by the compilation options described in the next three sections. In many cases, the behavior dictated by these options is affected by the **hdrstop** pragma. For more information on both precompiled headers and the **hdrstop** pragma, see “Using Precompiled Headers” in the *Programming Techniques* manual. The related option to specify precompiled header filenames (/Fp) is described earlier in this chapter.

/Yc (Create Precompiled Header)

Option

`/Yc[[filename]]`

The “create precompiled header” option (/Yc) instructs the compiler to create a precompiled header (.PCH) file that represents the state of compilation at a certain point. No space is allowed between /Yc and *filename*.

Using /Yc with a Filename

If you specify a filename with the /Yc option, it instructs the compiler to create a precompiled header in which to save the state of the compilation up to and including the preprocessing of the named include file. Consider the following code:

```
#include <afxwin.h> // Include header for class library
#include "resource.h" // Include resource definitions
#include "myapp.h" // Include information specific to this app.
...
```

When compiled with the command

```
CL /YcMYAPP.H PROG.CPP
```

the compiler saves all the preprocessing for AFXWIN.H, RESOURCE.H, and MYAPP.H in a precompiled header file called MYAPP.PCH.

Using /Yc Without a Filename

If you specify the /Yc option with no filename, the entire source file—including every included header file—is compiled. The state of the compilation is saved to a file with the base name of the source file and a .PCH extension.

Because precompilation is most useful for compiling a stable body of code for use with a body of code that is under development, you will want to focus the precompilation process by using `/Yc` with its *filename* argument or by putting a **hdrstop** pragma in your source file.

The **hdrstop** Pragma

Option

```
#pragma hdrstop[["filename"]]
```

If a C or a C++ file contains a **hdrstop** pragma, the compiler saves the state of the compilation up to the location of the pragma. The compiled state of any code that follows the pragma is not saved.

The **hdrstop** pragma cannot occur inside a header file. It must occur in the source file at the file level; that is, it cannot occur within any data or function declaration or definition.

The **hdrstop** pragma is ignored unless either `/Yu` or `/Yc` is specified without a *filename*.

Using **hdrstop** with a Filename

Use *filename* to name the precompiled header file in which the compiled state is saved. The *filename* must be a string (enclose it in quotation marks), and it must be enclosed in parentheses. A space between **hdrstop** and *filename* is optional. Note that you can also use the `/Fp` option to name the precompiled header file. If *filename* is not specified, the resulting filename is given the base name of the source file with a `.PCH` extension. The `/Fp` option is discussed earlier in this chapter.

Using **hdrstop** Without a Filename

If no *filename* is given with the **hdrstop** pragma, the name of the precompiled header file is derived from the base name of the source file with a `.PCH` extension.

`/Yd` (Include Debugging Information)

The “include debugging information” (`/Yd`) option instructs the compiler to place a precompiled header file’s debugging information (symbols and type information) into the precompiled header instead of the object file. The option takes no argument and has effect only if both the create precompiled header (`/Yc`) and the generate CodeView information (`/Zi`) options are in effect.

By default, any debugging information for a precompiled header is saved in the object file for which the precompiled header is created rather than in the precompiled

header. In subsequent inclusions of this precompiled header, the compiler does not insert symbol information in the object file; rather, it inserts cross-references to the original object file. Therefore, no matter how many times the header is included in compilations, the debugging information exists only in one object file.

Although this default behavior results in faster compilation and reduces disk-space demands, it can be undesirable if you are distributing a debugging library. The `/Yd` option overrides this behavior so that complete debugging information is included in each object file.

`/Yu` (Use Precompiled Header)

Option

`/Yu[[filename]]`

The “use precompiled header” option (`/Yu`) instructs the compiler to restore its state from a previous compilation using a precompiled header file. No space is allowed between the option and the *filename*.

Using `/Yu` with a Filename

If *filename* is specified, it must correspond to one of the header files included in the source file using the **#include** preprocessor directive. The compiler skips to the specified **#include** directive, restores the compiled state from the precompiled header file, and then compiles only code that follows *filename*.

Unless the `/Fp` option is used, the compilation state is restored from a file that has the same base name as the include file and a `.PCH` extension. Consider the following code:

```
#include <afxwin.h> // Include header for class library
#include "resource.h" // Include resource definitions
#include "myapp.h" // Include information specific to this app.
...
```

When compiled with the command line

```
CL /YuMYAPP.H PROG.CPP
```

the compiler does not process the three **#include** statements but restores its state from the precompiled header `MYAPP.PCH`, thereby saving the time involved in preprocessing all three of the files (and any files they might include).

You can use the `/Fp` option with the `/Yu` option to specify the name of the `.PCH` file if the name is different from either the *filename* argument to `/Yc` or the base name of the source file.

```
CL /YuMYAPP.H /FpMYPCH.PCH PROG.CPP
```

This command specifies a precompiled header file named MYPCH.PCH. The compiler uses its contents to restore the precompiled state of all header files up to and including MYAPP.H. The compiler then compiles the code that occurs after the MYAPP.H include statement.

Using /Yu Without a Filename

When you specify the /Yu option without a filename, your source program must contain a **hdrstop** pragma that specifies the filename. The compiler skips to the location of that pragma, restores the compiled state from the precompiled header file specified by the pragma, and then compiles only code that follows the pragma. If the **hdrstop** pragma does not specify a filename, the compiler looks for a file with a name derived from the base name of the source file, with the .PCH extension.

If you specify the /Yu option without a filename and fail to specify a **hdrstop** pragma, an error message is generated and the compilation is unsuccessful.

/Ze, /Za (Enable or Disable Language Extensions)

Microsoft C/C++ supports the ANSI C standard. In addition, it offers a number of features beyond those specified in the ANSI C standard. These features are enabled when the /Ze (default) option is in effect and disabled when the /Za option is in effect. They include the following:

- The **__based**, **__cdecl**, **__far**, **__fastcall**, **__fortran**, **__huge**, **__near**, **__pascal**, **__stdcall**, **__syscall**, and **__interrupt** keywords.
- Use of casts to produce l-values:

```
int *p;  
(( long * ) p )++;
```

The preceding example could be rewritten to conform with the ANSI C standard as follows:

```
p = ( int * )(( long * )p + 1 );
```

- Redefinitions of **extern** items as **static**:

```
extern int foo();  
static int foo()  
{}
```

- Use of trailing commas (,) rather than an ellipsis (...) in function declarations to indicate variable-length argument lists:

```
int printf( char *, );
```

- Use of benign **typedef** redefinitions within the same scope:

```
typedef int INT;
typedef int INT;
```

- Use of mixed character and string constants in an initializer:

```
char arr[5] = {'a', 'b', "cde"};
```

- Use of bit fields with base types other than **unsigned int** or **signed int**.
- Use of single-line comments, which are introduced with two slash characters:

```
// This is a single-line comment.
```

- Casting of a function pointer to a data pointer:

```
int ( * pfunc ) ();
int *pdata;

pdata = ( int * ) pfunc;
```

To perform the same cast while maintaining ANSI compatibility, you must cast the function pointer to an **int** before casting it to a data pointer:

```
pdata = ( int * ) (int) pfunc;
```

- Function declarators have file scope:

```
void func1()
{
    extern int func2( double );
}

void main( void )
{
    func2( 4 );    // /Ze passes 4 as type double
}                // /Za passes 4 as type int
```

- Use of declarators without either a storage class or a type:

```
x;

void main( void )
{
    x = 1;
}
```

- Use of zero-sized arrays as last field in structures and union:

```
struct zero
{
    char *c;
    int zarray[];
};
```

- Use of block scope variables initialized with nonconstant expressions:

```
int foo( int );
int bar( int );

void main( void )
{
    int array[2] = { foo( 2 ), bar( 4 ) };
}

int foo( int x )
{
    return x;
}

int bar( int x )
{
    return x;
}
```

- Previous function declarator specifies a variable number of arguments, but the function definition provides a type instead:

```
void myfunc( int x, ... );

void myfunc( int x, char * c )
{ }
```

Use the `/Za` option if you plan to port your program to other environments. The `/Za` option tells the compiler to treat extended keywords as simple identifiers and to disable the other extensions listed above.

When you specify `/Za`, the compiler automatically defines the `__STDC__` identifier. In the include files provided with the C run-time libraries, this identifier is used with `#ifndef` to control use of the `__cdecl` keyword on library function prototypes. For an example of this conditional compilation, see the file `STDIO.H`.

`/Zc` (Specify Pascal Naming)

This option ignores case at the source level for any name declared with the `__pascal` keyword.

`/Zg` (Generate Function Prototypes)

The `/Zg` option generates a function prototype for each function defined in the source file but does not compile the source file.

The function prototype includes the function return type and an argument-type list. The argument-type list is created from the types of the formal parameters of the function. Any function prototypes already present in the source file are ignored.

The generated list of prototypes is written to the standard output. You may find this list helpful to verify that actual arguments and formal parameters of a function are compatible. You can save the list by redirecting standard output to a file. Then you can use **#include** to make it a part of your source file as function prototypes. Doing so causes the compiler to perform argument type checking.

If you use the `/Zg` option and your program contains formal parameters that have structure, enumeration, or union type (or pointers to such types), the prototype for each structure, enumeration, or union type must have a tag.

`/Zi, /Zd (Compile for Debugging)`

The `/Zi` option produces an object file containing line numbers and full symbolic-debugging information for use with the CodeView window-oriented debugger. This symbolic information is a map of your source code that the debugger uses. It includes such things as variable names and their types, function names and their return types, and the number and names of all of the program's segments. The object file also includes full symbol-table information and line numbers.

The `/Zd` option produces an object file containing only global and external symbol information and line number information. Use this option when you want to reduce the size of an executable file that you will be debugging with the CodeView debugger. You can also use `/Zd` when you do not need to use the expression evaluator during debugging.

Example

```
CL /c /Zi TEST.C
```

This command produces an object file named TEST.OBJ that contains line numbers corresponding to the lines in TEST.C.

`/ZI (Remove Default-Library Name from Object File)`

Ordinarily, CL puts the name of the default library (`mLIBCE.LIB`) in the object file so the linker can automatically find the correct library to link with the object file.

The `/ZI` option tells the compiler not to place the default-library name in the object file. As a result, the object file is slightly smaller. The option affects all files that follow it on the command line.

Use the `/Zl` option when you are using the LIB utility (described in Chapter 19) to build a library. You can use `/Zl` to compile the object files you plan to put in your library, thereby omitting the default-library names from your object modules. Although the `/Zl` option saves only a small amount of space for a single object file, the total amount of space saved is significant in a library containing many object modules.

Example

```
CL ONE.C /Zl TWO.C
```

This example creates the following two object files:

- An object file named ONE.OBJ that contains the name of the C library SLIBCE.LIB
- An object file named TWO.OBJ that contains no default-library information

When ONE.OBJ and TWO.OBJ are linked, the default-library information in ONE.OBJ causes the default library to be searched for any unresolved references in either ONE.OBJ or TWO.OBJ.

`/Zp` (Pack Structure Members)

Option `/Zp[{{1|2|4}}]`

When storage is allocated for structures, structure members are ordinarily stored as follows:

- Items of type **char** or **unsigned char**, or arrays containing items of these types, are byte aligned.
- Structures are word aligned; structures of odd size are padded to an even number of bytes.
- All other types of structure members are word aligned.

To conserve space or to conform to existing data structures, you may want to store structures more or less compactly. The `/Zp` option and the **pack** pragma control how structure data are packed into memory.

Use the `/Zp` option to specify the same packing for all structures in a module. When you give the `/Zpn` option, where *n* is 1, 2, or 4, each structure member after the first is stored on *n*-byte boundaries depending on the option you choose. If you use the `/Zp` option without an argument, structure members are packed on one-byte boundaries. No space is allowed between `/Zp` and its argument.

On some processors, the `/Zp` option may result in slower program execution because of the time required to unpack structure members when they are accessed. For example, on an 8086 processor, this option can reduce efficiency if members with `int` or `long` type are packed in such a way that they begin on odd-byte boundaries.

Use the `pack` pragma in your source code to pack particular structures on boundaries different from the packing specified on the command line. Give the `pack(n)` pragma, where *n* is 1, 2, or 4, before structures that you want to pack differently. To reinstate the packing given on the command line, give the `pack()` pragma with no arguments.

Table 13.10 shows the interaction of the `/Zp` option with the `pack` pragma.

Table 13.10 Using the pack Pragma

Syntax	Compiled with <code>/Zp</code> Option?	Action
<code>#pragma pack()</code>	Yes	Reverts to packing specified on the command line for structures that follow
<code>#pragma pack()</code>	No	Reverts to default packing for structures that follow
<code>#pragma pack(<i>n</i>)</code>	Yes or no	Packs the following structures to the given byte boundary until changed or disabled

Example

```
CL /Zp PROG.C
```

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

`/Zf` (Accept `__far` Keyword)

When you use the `/Zf` option, the compiler ignores every instance of the `__far` keyword in your source code. This option is useful when porting code written for the segmented memory models of 16-bit computers to the flat memory model of 32-bit computers.

`/Zn` (Turn Off `SBRPACK` Utility)

Use the `/Zn` option, with either the `/FR` or the `/Fr` option, to turn off the `SBRPACK` utility. This option is only required when you have used the `/Iu` `BSCMAKE` option to keep unreferenced definitions in your code. The `SBRPACK` utility removes

unreferenced definitions from the .SBR file produced by the /FR and the /Fr options. As the BSCMAKE utility uses .SBR files to produce output for the PWB Source Browser, this smaller .SBR file gives BSCMAKE greater effective capacity and can also increase the speed of producing .BSC files. Also, smaller .SBR files save disk space.

Use of either the /FR or the /Fr option is part of the process you must follow in order to use the PWB Source Browser. Both options are described earlier in this chapter.

For information on using the PWB Source Browser to look for code in a project, to see where functions are invoked or where variables and types are used, or to generate call trees and cross-reference tables, see Chapter 21, “Browser Utilities.”

Example

```
CL /FR /Zn QSORT.C
```

This example compiles and links the source file QSORT.C and generates a file with an .SBR extension. The /Zn option ensures that the SBRPACK utility does not compact the .SBR file. The BSCMAKE utility uses the file to generate a .BSC file that you can examine with the PWB Source Browser.

/Zr (Check Pointers)

The /Zr option checks for null or out-of-range pointers in your program. A run-time error occurs if you try to run a program with such pointers. The /Zr option is only available with the fast compile (/f) option.

If you compile with the /Zr option, you can use the **check_pointer** pragma within your source file to turn checking on or off only for selected pointers, leaving the default (see below) for the remaining pointers in the program. When you want to turn on pointer checking, put the following line before the usage of the pointer you want to check:

```
#pragma check_pointer (on)
```

This line turns on pointer checking for all pointers that follow it in the source file, not just the pointers on the following line. To turn off pointer checking, insert the following line at the location you want pointer checking turned off:

```
#pragma check_pointer (off)
```

If you don't give an argument for the **check_pointer** pragma, pointer checking reverts to the behavior specified on the command line: turned on if the /Zr option is given or turned off otherwise.

Example

```
CL /Zr PROG.C
```

This command causes CL to check for null or out-of-range pointers in the file PROG.C. All pointers in the file are checked except those to which a **check_pointer(off)** pragma applies.

/Zs (Check Syntax Only)

The /Zs option tells the compiler to check only the syntax of the source files that follow the option on the command line. This option provides a quick way to find and correct syntax errors before you try to compile and link a source file.

When you give the /Zs option, the compiler does not generate code or produce object files, object listings, or executable files. The compiler, however, does display error messages if the source file has syntax errors.

Example

```
CL /Zs TEST*.C
```

This command causes the compiler to perform a syntax check on all source files in the current working directory that begin with TEST and end with the .C extension. The compiler displays messages for any errors found.

Specifying Options with the CL Environment Variable

Use the CL environment variable to specify files and options without giving them on the command line. The environment variable has the following format:

```
SET CL=[[ option ] ... [ file ] ...] [/link [ link-libinfo ] ]
```

The CL environment variable is useful if you often give a large number of files and options when you compile. Ordinarily, DOS limits the command line to 128 characters. The files and options that you define with the CL environment variable, however, do not count toward this limit. Therefore, you can define the files and options you use most often with the CL variable and then give only the files and options you need for specific purposes on the command line.

The information defined in the CL variable is treated as though it appeared before the information given on the CL command line.

Note that if you have given an option in the CL environment variable, you generally cannot turn off or change the option from the command line. You must reset

the CL environment variable and omit the file or option that you do not want to use.

With DOS, you cannot use CL to set options that use an equal sign (for example, the */Didentifier= string* option) With Microsoft C/C++, the define constants and macro option (*/D*) accept the number sign (#) as an equal sign (=). You can now use the CL environment variable to define preprocessor constants and macros (for example, */D identifier string*).

You cannot use wildcards in filenames to specify multiple files with CL.

Examples

In the following example, the CL environment variable tells the CL command to use the */Zp*, */Ox*, and */I* options during compilation and then to link with the object file `\LIB\BINMODE.OBJ`.

```
SET CL=/Zp2 /Ox /I\INCLUDE\MYINCLS \LIB\BINMODE.OBJ
CL INPUT.C
```

With CL defined as shown, the preceding CL command has the same effect as the command line:

```
CL /Zp2 /Ox /I\INCLUDE\MYINCLS \LIB\BINMODE.OBJ INPUT.C
```

That is, both specify structure packing on two-byte boundaries; perform maximum optimizations; search for include files in the `\INCLUDE\MYINCLS` directory; and suppress translation of carriage-return–linefeed character combinations for the source file `INPUT.C`.

In the following example, the CL environment variable instructs the CL command to compile and link the source files `FILE1.C` and `FILE2.C`.

```
SET CL=FILE1.C FILE2.C
CL FILE3.OBJ
```

The CL preceding command line has the same effect as the command line:

```
CL FILE1.C FILE2.C FILE3.OBJ
```

The following example illustrates how to turn off the effects of a CL option defined in the environment:

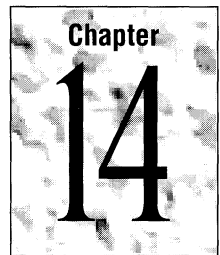
```
SET CL=/Za
CL FILE1.C /Ze FILE2.C
```

In this example, the CL environment variable is set to the `/Za` option, which tells the compiler not to recognize Microsoft extensions to the C language. This option causes Microsoft-specific keywords to be treated as ordinary identifiers rather than as reserved words. The CL command specifies the inverse option, `/Ze`, which tells the compiler to treat language extensions as reserved words. The effect is the same as compiling with the command line:

```
CL /Za FILE1.C /Ze FILE2.C
```

Therefore, `FILE1.C` is compiled with language extensions turned off, and `FILE2.C` is compiled with language extensions enabled.

Linking Object Files with LINK



This chapter describes the Microsoft Segmented Executable Linker (LINK) version 5.30. LINK combines compiled or assembled object files into an executable file. This chapter explains LINK's input syntax and fields and tells how to use options to control LINK.

LINK is distributed in the form of LINK.EXE for DOS. LINK is DOS-extended and uses extended memory if available.

When you link for debugging using the /CO option, LINK calls the CVPACK utility. CVPACK version 4.00 must be available on the path. For more information, see "CVPACK" on page 743.

This version of LINK does not support the Microsoft Incremental Linker (ILINK). To allow existing makefiles to remain compatible, a file called ILINK.EXE is provided. This version of ILINK always invokes a full link. The LINK options for incremental linking are no longer supported. If /INCR, /PADC, or /PADD is specified, LINK issues a warning and ignores the option.

14.1 New Features

This version of LINK has several new or changed features. This section summarizes changes in options. Module-definition statements are discussed in Chapter 16. An improved way to create overlaid DOS programs is described in Chapter 15.

The following options are new or changed in this version of LINK:

/DOSS[[EG]]

The minimum unique abbreviation for /DOSSEG option has changed from /DO to /DOSS. (See page 578.)

/DY[[NAMIC]][[:*number*]]

The new /DYNAMIC option lets you change the limit of interoverlay calls in an overlaid DOS program. (See page 579.)

/INC[[REMENTAL]]

The **/INCR** option is no longer supported.

/INF[[ORMATION]]

The **/INFO** option gives more detailed output. One new use is to get the number of interoverlay calls needed to specify with the **/DYNAMIC** option. (See page 582.)

/M[[AP]][:*maptype*]

The **/MAP** option has been enhanced. You can get more or less detail in the map file by specifying an optional qualifier. (See page 583.)

/NOPACKC[[ODE]]

The minimum unique abbreviation for **/NOPACKC** has changed from **/NOP** to **/NOPACKC** to distinguish it from the new **/NOPACKF** option. (See page 586.)

/NOPACKF[[UNCTIONS]]

The new **/NOPACKF** option keeps unreferenced packaged functions. (See page 586.)

/OL[[DOVERLAY]]

The new **/OLDOVERLAY** option links an overlaid DOS program using the Static Overlay Manager instead of the **MOVE** library. This option may not be supported in future versions of **LINK**. (See page 586.)

/ON[[ERROR]]:N[[OEXE]]

The **/ONERROR:NOEXE** option prevents **LINK** from creating the program output if an error occurs. (See page 586.)

/OV[[ERLAYINTERRUPT]]

The minimum unique abbreviation for this option has changed from **/O** to **/OV**, to distinguish it from the new **/OLDOVERLAY** option. (See page 587.)

/PACKF[[UNCTIONS]]

The new **/PACKF** option removes unreferenced packaged functions. (See page 589.)

/PADC[[ODE]]

The **/PADC** option is no longer supported.

/PADD[[ATA]]

The **/PADD** option is no longer supported.

/PM[[TYPE]]

The default for the **/PM** option has changed from **NOVIO** to **PM**. (See page 590.)

/r

The new **/r** option tells the linker not to use extended memory. (See page 591.)

14.2 Overview

LINK combines 80x86 object files into either an executable file or a dynamic-link library (DLL). The object-file format is the Microsoft Relocatable Object-Module Format (OMF), which is based on the Intel 8086 OMF. LINK uses library files in Microsoft library format.

LINK creates “relocatable” executable files and DLLs. The operating system can load and execute relocatable files in any unused section of memory. LINK can create DOS executable files with up to 1 megabyte of code and data (or up to 16 megabytes when using overlays). It can create segmented executable files with up to 16 megabytes.

For more information on the OMF, the executable-file format, and the linking process, see the *MS-DOS Encyclopedia*.

Use EXEHDR to examine the finished file.

When the file (either executable or DLL) is created, you can examine the information that LINK puts in the file’s header by using the Microsoft EXE File Header Utility (EXEHDR). For more information on EXEHDR, see Chapter 17.

Other programs can call LINK automatically.

The Microsoft Programmer’s WorkBench (PWB) invokes LINK to create the final executable file or DLL. Therefore, if you develop your software with PWB, you might not need to read this chapter. However, the detailed explanations of LINK options might be helpful when you use the LINK Options dialog box in PWB. This information is also available in Help.

The compiler or assembler supplied with your language (CL with C, FL with FORTRAN, ML with MASM) also invokes LINK. You can use most of the LINK options described in this chapter with the compiler or assembler. The Microsoft Advisor has more information about the compilers and assembler; select Help for the appropriate language from the Command Line box of the Help Contents screen.

Note Unless otherwise noted, all references to “library” in this chapter refer to a static library. This can be either a standard library created by the Microsoft Library Manager (LIB) or an import library created by the Microsoft Import Library Manager (IMPLIB), but not a DLL.

14.3 LINK Output Files

LINK can create executable files for DOS or Windows. The kind of file produced is determined by the way the source code is compiled and the information supplied to LINK. LINK’s output is either an executable file or a DLL. For simplicity, this chapter sometimes refers to this output as the “program” or “main output.” LINK creates the appropriate file according to the following rules:

- If a .DEF file is specified, LINK creates a segmented executable file. The type is determined by the EXETYPE and LIBRARY statements.
- If a .DEF file is not specified, LINK creates a DOS program.
- If an overlay number is specified in a **SEGMENTS** or **FUNCTIONS** statement, LINK creates an overlaid DOS program. This overrides a conflicting .DEF file specification.
- If /DYNAMIC or /OLDOVERLAY is specified, or if parentheses are used in the *objects* field, LINK creates an overlaid DOS program. This overrides a conflicting .DEF file specification.
- If an object file or library module contains an export definition (an EXPDEF record), LINK creates a segmented executable file. This overrides an overlay specification. The **__export** keyword creates an EXPDEF record in a C program. Microsoft C libraries for protect mode contain EXPDEF records, so linking with a protect-mode default library creates a segmented executable file.
- If an import library is specified, LINK creates a segmented executable file.

Map files list the segments and symbols in a program.

LINK can also create a “map” file, which lists the segments in the executable file and can list additional information. The /LINE and /MAP options control the content of the map file.

LINK produces other files when certain options are used.

Other options tell LINK to create other kinds of output files. LINK produces a .COM file instead of an .EXE file when the /TINY option is specified. The combination of /CO and /TINY puts debugging information into a .DBG file. A Quick library results when the /Q option is specified. For more information on these and other options, see “LINK Options” on page 575.

14.4 LINK Syntax and Input

The LINK command has the following syntax:

```
LINK objfiles[[, exefile]] [[, mapfile]][[, libraries]][[, deffile]] ] ] ] ] ;
```

The LINK fields perform the following functions:

- The *objfiles* field is a list of the object files that are to be linked into an executable file or DLL. It is the only required field.
- The *exefile* field lets you change the name of the output file from its default.
- The *mapfile* field creates a map file or gives the map file a name other than its default name.
- The *libraries* field specifies additional (or replacement) libraries to search for unresolved references.
- The *deffile* field gives the name of a module-definition (.DEF) file.

Fields are separated by commas. You can specify all the fields, or you can leave one or more fields (including *objfiles*) blank; LINK then prompts you for the missing input. (For information on LINK prompts, see “Running LINK” on page 572.) To leave a field blank, enter only the field’s trailing comma.

Options can be specified in any field. For descriptions of each of LINK’s options, see “LINK Options” on page 575.

The fields must be entered in the order shown, whether they contain input or are left blank. A semicolon (;) at the end of the LINK command line terminates the command and suppresses prompting for any missing fields. LINK then assumes the default values for the missing fields.

If your file appears in or is to be created in another directory or device, you must supply the full path. Filenames are not case sensitive. If the filename contains a space (supported on some installable file systems), enclose the name in single or double quotation marks (' or ").

The next five sections explain how to use each of the LINK fields.

The objfiles Field

The *objfiles* field specifies one or more object files to be linked. At least one filename must be entered. If you do not supply an extension, LINK assumes a default .OBJ extension. If the filename has no extension, add a period (.) to the end of its name.

If you name more than one object file, separate the names with a plus sign (+) or a space. To extend *objfiles* to the following line, type a plus sign (+) as the last character on the current line, then press ENTER, and continue. Do not split a name across lines.

How LINK Searches for Object Files

When it searches for object files, LINK looks in the following locations in the order specified:

1. The directory specified for the file (if a path is included). If the file is not in that directory, the search ends.
2. The current directory.
3. Any directories specified in the LIB environment variable.

If LINK cannot find an object file, and a floppy drive is associated with that object file, LINK pauses and prompts you to insert a disk that contains the object file.

Load Libraries

If you specify a library in the *objfiles* field, it becomes a “load library.” LINK treats a load library like any other object file. It does not search for load libraries in directories named in the *libraries* field. You must specify the library’s filename extension; otherwise, LINK assumes an .OBJ extension.

LINK puts every object module from a load library into the executable file, regardless of whether a module resolves an external reference. The effect is the same as if you had specified all the library’s object-module names in the *objfiles* field.

Specifying a load library can create an executable file or DLL that is larger than it needs to be. (A library named in the *libraries* field adds only those modules required to resolve external references.) However, loading an entire library can be useful when:

- Repeatedly specifying the same group of object files.
- Placing a library in an overlay.
- Debugging so you can call library routines that would not be included in the release version of the program.

Overlays

A special syntax for the *objfiles* field lets you assign the contents of object files to specific overlays in a DOS program. To place one or more object files in an overlay, enclose the filenames in parentheses. This syntax may not be supported in future versions of LINK. For more information about overlays, see Chapter 15.

The exefile Field

The *exefile* field is used to specify a name for the main output file. If you do not supply an extension, LINK assumes a default extension, either .EXE, .COM (when using the /TINY option), .DLL (when using a module-definition file containing a **LIBRARY** statement), or .QLB (when using the /Q option).

If you do not specify an *exefile*, LINK assigns a default name to the main output. This name is the base name of the first file listed in the *objfiles* field, plus the extension appropriate for the type of executable file being created.

LINK creates the main file in the current directory unless you specify an explicit path with the filename.

The mapfile Field

The *mapfile* field is used to specify a filename for the map file or to suppress the creation of a map file. A map file lists the segments in the executable file or DLL.

You can specify a path with the filename. The default extension is .MAP. Specify NUL to suppress the creation of a map file. The default for the *mapfile* field is one of the following:

- If this field is left blank on the command line or in a response file, LINK creates a map file with the base name of the *exefile* (or the first object file if no *exefile* is specified) and the extension .MAP. If the field contains a dot (.), the map file is given the base name without an extension.
- When using LINK prompts, LINK assumes either the default described previously (if an empty *mapfile* field is specified) or NUL.MAP, which suppresses creation of a map file.

To add line numbers to the map file, use the /LINE option. To add public symbols and other information, use the /MAP option. Both /LINE and /MAP force a map file to be created unless NUL is explicitly specified in *mapfile*.

The libraries Field

You can specify one or more standard or import libraries (not DLLs) in the *libraries* field. If you name more than one library, separate the names with a plus sign (+) or a space. To extend *libraries* to the following line, type a plus sign (+) as the last character on the current line, press ENTER, and continue. Do not split a name across lines. If you specify the base name of a library without an extension, LINK assumes a default .LIB extension.

If no library is specified, LINK searches only the default libraries named in the object files to resolve unresolved references. If one or more libraries are specified, LINK searches them in the order named before searching the default libraries.

You can tell LINK to search additional directories for specified or default libraries by giving a drive name or path specification in the *libraries* field; end the specification with a backslash (\). (If you don't include the backslash, LINK assumes that the specification is for a library file instead of a directory.) LINK looks for files ending in .LIB in these directories.

You can specify a total of 32 paths or libraries in the field. If you give more than 32 paths or libraries, LINK ignores the additional specifications without warning you.

You might need to specify library names to:

- Use a default library that has been renamed.
- Specify a library other than the default named in the object file (for example, a library that handles floating-point arithmetic differently from the default library).
- Search additional libraries.
- Find a library that is not in the current directory and not in a directory specified by the LIB environment variable.

Default Libraries

Most compilers insert the names of the required language libraries in the object files. LINK searches for these default libraries automatically; you do not need to specify them in the *libraries* field. The libraries must already exist with the name specified in the object file. Default-library names usually refer to combined libraries built and named during setup; consult your compiler documentation for more information about default libraries.

To make LINK ignore the default libraries, use the /NOD option. This leaves unresolved references in the object files. Therefore, you must use the *libraries* field to specify the alternative libraries that LINK is to search.

Import Libraries

You can specify import libraries created by the IMPLIB utility anywhere you can specify standard libraries. You can also use the LIB utility to combine import libraries and standard libraries. These combined libraries can then be specified in the *libraries* field. For more information on LIB, see Chapter 19. For information on IMPLIB, see page 745.

How LINK Resolves References

LINK searches static libraries to resolve external references. A static library is either a standard library created by the LIB utility or an import library created by the IMPLIB utility.

LINK searches object files and libraries for a definition of each external reference. When LINK finds a needed definition in a module in a library, LINK adds the entire module (but not necessarily all modules in the library) to the program.

You provide a library to LINK in the following ways:

- Specify the name of a library in the *libraries* field.
- Specify the name of a library as a load library in the *objects* field. A load library adds all its modules to the program. For more information, see “Load Libraries” on page 566.
- Compile a program that uses definitions provided in a default library for that compiler. The compiler places a library comment record in the object file. LINK uses the library named in this record.
- Embed a library comment record in the object file by using the comment pragma in a C program. This record precedes a record for a default library placed in the object file by the compiler; therefore, LINK looks in this library before it searches a default library named in the same object file.

LINK first looks for a definition in files specified in the *objects* field, then it looks in libraries specified in the *libraries* field. The search order is the order in which the files are specified in the fields. LINK then looks in libraries specified in comment records in the object files, again in the specified order.

If LINK cannot find a needed definition, it issues an error message:

```
unresolved external
```

If a reference is defined in more than one library, LINK uses the first definition it finds as it searches the libraries in order. A duplicate definition may not be a problem if the later definition is in a module that is not linked into the program. However, if the duplicate definition is in a module that contains another needed definition, that module is linked into the program, and the duplicate definition causes an error:

```
symbol defined more than once
```

Multiple definitions can also cause a problem if LINK is using extended dictionaries in libraries. An extended dictionary is a summary of the definitions contained in all modules of a library. LINK uses this summary to speed the process of searching libraries. If LINK finds a previously resolved reference listed in an extended dictionary, it assumes that a duplicate definition exists and issues an error message:

```
symbol multiply defined, use /NOE
```

If this error occurs, link your program using the `/NOE` option.

How LINK Searches for Library Files

When searching for a library, LINK looks in the following locations in this order:

1. The directory specified for the file, if a path is included. (The default libraries named in object files by Microsoft compilers do not include path specifications.)
2. The current directory.
3. Any directories specified in the *libraries* field.
4. Any directories specified in the LIB environment variable.

If LINK cannot locate a library file, it prompts you to enter the location. The /BATCH option disables this prompting.

Example

The following is a specification in the *libraries* field:

```
C:\TESTLIB\ NEWLIBV3 C:\MYLIBS\SPECIAL
```

LINK searches NEWLIBV3.LIB first for unresolved references. Since no directory is specified for NEWLIBV3.LIB, LINK looks in the following locations in this order:

1. The current directory
2. The C:\TESTLIB\ directory
3. The directories in the LIB environment variable

If LINK still cannot find NEWLIBV3.LIB, it prompts you with the message:

```
Enter new file spec:
```

Enter either a path to the library or a path and filename for another library.

If unresolved references remain after LINK searches NEWLIBV3.LIB, it then searches the library C:\MYLIBS\SPECIAL.LIB. If LINK cannot find this library, it prompts you as described previously for NEWLIBV3.LIB. If there are still unresolved references, LINK searches the default libraries.

The deffile Field

Use the *deffile* field to specify a module-definition file. A module-definition file is required for an overlaid DOS program or a DLL. It is optional for a Windows application. If you specify a base name with no extension, LINK assumes a .DEF extension. If the filename has no extension, put a period (.) at the end of the name.

By default, LINK assumes that a *deffile* needs to be specified. If you are linking without a .DEF file, use a semicolon to terminate the command line before the *deffile* field (or accept the default NUL.DEF at the Definitions File prompt).

How LINK Searches for Module-Definition Files

LINK searches for the module-definition file in the following order:

1. The directory specified for the file (if a path is included). If the file is not in that directory, the search terminates.
2. The current directory.

For information on module-definition files, see Chapter 16.

Examples

The following examples illustrate various uses of the LINK command line.

Example 1

```
LINK FUN+TEXT+TABLE+CARE, , FUNLIST, FUNPROG.LIB;
```

This command line links the object files FUN.OBJ, TEXT.OBJ, TABLE.OBJ, and CARE.OBJ. By default, the executable file is named FUN.EXE because the base name of the first object file is FUN and no name is specified for the executable file. The map file is named FUNLIST.MAP. LINK searches for unresolved external references in the library FUNPROG.LIB before searching in the default libraries. LINK does not prompt for a .DEF file because a semicolon appears before the *deffile* field.

Example 2

```
LINK FUN, , ;
```

This command produces a map file named FUN.MAP because a comma appears as a placeholder for the *mapfile* field on the command line.

Example 3

```
LINK FUN, ;  
LINK FUN;
```

Neither of these commands produces a map file because commas do not appear as placeholders for the *mapfile* field. The semicolon (;) ends the command line and accepts all remaining defaults without prompting; the prompting default for the map file is not to create one.

Example 4

```
LINK MAIN+GETDATA+PRINTIT, , GETPRINT.LST;
```

This command links the files MAIN.OBJ, GETDATA.OBJ, and PRINTIT.OBJ. No module-definition file is specified, so LINK creates a DOS file if the real-mode default combined libraries are provided or a segmented executable file if the protect-mode libraries are provided. The map file GETPRINT.LST is created.

Example 5

```
LINK GETDATA+PRINTIT, , , , GETPRINT.DEF
```

This command links GETDATA.OBJ and PRINTIT.OBJ, using the information in GETPRINT.DEF. LINK creates a map file named GETDATA.MAP.

14.5 Running LINK

The simplest use of LINK is to combine one or more object files with a run-time library to create an executable file. You type `LINK` at the command-line prompt, followed by the names of the object files and a semicolon (;). LINK combines the object files with any language libraries specified in the object files to create an executable file. By default, the executable file takes the name of the first object file in the list.

To interrupt LINK and return to the operating-system prompt, press `CTRL+C` at any time.

LINK has five input fields, all optional except one (the *objfiles* field). There are several ways to supply the input fields LINK expects:

- Enter all the required input directly on the command line.
- Omit one or more of the input fields and respond when LINK prompts for the missing fields.
- Put the input in a response file and enter the response-file name (preceded by @) in place of the expected input.

These methods can be used in combination. The LINK command line was discussed on page 564. The following sections explain the other two methods.

Specifying Input with LINK Prompts

If any field is missing from the LINK command line and the line does not end with a semicolon, or if any of the supplied fields are invalid, LINK prompts you

for the missing or incorrect information. LINK displays one prompt at a time and waits until you respond:

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

The LINK prompts correspond to the command-line fields described earlier in this chapter. If you want LINK to prompt you for every input field, including *objfiles*, type the command `LINK` by itself.

Options can be entered anywhere in any field, before the semicolon if it is specified.

Defaults

The default values for each field are shown in brackets. Press `ENTER` to accept the default, or type in the filename(s) you want. The *basename* is the base name of the first object file you specified. To select the default responses for all the remaining prompts and terminate prompting, type a semicolon (;) and press `ENTER`.

If you specify a filename without giving an extension, LINK adds the appropriate default extension. To specify a filename that does not have an extension, type a period (.) after the name.

Use a space or plus sign (+) to separate multiple filenames in the *objfiles* and *libraries* fields. To extend a long *objfiles* or *libraries* response to a new line, type a plus sign (+) as the last character on the current line and press `ENTER`. You can continue entering your response when the same prompt appears on a new line. Do not split a filename or a path across lines.

Specifying Input in a Response File

You can supply input to LINK in a response file. A response file is a text file containing the input LINK expects on the command line or in response to prompts. You can use response files to hold frequently used options or responses or to overcome the 128-character limit on the length of a DOS command line.

Usage

Specify the name of the response file in place of the expected command-line input or in response to a prompt. Precede the name with an at sign (@), as in:

```
@responsefile
```

You must specify an extension if the response file has one; there is no default extension. You can specify a path with the filename.

You can specify a response file in any field (either on the command line or after a prompt) to supply input for one or more consecutive fields or all remaining fields. Note that LINK assumes nothing about the contents of the response file; LINK simply reads the fields from the file and applies them in order to the fields for which it has no input. LINK ignores any fields in the response file or on the command line after the five expected fields are satisfied or a semicolon (;) appears.

Example

The following command invokes LINK and supplies all input in a response file, except the last input field:

```
LINK @input.txt, mydefs
```

Contents of the Response File

Each input field must appear on a separate line, or separated from other fields on the same line by a comma. You can extend a field to the following line by adding a plus sign (+) at the end of the current line. A blank field can be represented by either a blank line or a comma.

Options can be entered anywhere in any field, before the semicolon if it is specified.

If a response file does not specify all the fields, LINK prompts you for the rest. Use a semicolon (;) to suppress prompting and accept the default responses for all remaining fields.

Example

```
FUN TEXT TABLE+
CARE
/MAP
FUNLIST
GRAF.LIB ;
```

If the preceding response file is named FUN.LNK, the command

```
LINK @FUN.LNK
```

causes LINK to:

- Link the four object files FUN.OBJ, TEXT.OBJ, TABLE.OBJ, and CARE.OBJ into an executable file named FUN.EXE.
- Include public symbols and addresses in the map file.

- Make the name of the map file FUNLIST.MAP.
- Link any needed routines from the library file GRAF.LIB.
- Assume no module-definition file.

14.6 LINK Options

This section explains how to use options to control LINK's behavior and modify LINK's output. It contains a brief introduction on how to specify options followed by a description of each option.

Specifying Options

The following paragraphs discuss rules for using options.

Syntax

All options begin with a slash (/). (A dash, -, is not a valid option specifier for LINK.) You can specify an option with its full name or an abbreviation, up to the shortest sequence of characters that uniquely identifies the option (except for /DOSSEG). The description for each option shows the minimum legal abbreviation with the optional part enclosed in double brackets. No gaps or transpositions of letters are allowed. For example,

`/B[[ATCH]]`

indicates that either /B or /BATCH can be used, as can /BA, /BAT, or /BATC. Option names are not case sensitive (except for /r), so you can also specify /batch or /Batch. This chapter uses meaningful yet legal forms of the option names. If an option is followed by a colon (:) and an argument, no spaces can appear before or after the colon.

Usage

LINK options can appear on the command line, in response to a prompt, or as part of a field in a response file. They can also be specified in the LINK environment variable. (For more information, see "Setting Options with the LINK Environment Variable" on page 593.) Options can appear in any field before the last input, except as noted in the descriptions.

If an option appears more than once (for example, on the command line and in the LINK variable), the effect is the same as if the option was given only once. If two options conflict, the most recently specified option takes effect. This means that a command-line option or one given in response to a prompt overrides one specified

in the LINK environment variable. For example, the command-line option /SEG:512 cancels the effect of the environment-variable option /SEG:256.

Numeric Arguments

Some LINK options take numeric arguments. You can enter numbers either in decimal format or in standard C-language notation.

The /ALIGN Option

Option

/A[[LIGNMENT]]:*size*

The /ALIGN option aligns segments in a segmented executable file at the boundaries specified by *size*. LINK ignores /ALIGN for DOS programs.

The alignment size is in bytes and must be an integer power of two. LINK rounds up to the next power of two if another value is specified. The default alignment is 512 bytes.

This option reduces the size of the file as it is stored on disk by reducing the size of gaps between segments. It has no effect on the size of the file when loaded in memory. The size of an executable file is limited to 64K times the alignment.

The /BATCH Option

Option

/B[[ATCH]]

The /BATCH option suppresses prompting for libraries or object files that LINK cannot find. By default, the linker prompts for a new path whenever it cannot find a library it has been directed to use. It also prompts you if it cannot find an object file that it expects to find on a floppy disk. When /BATCH is used, the linker generates an error or warning message (if appropriate). The /BATCH option also suppresses the LINK copyright message and echoed input from response files.

Using this option can cause unresolved external references. It is intended primarily for users who use batch files or makefiles for linking many executable files with a single command and who wish to prevent linker operation from halting.

Note This option does not suppress prompts for input fields. Use a semicolon (;) at the end of the LINK input to suppress input prompting.

The /CO Option

Option

/CO[[DEVIEW]]

The /CO option adds Microsoft Symbolic Debugging Information to the executable file. Debugging information can be used with the Microsoft CodeView debugger. If the object files do not contain debugging information (that is, if they were not compiled or assembled using either /Zi or /Zd), this option places only public symbols in the executable file.

You can run the resulting executable file outside CodeView; the debugging data in the file is ignored. However, it increases file size. You should link a separate release version without the /CO option after the program has been debugged.

When /CO is used with the /TINY option, debugging information is put in a separate file with the same base name as the .COM file and with the .DBG extension.

The /CO option is not compatible with the /EXEPACK option for DOS executable files.

The /CPARM Option

Option

/CP[[ARMAXALLOC]]:*number*

The /CPARM option sets the maximum number of 16-byte paragraphs needed by the program when it is loaded into memory. DOS uses this value to allocate space for the program before loading it. This option is useful when you want to execute another program from within your program and you need to reserve memory for the program. The /CPARM option is valid only for DOS programs.

LINK normally requests DOS to set the maximum number of paragraphs to 65,535. Since this is more memory than DOS can supply, DOS always denies the request and allocates the largest contiguous block of memory it can find. If the /CPARM option is used, DOS allocates no more space than the option specified. Any memory in excess of that required for the program loaded 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 the minimum value. This minimum 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 models, link with /CPARM:1. This leaves no space for the near heap.

Note You can change the maximum allocation after linking by using the EXEHDR utility, which modifies the executable-file header. For more information on EXEHDR, see Chapter 17.

The /DOSSEG Option

Option

/DOSS[[EG]]

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

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

In addition, the /DOSSEG option defines the following two labels:

```
__edata = DGROUP : BSS
__end   = DGROUP : STACK
```

The variables `__edata` and `__end` have special meanings for Microsoft compilers. It is recommended that you do not define program variables with these names. Assembly-language programs can reference these variables but should not change them.

The /DOSSEG option also inserts 16 null bytes at the beginning of the `_TEXT` segment (if this segment is defined); unassigned pointers point to this area. This behavior of the option is overridden by the /NONULLS option when both are used; use /NONULLS to override the DOSSEG comment record commonly found in standard Microsoft libraries.

This option is principally for use with assembly-language programs. When you link high-level-language programs, a special object-module record in the Microsoft language libraries automatically enables the /DOSSEG option. This option is also enabled by assembly modules that use Microsoft Macro Assembler (MASM) directive `.DOSSEG`.

Note The minimum abbreviation allowed for this option is /DOSS.

The /DSALLOC Option

Option

/DS[[ALLOCATE]]

The /DSALLOC option tells LINK to load all data starting at the high end of the data segment. At run time, the data segment (DS) register is set to the lowest data-segment address that contains program data.

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

The /DSALLOC option is most often used with the /HIGH option to take advantage of unused memory within the data segment. These options are valid only for assembly-language programs that create DOS .EXE files.

The /DYNAMIC Option

Option

/DY[[NAMIC]]:*number*

The /DYNAMIC option changes the limit on the number of interoverlay calls in an overlaid DOS program. (For more information on overlays, see Chapter 15.) The default limit is 256. The *number* is a decimal integer from 1 to 10,922. Specify a higher *number* to raise the limit if LINK issues the error `too many interoverlay calls`. Lower the limit to create a smaller table of interoverlay calls, saving space in your program.

To determine the most efficient *number*, run LINK using the /INFO option. The displayed information contains the line

```
NUMBER OF INTEROVERLAY CALLS: requested number; generated calls
```

The *number* of interoverlay calls requested is the *number* set by /DYNAMIC or the default of 256. The *calls* number reports the number of interoverlay calls actually generated. For maximum efficiency, run LINK using /INFO, then relink using /DYNAMIC:*calls*.

The /EXEPACK Option

Option

/E[[XEPACK]]

The /EXEPACK option directs LINK to remove sequences of repeated bytes (usually 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.)

The /EXEPACK option does not always produce 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 issues a warning if the packed file is larger than the unpacked file. The time required to expand a packed file may cause it to load more slowly than a file linked without this option.

You cannot debug packed DOS files with CodeView because the unpacker that /EXEPACK appends to a DOS program is incompatible with CodeView. In a Windows program, the unpacker is in the loader, and there is no conflict with CodeView.

The /EXEPACK option is not compatible with the /HIGH or /Q option.

The /FARCALL Option

Option

/F[[ARCALLTRANSLATION]]

The /FARCALL option directs the linker to optimize far calls to procedures that lie in the same segment as the caller. This can result in slightly faster code; the gain in speed is most apparent on 80286-based machines and later.

The /FARCALL option is on by default for overlaid DOS programs and programs created with the /TINY option. It is off by default for other programs. If an environment variable (such as LINK or CL) includes /FARCALL, you can use the /NOFARCALL option to override it. The /PACKC option is not recommended when linking Windows applications with /FARCALL.

FARCALL optimizes by creating more efficient code.

A program that has multiple code segments may make a far call to a procedure in the same segment. Since the segment address is the same (for both the code and the procedure it calls), only a near call is necessary. Far calls appear in the relocation table; a near call does not require a table entry. By converting far calls to near calls in the same segment, the /FARCALL option both reduces the size of the relocation table and increases execution speed because only the offset needs to be

loaded, not a new segment. The `/FARCALL` option has no effect on programs that make only near calls since there are no far calls to convert.

When `/FARCALL` is specified, the linker optimizes code by removing the instruction `call FAR label` and substituting the following sequence:

```
    nop
    push    cs
    call    NEAR label
```

During execution, the called procedure still returns with a far-return instruction. However, because both the code segment and the near address are on the stack, the far return is executed correctly. The `nop` (no-op) instruction is added so that exactly five bytes replace the five-byte far-call instruction.

**In rare cases,
/FARCALL should be
used with caution.**

There is a small risk with the `/FARCALL` option. If LINK sees the far-call opcode (9A hexadecimal) followed by a far pointer to the current segment, and that segment has a class name ending in `CODE`, it interprets that as a far call. This problem can occur when using `__based (__segment ("_CODE"))` in a C program. If a program linked with `/FARCALL` fails for no apparent reason, try using `/NOFARCALL`.

Object modules produced by Microsoft high-level languages are safe from this problem because little immediate data is stored in code segments. Assembly-language programs are generally safe for use with the `/FARCALL` option if they do not involve advanced system-level code, such as might be found in operating systems or interrupt handlers.

The /HELP Option

Option

`/HE[[LP]]`

The `/HELP` option calls the QuickHelp utility. If LINK cannot find the Help file or QuickHelp, it displays a brief summary of LINK command-line syntax and options. Do not give a filename when using the `/HELP` option.

The /HIGH Option

Option

`/HI[[GH]]`

At load time, the executable file can be placed either as low or as high in memory as possible. The `/HIGH` option causes DOS to place the executable file as high as possible in memory. Without the `/HIGH` option, DOS places the executable file as

low as possible. This option is usually used with the `/DSALLOC` option. These options are valid only for assembly-language programs that create DOS `.EXE` files.

The `/INFO` Option

Option

`/INF[[ORMATION]]`

The `/INFO` option displays to the standard output information about the linking process, including the phase of linking, the object files being linked, and the library modules used. This option is useful for determining the locations of the object files and modules, the number of segments, and the order in which they are linked.

An important use of `/INFO` is to get the number of interoverlay calls generated. You can then specify this number with the `/DYNAMIC` option.

The `/LINE` Option

Option

`/LI[[NENUMBERS]]`

The `/LINE` option adds the line numbers and associated addresses from source files to the map file. The object file must contain line-number information for it to appear in the map file. If the object file has no line-number information, the `/LINE` option has no effect. (Use the `/Zd` or `/Zi` option with Microsoft compilers such as `CL`, `FL`, and `ML` to add line numbers to the object file.) If you also want to add public symbols or other information to the map file, use the `/MAP` option. For more information on the map file, see the description of `/MAP`.

The `/LINE` option causes a map file to be created even if you did not explicitly tell the linker to do so. `LINK` creates a map file when a filename is specified in the *mapfile* field or when the default map-file name is accepted. (The `/MAP` option also forces creation of a map file.) For more information, see the description of *mapfile* on page 567.

By default, the map file is given the same base name as the executable file with the extension `.MAP`. You can override the default name by specifying a new map-file name in the *mapfile* field or in response to the `List File` prompt.

The /MAP Option

Option

/M[[AP]][[:{*maptype*}]]

The /MAP option controls the information contained in the map file. The /MAP option causes a map file to be created even if you did not explicitly tell the linker to do so.

LINK creates a map file when a filename is specified in the *mapfile* field or when the default map-file name is accepted. (The /LINE option also forces creation of a map file.) For more information, see the description of *mapfile* on page 567.

A map file by default contains only a list of segments. A map file created with /MAP contains public symbols sorted by name and by address, in addition to the segments list. Symbols in C++ appear in the form of decorated names. To add or omit information, specify /MAP followed by a colon (:) and a *maptype* qualifier:

[[ADDRESS]]

Omits the list of public symbols sorted by name.

[[FULL]]

Adds information about each object file's contribution to a segment. Adds undecorated names following the decorated names for C++ symbols in the listings by name and by address.

If you also want to add line numbers to the map file, use the /LINE option.

By default, the map file is given the same base name as the executable file with the extension .MAP. You can override the default name by specifying a new map filename in the *mapfile* field or in response to the List File prompt.

Under some circumstances, adding symbols slows the linking process. If this is a problem, do not use /MAP.

The /NOD Option

Option

/NOD[[EFAULTLIBRARYSEARCH]][[:*libraryname*]]

The /NOD option tells LINK not to search default libraries named in object files. Specify *libraryname* to tell LINK to exclude only *libraryname* from the search. If you want LINK to ignore more than one library, specify /NOD once for each library. To tell LINK to ignore all default libraries, specify /NOD without a *libraryname*. For more information, see “Default Libraries” on page 568.

High-level-language object files usually must be linked with a run-time library to produce an executable file. Therefore, if you use the /NOD option, you must also use the *libraries* field to specify an alternate library that resolves the external references in the object files. If you compile a program using Microsoft C 7.0 or later and you specify /NOD, you must also specify OLDNAMES.LIB.

The /NOE Option

Option

/NOE[[XTDICTIONARY]]

The /NOE option prevents the linker from searching extended dictionaries when resolving references. An extended dictionary is a list of symbol locations in a library created with LIB. The linker consults extended dictionaries to speed up library searches. Using /NOE slows the linker.

When LINK uses extended dictionaries, it gives an error if a duplicate definition is found. Use this option when you redefine a symbol and an error occurs. For more information, see “How LINK Resolves References” on page 568.

The /NOFARCALL Option

Option

/NOF[[ARCALLTRANSLATION]]

The /NOFARCALL option turns off far-call optimization (translation). Far-call optimization is off by default. However, if an environment variable (such as LINK or FL) includes the /FARCALL option, you can use /NOFARCALL to override /FARCALL.

The /NOGROUP Option

Option

/NOG[[ROUPASSOCIATION]]

The /NOGROUP option ignores group associations when assigning addresses to data and code items. It is provided primarily for compatibility with previous versions of the linker (2.02 and earlier) and early versions of Microsoft compilers. This option is valid only for assembly-language programs that create DOS .EXE files.

The /NOI Option

Option

`/NOI[[GNORECASE]]`

This option preserves case in identifiers. By default, LINK treats uppercase 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 and lowercase and considers these identifiers to be three different names.

In most high-level languages, identifiers are not case sensitive, so this option has no effect. However, case is significant in C. It's a good idea to use this option with C programs to catch misnamed identifiers.

The /NOLOGO Option

Option

`/NOL[[OGO]]`

The `/NOLOGO` option suppresses the copyright message displayed when LINK starts. This option has no effect if not specified first on the command line or in the LINK environment variable.

The /NONULLS Option

Option

`/NON[[ULLSDOSSEG]]`

The `/NONULLS` option arranges segments in the same order 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), but `/NONULLS` does not insert the extra bytes.

If both the `/DOSSEG` and `/NONULLS` options are given, the `/NONULLS` option takes precedence. Therefore, you can use `/NONULLS` to override the `DOSSEG` comment record found in run-time libraries. This option is for segmented executable files.

The /NOPACKC Option

Option

`/NOPACKC[[ODE]]`

This option turns off code-segment packing. Code-segment packing is on by default for segmented executable files and for DOS programs created with overlays or with the /TINY option. It is off by default for other DOS programs. If an environment variable (such as LINK or CL) includes the /PACKC option to turn on code-segment packing, you can use /NOPACKC to override /PACKC. For more information on packing, see “The /PACKC Option” on page 587.

NOTE The minimum unique abbreviation for /NOPACKC has changed from /NOP to /NOPACKC.

The /NOPACKF Option

Option

`/NOPACKF[[UNCTIONS]]`

This option prevents the removal of unreferenced packaged functions. Removal of such definitions (the /PACKF option) is usually on by default. Use /NOPACKF to preserve these definitions. For example, you may want to keep unreferenced code in a debugging version of your program. For more information on /PACKF and packaged functions, see page 589.

The /OLDOVERLAY Option

Option

`/OL[[DOVERLAY]]`

This option links an overlaid DOS program using the Static Overlay Manager instead of the MOVE library. This option may not be supported in future versions of LINK. For information about overlays, see Chapter 15.

The /ONERROR Option

Option

`/ON[[ERROR]]:N[[OEXE]]`

The /ONERROR option tells LINK what to do if an error occurs. By default, if certain errors occur, LINK writes an executable file to disk and overwrites any

existing file having the same name. The resulting executable file has the error bit set in its header. Specify `/ONERROR:NOEXE` to prevent such a file from being written to disk and preserve any existing file having the same name. The `/ONERROR` option can be useful in makefiles.

The /OV Option

Option

`/OV[[ERLAYINTERRUPT]]:number`

This option sets an interrupt number for passing control to overlays. By default, the interrupt number used for passing control to overlays is 63 (3F hexadecimal). The `/OV` option allows you to select a different interrupt number. This option is valid only when linking overlaid DOS programs.

The *number* can be any number from 0 to 255, specified in decimal format or in C-language notation. 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 use overlays with a program that reserves interrupt 63 for some other purpose.

Note The minimum unique abbreviation for `/OV` has changed from `/O` to `/OV`.

The /PACKC Option

Option

`/PACKC[[ODE]][:number]`

The `/PACKC` option turns on code-segment packing. Code-segment packing is on by default for segmented executable files and for DOS programs created with overlays or with the `/TINY` option. It is off by default for other DOS programs. You can use the `/NOPACKC` option to override `/PACKC`.

The linker packs physical code segments by grouping neighboring logical code segments that have the same attributes. Segments in the same group are assigned the same segment address; offset addresses are adjusted accordingly. All items have the same physical address whether or not the `/PACKC` option is used. However, `/PACKC` changes the segment and offset addresses so that all items in a group share the same segment.

The *number* specifies the maximum size of groups formed by `/PACKC`. The linker stops adding segments to a group when it cannot add another segment without exceeding *number*. It then starts a new group. The default segment size without `/PACKC` (or when `/PACKC` is specified without *number*) is 65,500 bytes (64K – 36 bytes).

The `/PACKC` option produces slightly faster and more compact code. It affects only programs with multiple code segments.

Code-segment packing provides more opportunities for far-call optimization (which is enabled with the `/FARCALL` option). The `/FARCALL` and `/PACKC` options together produce faster and more compact code. However, this combination is not recommended for Windows applications.

Use caution when packing assembly-language programs.

Object code created by Microsoft compilers can safely be linked with the `/PACKC` option. This option is unsafe only when used with assembly-language 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 produces incorrect results when used with `/PACKC` because `/PACKC` causes the two segments to share the same 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
            shl  ax, cl     ; Convert distance from paragraphs
                                ; to bytes

codesize  ENDP

CSEG2      ENDS
```

The `/PACKD` Option

Option

`/PACKD`[[`ATA`]][[:*number*]]

The `/PACKD` option turns on data-segment packing. The linker considers any segment definition with a class name that does not end in `CODE` as a data segment. Adjacent data-segment definitions are combined into the same physical segment. The linker stops adding segments to a group when it cannot add another segment without exceeding *number* bytes. It then starts a new group. The default segment size

without `/PACKD` (or when `/PACKD` is specified without *number*) is 65,536 bytes (64K).

The `/PACKD` option produces slightly faster and more compact code. It affects only programs with multiple data segments and is valid only for segmented executable files. It might be necessary to use the `/PACKD` option to get around the limit of 254 physical data segments per executable file imposed by an operating system. Try using `/PACKD` if you get the following LINK error:

```
L1073 file-segment limit exceeded
```

This option may not be safe with other compilers that do not generate fixup records for all far data references.

The `/PACKF` Option

Option

`/PACKF[[UNCTIONS]]`

The `/PACKF` option removes unreferenced “packaged functions.” This behavior is the default. However, if an environment variable (such as `LINK` or `CL`) includes the `/NOPACKF` option, you can use `/PACKF` to override `/NOPACKF`.

A packaged function is visible to the linker in the form of a `COMDAT` record. Packaged functions are created when you use the `/Gy` option on the `CL` command line (or, in `PWB`, when you choose Enable Function Level Linking in the Additional Global Options dialog box, which is available from the C or C++ Compiler Options dialog boxes). Member functions in a C++ program are automatically packaged.

If a packaged function is defined but not called, this option removes the function definition from the executable file. `/PACKF` is not recursive; LINK does not remove any external definitions brought in by the unused packaged function.

The `/PAUSE` Option

Option

`/PAU[[SE]]`

The `/PAUSE` option pauses the session before LINK writes the executable file or DLL to disk. This option is supplied for compatibility with machines that have two floppy drives but no hard disk. It allows you to swap floppy disks before LINK writes the executable file.

If you specify the `/PAUSE` option, LINK displays the following message before it creates the main output:

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

The *letter* is the current drive. LINK resumes processing when you press ENTER.

Do not remove a disk that contains either the map file or the temporary file. If LINK creates a temporary file on the disk you plan to remove, end the LINK session and rearrange your files so that the temporary file is on a disk that does not need to be removed. For more information on how LINK determines where to put the temporary file, see “LINK Temporary Files” on page 595.

The /PM Option

Option

`/PM[[TYPE]]:type`

This option specifies the type of Windows application being generated. The `/PM` option is equivalent to including a type specification in the `NAME` statement in a module-definition file.

The *type* field can take one of the following values:

PM

The default. Windows application. The application uses the API provided by Windows and must be executed in the Windows environment. This is equivalent to `NAME WINDOWAPI`.

VIO

Character-mode application to run in a text window in the Windows session. This is equivalent to `NAME WINDOWCOMPAT`.

NOVIO

Character-mode application that must run full screen in Windows. This is equivalent to `NAME NOTWINDOWCOMPAT`.

The /Q Option

Option

`/Q[[UICKLIBRARY]]`

The `/Q` option directs the linker to produce a “Quick library” instead of an executable file. A Quick library is similar to a standard library because both contain routines that can be called by a program. However, a standard library is linked

with a program at link time; in contrast, a Quick library is linked with a program at run time.

When /Q is specified, the *exefile* field refers to a Quick library instead of an application. The default extension for this field is then .QLB instead of .EXE.

Quick libraries can be used only with programs created with Microsoft Quick-Basic or early versions of Microsoft QuickC. These programs have the special code that loads a Quick library at run time.

The /r Option

Option

/r

Prevents LINK from using extended memory under DOS. The /r option must appear first in the options field on the command line and cannot appear in a response file or the LINK environment variable. LINK.EXE is extender-ready and uses extended memory if it exists. This option forces LINK to use only conventional memory. The option name is case sensitive.

For LINK to run in DOS-extended mode, sufficient extended memory must be available. The memory must be provided by one of the following:

- A DOS Protected-Mode Interface (DPMI) server, such as that provided in a DOS box in Windows enhanced mode
- A Virtual Control Program Interface (VCPI) server, such as Microsoft's EMM386.EXE
- An XMS driver, such as Microsoft's HIMEM.SYS

The /SEG Option

Option

/SE[[GMENTS]][[:*number*]]

The /SEG option sets the maximum number of program segments. The default without /SEG or *number* is 128. You can specify *number* as any value from 1 to 16,384 in decimal format or C-language notation. However, the number of segment definitions is constrained by available memory.

LINK must allocate some memory to keep track of information for each segment; the larger the *number* you specify, the less free memory LINK has to run in. A relatively low segment limit (such as the 128 default) reduces the chance that LINK will run out of memory. For programs with fewer than 128 segments, you

can minimize LINK's memory requirements by setting *number* to reflect the actual number of segments in the program. If a program has more than 128 segments, however, you must set a higher value.

If the number of segments allocated is too high for the amount of memory available while linking, LINK displays the error message:

```
L1054 requested segment limit too high
```

When this happens, try linking again after setting /SEG to a smaller number.

The /STACK Option

Option

/ST[[ACK]]:*number*

The /STACK option lets you change the stack size from its default value of 2048 bytes. The *number* is any positive even value in decimal or C-language notation up to 64K-2. If an odd number is specified, LINK rounds up to the next even value. Do not specify /STACK for a DLL.

Programs that pass large arrays or structures by value or with deeply nested sub-routines may need additional stack space. In contrast, if your program uses the stack very little, you might be able to save space by decreasing the stack size. If a program fails with a stack-overflow message, try increasing the size of the stack.

Note You can also use the EXEHDR utility to change the default stack size by modifying the executable-file header. For more information on EXEHDR, see Chapter 17.

The /TINY Option

Option

/T[[INY]]

The /TINY option produces a .COM file instead of an .EXE file. The default extension of the output file is .COM. When the /CO option is used with /TINY, debug information is put in a separate file with the same base name as the .COM file and with the .DBG extension.

Not every program can be linked in the .COM format. The following restrictions apply:

- The program must consist of only one physical segment. You can declare more than one segment in assembly-language programs; however, the segments must be in the same group.
- The code must not use far references.
- Segment addresses cannot be used as immediate data for instructions. For example, you cannot use the following instruction:

```
mov     ax, CODESEG
```
- Windows programs cannot be converted to a .COM format.

The /W Option

Option

/W[[ARNFIXUP]]

The /W option issues the L4000 warning when LINK uses a displacement from the beginning of a group in determining a fixup value. This option is provided because early versions of the Windows linker (LINK4) performed fixups without this displacement. This option is for linking segmented executable files.

The /? Option

Option

/?

The /? option displays a brief summary of LINK command-line syntax and options.

14.7 Setting Options with the LINK Environment Variable

You can use the LINK environment variable to set options that will be in effect each time you link. (Microsoft compilers such as CL, FL, and ML also use the options in the LINK environment variable.)

Setting the LINK Environment Variable

You set the LINK environment variable with the following operating-system command:

```
SET LINK=options
```

LINK expects to find *options* listed in the variable exactly as you would type them in fields on the command line, in response to a prompt, or in a response file. It does not accept values for LINK's input fields; filenames in the LINK variable cause an error.

Example

```
SET LINK=/NOI /SEG:256 /CO
LINK TEST;
LINK /NOD PROG;
```

In the preceding example, the commands are specified at the system prompt. The file TEST.OBJ is linked using the options /NOI, /SEG:256, and /CO. The file PROG.OBJ is then linked with the option /NOD, in addition to /NOI, /SEG:256, and /CO.

Behavior of the LINK Environment Variable

You can specify options in the LINK input fields and in the LINK environment variable. LINK reads the options set in the LINK environment variable before it reads options specified in LINK input fields. This priority has the following effects:

- The option LINK considers to be first is the first one in the LINK environment variable, if set. The /NOLOGO option behaves differently depending on whether or not it is first. However, the /r option cannot be specified in the LINK variable and must be specified first on the command line.
- An option specified multiple times with different values will get the last value read by LINK. For example, if /SEG:512 is set in an input field, it overrides a setting of /SEG:256 in the LINK variable.
- For some options, if an option appears in the LINK variable and a conflicting option appears in an input field, the input-field option overrides the environment-variable option. For example, the input-field option /NOPACKC overrides the environment-variable option /PACKC.

Clearing the LINK Environment Variable

You must reset the LINK environment variable to prevent LINK from using its options. To clear the LINK variable, use the operating-system command:

```
SET LINK=
```

To see the current setting of the LINK variable, type `SET` at the operating-system prompt.

14.8 LINK Temporary Files

LINK uses available memory during the linking session. If LINK runs out of memory, it creates a disk file to hold intermediate files. LINK deletes this file when it finishes.

When the linker creates a temporary disk file, you see the following message:

```
Temporary file tempfile has been created.  
Do not change diskette in drive, letter.
```

In the preceding message, *tempfile* is the name of the temporary file, and *letter* is the drive containing the temporary file. (The second line appears only for a floppy drive.)

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 might see the following message:

```
Unexpected end-of-file on scratch file
```

If this happens, run LINK again.

Location of the Temporary File

If the TMP environment variable defines a temporary directory, LINK creates temporary files there. If the TMP environment variable is undefined or the temporary directory doesn't exist, LINK creates temporary files in the current directory.

Name of the Temporary File

When running under DOS version 3.0 or later, LINK asks the operating system to create a temporary file with a unique name in the temporary-file directory.

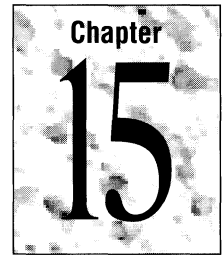
Under DOS versions earlier than 3.0, LINK creates a temporary file named VM.TMP. Do not use this name for your files. LINK generates an error message if it encounters an existing file with this name.

14.9 LINK Exit Codes

LINK returns an exit code (also called return code or error code) that you can use to control the operation of batch files or makefiles.

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: <ul style="list-style-type: none">▪ Ran out of space on output files.▪ Was unable to reopen the temporary file.▪ Experienced an internal error.▪ Was interrupted by the user.

Creating Overlaid DOS Programs



This chapter describes how to create a DOS program that uses overlays. An overlaid program can run in conventional memory that would be insufficient for a program that doesn't use overlays.

The tools for creating overlaid programs are LINK 5.30 and the Microsoft Overlaid Virtual Environment (MOVE). MOVE is provided in the library MOVE.LIB, which is a component of the combined libraries for medium and large models provided with Microsoft C version 7.0 and later.

The MOVE method of creating overlays is a successor to the Microsoft Static Overlay Manager supported by earlier versions of LINK. For a comparison of the two methods and a discussion of the features still supported, see "Dynamic and Static Overlays" on page 604.

MOVE provides ways to customize your overlaid program. These advanced features are discussed in the MOVEAPI.TXT file provided on disk.

15.1 Overview

LINK can create DOS programs with up to 2047 "overlays." Overlays are sections of a program that are loaded into memory only as needed. With overlays, you can run a DOS program that would otherwise be too large to fit into available memory. Under DOS, available memory is the portion of conventional memory that is not taken up by system or other programs. An overlaid program can fit into available memory if the size of its root (the part of the program that always remains in memory) and the memory needed for the overlay area do not exceed the size of available memory.

An overlaid program consists of one .EXE file. The file remains open during program execution. MOVE reads this file when it needs to load an overlay. Overlays are loaded on call; initially, only the root of the program is loaded into memory.

Overlays occupy a heap located in the remaining available memory that is not occupied by the root. More than one overlay can be in memory at a time. When the program calls a function contained in an overlay that is not already in memory, the overlay is copied into the overlay heap from the executable file. If the overlay heap does not have enough room for the new overlay, an existing overlay is discarded. MOVE uses a form of a least recently used (LRU) algorithm to determine which overlay to discard.

If expanded or extended memory is available, MOVE can cache the discarded overlays in this memory. Caching saves time when a discarded overlay is needed again because copying an overlay from expanded or extended memory to conventional memory takes less time than reading it in from the disk. If the cache is too small to hold all discarded overlays, MOVE uses a form of an LRU algorithm to decide which overlay to overwrite. If neither extended nor expanded memory is available, MOVE must open the executable file and read in each overlay as it is needed.

Overlaid programs execute more slowly than nonoverlaid programs because of the time needed to swap overlays. You can optimize your program by grouping functions that call each other into the same overlay or into overlays that are in memory at the same time.

Only code segments can be put into overlays. Data segments must go into the root. To swap data to disk or to expanded or extended memory, use virtual memory. See “Using the Virtual Memory Manager” in Chapter 4, “Managing Memory in C,” in the *Programming Techniques* manual.

15.2 How to Create an Overlaid Program

To create a program that uses overlays, first decide how you want to organize your code. All code not explicitly or implicitly assigned to an overlay is placed in the root. You can use both compiler and linker features to specify the organization of your code. MOVE uses a module-definition (.DEF) file to describe overlays.

LINK automatically creates an overlaid DOS program when you provide a module-definition file in which a **SEGMENTS** or **FUNCTIONS** statement specifies an overlay number. (LINK also creates an overlaid program when it detects parentheses in the objects field on the command line; see “Dynamic and Static Overlays” on page 604.)

The sections that follow describe overlay-related compiler features, module-definition statements, and LINK options.

Compiling for Overlays

You need to be aware of the following when you write and compile a C program that will be overlaid:

Memory Models

An overlaid program must be in either a medium or large model. Only the medium and large combined libraries in C 7.0 contain the MOVE library. Code references in an overlaid program must be far.

Packaged Functions

An individual function can be assigned to an overlay independently from other functions in the same segment only if it is a “packaged function.” A packaged function is visible to the linker in the form of a COMDAT record. To compile a C function as a packaged function, use the /Gy option on the CL command line (or in the Microsoft Programmer’s WorkBench, choose Enable Function Level Linking from the Additional Global Options dialog box, which is available from the C or C++ Compiler Options dialog boxes). In a C++ program, member functions are automatically packaged.

If an object file is compiled without /Gy, relocation information is not available to the linker. A function that is not packaged cannot be assigned to an overlay; however, the segment in which it resides can be placed in an overlay.

Explicit Allocation

An explicitly allocated function is one that is assigned to a segment at compile time. Its behavior is described in “Creating the Module-Definition File” on page 600. The following methods of explicit allocation are available:

- In C source code, the **__based** keyword (or its predecessor, the **alloc_text** pragma) specifies the segment where an individual function is to reside.
- On the CL command line, the /NT option specifies the segment where all functions in an object file are to reside.

Anonymous Allocation

A function not explicitly allocated to a segment is sometimes referred to as an anonymous function. In programs compiled in medium and large models, anonymous functions are allocated to a segment that has a name in the form:

*objfile_***TEXT**

where *objfile* is the name of the object file containing the functions.

MOVE Library Routines

The MOVE library provides several routines for use in advanced techniques of overlay programming. These routines are described in the MOVEAPI.TXT file distributed with Microsoft C version 7.0. Some uses of these routines are discussed in “Memory Allocation” on page 602.

Creating the Module-Definition File

To specify the organization of your overlaid program, write a “module-definition file.” A module-definition (.DEF) file is a text file that describes a program’s characteristics. This information is used by LINK. Valid statements in a .DEF file for an overlaid program are **SEGMENTS**, **FUNCTIONS**, and **INCLUDE**. LINK ignores other module-definition statements when linking an overlaid DOS program. Chapter 16 describes all the features of .DEF files.

Note The predecessor to MOVE, the Microsoft Static Overlay Manager, uses parentheses in the command-line syntax to place entire object files into overlays. This method of specifying overlays is still supported; however, conflicts may arise if you also use a module-definition file. A .DEF file is the preferred method and gives more control in assigning code to overlays.

The FUNCTIONS and SEGMENTS Statements

The **FUNCTIONS** and **SEGMENTS** statements place code in overlays. Use the **FUNCTIONS** statement to assign an individual function to either an overlay or to a segment. Use the **SEGMENTS** statement to assign a segment an overlay. The **SEGMENTS** statement is described on page 619; the **FUNCTIONS** statement is described on page 625.

Valid overlay numbers are from 1 through 65,535; to place code in the root, specify 0. The space allocated for overlay information is determined by the highest overlay number, whether or not every intermediate number is actually used. For the most efficient use of space, it is recommended that you use a continuous sequence of numbers beginning with 1.

You can use the **FUNCTIONS** and **SEGMENTS** statements to organize overlays in the following ways:

- To place an entire segment into an overlay, use the **SEGMENTS** statement specified with an overlay number. For example, the following statement places the segment called `myseg` into the second overlay:

```
SEGMENTS myseg OVL:2
```

- To place individual functions into an overlay, use the **FUNCTIONS** statement specified with an overlay number. The functions must be compiled as packaged functions (see “Compiling for Overlays” on page 599). The following statement places three functions into the second overlay:

```
FUNCTIONS:2 myfunc1 myfunc2 myfunc3
```

If the function is explicitly allocated, it cannot be assigned to a different overlay from the segment that contains the function. Explicit allocation is described in “Compiling for Overlays” on page 599.

- To place individual functions into a segment and then place the segment into an overlay, use the **FUNCTIONS** statement specified with a segment name and the **SEGMENTS** statement specified with the segment name and an overlay number. The functions must be compiled as packaged functions. An explicitly allocated function cannot be assigned to a segment different from the one to which it was allocated. The following statements assign `myseg` to the second overlay and place three functions into `myseg`:

```
SEGMENTS myseg OVL:2  
FUNCTIONS:myseg myfunc1 myfunc2 myfunc3
```

- To assign an explicitly allocated function to an overlay, use the **FUNCTIONS** statement specified with an overlay number, and do not assign its segment or any other functions in that segment to an overlay. The **FUNCTIONS** statement, in this case, implicitly assigns the entire segment to the overlay. For example, if `myfunc` is explicitly allocated to the segment `myseg`, and `myseg` and its other functions are not assigned to an overlay, the following statement implicitly places all code in `myseg` into the second overlay:

```
FUNCTIONS:2 myfunc
```

The INCLUDE Statement

The **INCLUDE** statement inserts a specified text file at the place where it is specified in a `.DEF` file. The **INCLUDE** statement behaves the same way in `.DEF` files for all program types. This statement is described on page 627. For example, the following statement tells `LINK` to read `SEGMENTS.TXT` before it processes the rest of the `.DEF` file:

```
INCLUDE segments.txt
```

Linking the Overlaid Program

`LINK` automatically creates an overlaid program when overlays are specified; no special option is required. However, you can use the `/DYNAMIC` and `/INFO` options to make your program more efficient.

The `/DYNAMIC` option sets the limit on the number of interoverlay calls; the default is 256. `LINK` creates table space in multiples of this limit. You can save space in your program by setting a lower limit. Use the `/INFO` option to get the actual number of calls generated. Raise the limit if `LINK` issues the error `too many interoverlay calls`.

The `/DYNAMIC` and `/INFO` options are described in Chapter 14.

Note The predecessor to `MOVE`, the Microsoft Static Overlay Manager, is still available. To link using static overlays, specify the `/OLDOVERLAY` option. For more information, see “Using the Static Overlay Manager” on page 605.

15.3 How MOVE Works

This section describes how memory management is handled under `MOVE` and the restrictions you need to be aware of when creating an overlaid program.

Memory Allocation

`MOVE` allocates a heap in conventional memory for overlays in use. `MOVE` can also allocate a cache in extended and expanded memory. The following sections describe how allocations are performed.

The Overlay Heap

The overlay heap is allocated in available conventional memory at run time. The minimum size of the heap allocated by `MOVE` is the size of the largest overlay, and the maximum is the sum of the sizes of the three largest overlays. You can change these defaults by rewriting the `_moveinit` routine so that different values are passed to the `_movesetheap` routine. These and other `MOVE` routines are described in the `MOVEAPI.TXT` file on your distribution disks. The `_moveinit` routine is provided in the `MOVEINIT.C` file, and all `MOVE` routines are defined in the `MOVEAPI.H` include file.

The Overlay Cache

The cache for discarded overlays is allocated at run time in extended and expanded memory. The amount of memory available in each affects whether the cache is allocated in either extended memory or expanded memory or both. The `_moveinit` routine calls the `_movesetcache` routine to set how much of each kind of memory is used. You can rewrite `_moveinit` to change the values that are passed to `_movesetcache`.

If your program spawns another program that requires expanded or extended memory, you can use the `_movepause` and `_moveresume` routines to release the cache memory temporarily and subsequently restore it.

Limits and Requirements

The following sections describe various MOVE requirements and restrictions.

Compatibility

Overlaid programs created by MOVE require DOS version 3.0 or later.

MOVE can make use of expanded memory and extended memory. Expanded memory must conform to the Lotus/Intel/Microsoft (LIM) EMS version 4.0 or later. Extended memory must conform to the XMS provided in a memory manager such as HIMEM.SYS.

The Microsoft CodeView debugger version 4.0 is compatible with overlaid programs created with MOVE. CodeView version 4.0 does not support the Static Overlay Manager.

Space Restrictions

An overlaid program takes up more space on disk than the same program created without overlays. Each overlay contains an executable-file header with the fixups for that overlay; an overlay header occupies a multiple of 512 bytes. The root contains the overlay manager and a segment that holds information on each function in each overlay.

The amount of conventional memory the program requires is the sum of:

- The size of the root. The root includes the overlay manager (about 5K), the segment that contains information on overlays, and all functions and segments that are not placed into overlays. To determine this size, run the EXEHDR utility on the overlaid program and read the value in the `Memory needed:` field.
- The size of the overlay heap. The minimum heap allocated by MOVE is equal to the size of the largest overlay.

The Overlaid Program

You can use overlays only in programs with multiple code segments because separate segment names are needed for overlays. Far call and return instructions are required between overlays.

Up to 2047 overlays can be specified. The program can define up to 16,375 logical segments (segments with different names).

The entry point to the program must be in the root.

Only code segments are overlaid, not data. Data segments become part of the root and are always in memory. Code linked in from the run-time libraries is placed in the root.

Note An advanced technique can be used to place data into an overlay. You can create a source file that declares a code segment containing data and assign this segment to an overlay. In a Microsoft C program, you can do this by using `__based(__segname("_CODE"))`; you can also use assembly language. Because an overlay may be discarded and changes in data may be lost, this method is useful only for large data items that do not change and are not frequently accessed.

An Individual Overlay

An overlay size is limited to 64K. The optimal size is around 4K. An overlay can contain one or more segments.

Programming Considerations

You cannot jump between overlays using the **setjmp** and **longjmp** C library functions. You can use long jumps from an overlay to the root. It is possible but not recommended to use long jumps within an overlay.

You cannot use the same public name in different overlays.

The interrupt number used by **MOVE** is 63 (3F hexadecimal). You can use **LINK's /OV** option to change the interrupt number.

Do not place **MOVE.LIB** into an overlay.

15.4 Dynamic and Static Overlays

MOVE, the Microsoft Overlaid Virtual Environment, differs from its predecessor, the Microsoft Static Overlay Manager, in several ways. This section discusses some of these differences.

Specifying Overlays on the Command Line

The syntax for the Static Overlay Manager is supported by **MOVE**, with caution. In this syntax, static overlays are specified in parentheses on the command line.

For details about this syntax, consult the documentation for an earlier version of LINK. This syntax may not be supported by future versions of MOVE and LINK.

It is recommended that you replace the parentheses specification with a module-definition file that specifies the desired organization. If both parentheses and a .DEF file are used, and a conflict occurs, the .DEF file statements take precedence. Overlays specified with parentheses are assigned overlay numbers sequentially beginning with 1.

Using the Static Overlay Manager

You can still link with the Static Overlay Manager instead of MOVE. Specify parentheses in the command-line syntax, and then link using the /OLDOVERLAY option. For information on /OLDOVERLAY, see page 586. This option may not be supported by future versions of LINK.

Advantages of MOVE

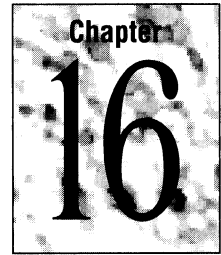
A program created with MOVE can have multiple overlays in memory at one time. The Static Overlay Manager permits only one overlay in memory at a time.

MOVE permits interoverlay calls through pointers to functions. Overlays created with the Static Overlay Manager cannot call each other in this way.

MOVE can cache discarded overlays in expanded and extended memory; this provides quick access if an overlay is called again. The Static Overlay Manager has to reopen the executable file each time an overlay is called.

The maximum number of overlays allowed by MOVE is 2047. The Static Overlay Manager allows only 255 overlays.

Creating Module-Definition Files



This chapter describes the contents of a module-definition (.DEF) file. It begins with a brief overview of the purpose of .DEF files. The rest of the chapter discusses each statement in a module-definition file and describes syntax rules, argument fields, attributes, and keywords for each statement.

The statements in this chapter are supported by the following utilities:

- Microsoft Segmented Executable Linker (LINK) version 5.30
- Microsoft Import Library Manager (IMPLIB) version 1.30
- Microsoft EXE File Header Utility (EXEHDR) version 2.02

16.1 New Features

The latest version of the linker and other utilities support the statements and keywords described in this chapter. The following sections introduce features that are new with these versions.

Overlays

A new overlay manager, the Microsoft Overlay Virtual Environment (MOVE), replaces the Microsoft Static Overlay Manager. For information on creating overlaid DOS programs using MOVE, see Chapter 15.

DOS Programs

You can now use a module-definition file when you link a DOS application. LINK creates a DOS executable file instead of a segmented executable file if the .DEF file contains any of the following:

- An **EXETYPE** statement that specifies the type **DOS**
- A **SEGMENTS** statement that specifies an overlay number

- A **FUNCTIONS** statement that specifies an overlay number

Other conditions also determine the type of executable file that LINK creates; for details, see “LINK Output Files” on page 563. The only valid statements in a .DEF file for a DOS program are **EXETYPE**, **SEGMENTS**, **FUNCTIONS**, and **INCLUDE**. All other statements are ignored.

Statements

New statements and changes to existing statements described in this chapter are:

- The **NAME** statement’s default *apptype* is now **WINDOWAPI** (formerly **NOTWINDOWCOMPAT**). (See page 611.)
- The **EXETYPE** statement’s default is now **WINDOWS**. (See page 615.)
- **EXETYPE WINDOWS** now assumes **PROTMODE** by default. (See page 616.)
- The **EXETYPE** statement has a new type argument, **DOS**. (See page 615.)
- The new **SECTIONS** and **OBJECTS** keywords are synonyms for the **SEGMENTS** statement. (See page 619.)
- The **SEGMENTS** statement accepts a new argument, **OVL:number**. This argument specifies the overlay in which the segment belongs in an overlaid DOS program. (See page 619.)
- The new **FUNCTIONS** statement specifies the order in which functions appear in the executable file. It can also assign functions to a specific segment. In an overlaid DOS program, **FUNCTIONS** can specify the overlay in which functions belong. (See page 625.)
- The new **INCLUDE** statement inserts module statements from a separate text file. (See page 627.)

16.2 Overview

A module-definition (.DEF) file is a text file that describes the name, attributes, exports, imports, system requirements, and other characteristics of an application or dynamic-link library (DLL). This file is required for DLLs and overlaid DOS programs. It is optional (but desirable) for other segmented executable files, such as Windows applications, and is usually not necessary for other DOS programs. For information on using .DEF files for overlays, see Chapter 15.

You use module-definition files in the following two situations:

- You can specify a module-definition file in LINK’s *deffile* field. The module-definition file gives the LINK utility the information that is necessary for link-

ing the program. For specific information on using a .DEF file when linking, see page 570.

- You can use the Microsoft Import Library Manager utility (IMPLIB) to create an import library from a module-definition file for a DLL (or from the DLL created by a module-definition file). You then specify the import library in LINK's *libraries* field when linking an application that uses functions and data in the DLL. For information on IMPLIB, see page 745.

Note The term “function” as used in this chapter refers to any routine for the programming language being used: function, procedure, or subroutine.

16.3 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 name, the attributes of program segments, and the number and names of exported and imported functions and data. Table 16.1 summarizes the purpose of the module statements and shows the order in which they are discussed in this chapter.

Table 16.1 Module Statements

Statement	Purpose
NAME	Names the application (no library created)
LIBRARY	Names the DLL (no application created)
DESCRIPTION	Embeds text in the application or DLL
STUB	Adds a DOS executable file to the beginning of the file
APPLoader	Replaces the default Windows loader with a custom loader
EXETYPE	Identifies the target operating system
PROTMODE	Specifies a protected-mode Windows program
REALMODE	Specifies a real-mode Windows program
STACKSIZE	Sets stack size in bytes
HEAPSIZE	Sets local heap size in bytes
CODE	Sets default attributes for all code segments
DATA	Sets default attributes for all data segments
SEGMENTS	Sets attributes for specific segments
OLD	Preserves ordinals from a previous DLL
EXPORTS	Defines exported functions
IMPORTS	Defines imported functions
FUNCTIONS	Specifies function order and location
INCLUDE	Inserts a file containing module statements

Syntax Rules

The syntax rules in this section apply to all statements in a module-definition file. Other rules specific to each statement are described in the sections that follow.

- Statement and attribute keywords are not case sensitive. User-specified identifiers are case insensitive by default; however, they can be made case sensitive by using LINK's (or IMPLIB's) /NOI option.
- Use one or more spaces, tabs, or newline characters to separate a statement keyword from its arguments and to separate statements from each other. A colon (:), or equal sign (=) that designates an argument is surrounded by zero or more spaces, tabs, or newline characters.
- A **NAME** or **LIBRARY** statement, if used, must precede all other statements.
- Most statements appear at most once in a file and accept one specification of parameters and attributes. The specification follows the statement keyword on the same or subsequent line(s). If the statement is repeated with a different specification later in the file, the later statement overrides the earlier one.
- The **INCLUDE** statement can appear more than once in the file. Each statement takes one filename specification.
- The **SEGMENTS**, **EXPORTS**, **IMPORTS**, and **FUNCTIONS** statements can appear more than once in the file. Each statement can take multiple specifications, which must be separated by one or more spaces, tabs, or newline characters. The statement keyword must appear once before the first specification and can be repeated before each additional specification.
- Comments in the file are designated by a semicolon (;) at the beginning of each comment line. A comment cannot share a line with part or all of a statement, but it can appear between lines of a multiline statement.
- Numeric arguments can be specified in decimal or in C-language notation.
- If a string argument matches a reserved word it must be enclosed in double quotation marks ("").

Example

The following module-definition file gives a description for a DLL. This sample file includes one comment and five statements.

```
; Sample module-definition file
LIBRARY FIRSTLIB WINDOWAPI
EXETYPE WINDOWS 3.0
CODE     PRELOAD MOVABLE DISCARDABLE
DATA     PRELOAD SINGLE
HEAPSIZE 1024
```

Reserved Words

The following words are reserved by the linker for use in module-definition files. These names can be used as arguments in module-definition statements only if the name is enclosed in double quotation marks ("").

ALIAS	INITGLOBAL	OVERLAY
APPLoader	INITINSTANCE	OVL
BASE	INVALID	PERMANENT
CLASS	IOPL	PHYSICAL
CODE	LIBRARY	PRELOAD
CONFORMING	LOADONCALL	PRIVATELIB
CONSTANT	LONGNAMES	PROTMODE
CONTIGUOUS	MACINTOSH	PURE
DATA	MAXVAL	READONLY
DESCRIPTION	MIXED1632	READWRITE
DEV386	MOVABLE	REALMODE
DEVICE	MOVEABLE	RESIDENT
DISCARDABLE	MULTIPLE	RESIDENTNAME
DOS	NAME	SECTIONS
DYNAMIC	NEWFILES	SEGMENTS
EXECUTE-ONLY	NODATA	SHARED
EXECUTEONLY	NOEXPANDDOWN	SINGLE
EXECUTEREAD	NOIOPL	STACKSIZE
EXETYPE	NONAME	STUB
EXPANDDOWN	NONCONFORMING	SUBSYSTEM
EXPORTS	NONDISCARDABLE	SWAPPABLE
FIXED	NONE	TERMINSTANCE
FUNCTIONS	NONPERMANENT	UNKNOWN
HEAPSIZE	NONSHARED	VERSION
HUGE	NOTWINDOWCOMPAT	VIRTUAL
IMPORTS	NULL	WINDOWAPI
IMPURE	OBJECTS	WINDOWCOMPAT
INCLUDE	OLD	WINDOWS

16.4 The NAME Statement

The **NAME** statement identifies the executable file as an application (rather than a DLL). It can also specify the name and application type. The **NAME** or **LIBRARY** statement must precede all other statements. If **NAME** is specified, the **LIBRARY** statement cannot be used. If neither is used, the default is **NAME**, and **LINK** creates an application.

Syntax

NAME [*appname*] [*apptype*] [**NEWFILES**]

Remarks

The fields can appear in any order.

If *appname* is specified, it becomes the name of the application as it is known by the operating system. This name can be any valid filename. If *appname* contains a space (allowed under some installable file systems), begins with a nonalphabetic character, or is a reserved word, enclose *appname* in double quotation marks. The name cannot exceed 255 characters (not including the surrounding quotation marks). If *appname* is not specified, the base name of the executable file becomes the name of the application.

If *apptype* is specified, it defines the type of application. This information is kept in the executable-file header. The *apptype* field can take one of the following values:

WINDOWAPI

The default. Creates a Windows application. The application uses the API provided by Windows and must be executed in the Windows environment. This is equivalent to the LINK option /PM:PM.

WINDOWCOMPAT

Creates a character-mode application to run in a text window in the Windows session. This is equivalent to the LINK option /PM:VIO.

NOTWINDOWCOMPAT

Creates a character-mode application that must run full screen and cannot run in a text window in Windows. This is equivalent to the LINK option /PM:NOVIO.

The **NEWFILES** keyword sets a bit in the file header to notify the loader that the application may be using an installable file system. The synonym **LONGNAMES** is supported for compatibility.

Example

The following example assigns the name `calendar` to an application that can run in a text window in Windows:

```
NAME calendar WINDOWCOMPAT
```

16.5 The LIBRARY Statement

The **LIBRARY** statement identifies the executable file as a DLL. It can also specify the .DLL filename. The **LIBRARY** or **NAME** statement must precede all other statements. If **LIBRARY** is specified, the **NAME** statement cannot be used. If neither is used, the default is **NAME**.

Syntax

LIBRARY [*libraryname*] [**PRIVATELIB**]

Remarks

The fields can appear in any order.

If *libraryname* is specified, it becomes the base name of the .DLL file. This name can be any valid filename. LINK assumes a .DLL extension whether or not an extension is specified. If *libraryname* contains a space (allowed under some installable file systems), begins with a nonalphabetic character, or is a reserved word, enclose it in double quotation marks (""). The name cannot exceed 255 characters.

The *libraryname* filename overrides a name specified in LINK's *exefile* field.

Specify **PRIVATELIB** to tell Windows that only one application may use the DLL.

Example

The following example assigns the name `calendar` to the DLL being defined.

```
LIBRARY calendar
```

16.6 The DESCRIPTION Statement

The **DESCRIPTION** statement inserts specified text into the application or DLL. This statement is useful for embedding source-control or copyright information into a file.

Syntax

DESCRIPTION '*text*'

Remarks

The *text* is a string of up to 255 characters enclosed in single or double quotation marks (' or "). To include a literal quotation mark in the text, either specify two consecutive quotation marks of the same type or enclose the text with the alternate type of quotation mark. If a **DESCRIPTION** statement is not specified, the default text is the name of the main output file as specified in LINK's *exefile* field.

You can view this string by using the EXEHDR utility. The string appears in the `Description:` field. For more information, see Chapter 17.

The **DESCRIPTION** statement is different from a comment. A comment is a line that begins with a semicolon (;). LINK does not place comments into the program.

Example

The following example inserts the text `Tester's Version, Test "A"`, which contains a literal single quotation mark and a pair of literal double quotation marks, into the application or DLL:

```
DESCRIPTION "Tester's Version, Test ""A"""
```

16.7 The STUB Statement

The **STUB** statement adds a DOS executable file to the beginning of a segmented executable file. The stub is invoked whenever the file is executed under DOS. Usually, the stub displays a message and terminates execution. However, the program can be of any size and may perform other actions. By default, LINK adds a standard stub for this purpose; the default message is different if the **EXETYPE WINDOWS** statement is used.

Use the STUB statement when creating a dual-mode program.

Syntax

```
STUB { 'filename' | NONE }
```

Remarks

The *filename* specifies the DOS executable file to be added. LINK searches for *filename* first in the current directory and then in directories specified with the PATH environment variable. If you specify a path with *filename*, LINK looks only in that location. The *filename* must be enclosed in single or double quotation marks (' or ").

The alternate specification **NONE** prevents LINK from adding a default stub. This saves space in the application or DLL. However, the resulting file will hang the system if loaded under DOS.

Example

The following example inserts the DOS executable file STOPIT.EXE at the beginning of the application or DLL:

```
STUB 'STOPIT.EXE'
```

The file STOPIT.EXE is executed when you attempt to run the application or DLL under DOS.

16.8 The APPLOADER Statement

The **APPLOADER** statement tells the linker to replace the default Windows loader with a custom loader. Use **APPLOADER** when you want your Windows program to be loaded by a different loader from the one Windows calls automatically at load time. This statement applies only to Windows programs.

Syntax

APPLOADER [[']]*loadername*[[']]

Remarks

The *loadername* is an identifier for an externally defined loader. The name is optionally enclosed in single or double quotation marks (' or "). The identifier is an external reference that must be resolved at link time in an object file or static library. It is not case sensitive unless the */NOI* option is used with the linker.

When **APPLOADER** appears in a module-definition file, LINK sets a bit in the header of the executable file to tell Windows that a custom loader is present. At load time, Windows loads only the first segment of the program and transfers control to that segment.

At link time, LINK creates a new logical segment called **LOADER_***loadername* and makes it the first physical segment of the program. LINK places the *loadername* function in this segment. Nothing else is contained in **LOADER_***loadername*; the */PACKC* option does not affect this segment.

Example

The following statement replaces the default loader with a loader called `__MSLANGLOAD`, which is defined in the Microsoft FORTRAN run-time libraries:

```
APPLOADER __MSLANGLOAD
```

Windows programs that use huge arrays will fail unless loaded by the custom loader provided in the default FORTRAN libraries. This statement appears in the default `.DEF` file used for FORTRAN QuickWin programs.

16.9 The EXETYPE Statement

The **EXETYPE** statement specifies under which operating system the program is to run. This statement provides an additional degree of protection against the program's being run under an incorrect operating system.

Syntax

EXETYPE [*descriptor*]

Remarks

The *descriptor* sets the target operating system. **EXETYPE** sets bits in the header which can be checked by operating systems. The *descriptor* field can accept one of the following values:

WINDOWS [*version*]

The default. Creates a Windows program. If a **STUB** statement is not specified, **WINDOWS** changes the default message to one that is the same as is provided in **WINSTUB.EXE**. The *version* is optional; for a description, see the next section, “Windows Programming.”

DOS

Creates a nonsegmented executable file. This statement is not required for an overlaid DOS program; **LINK** assumes **EXETYPE DOS**. For information on creating an overlaid program, see Chapter 15. For information on how **LINK** determines the type of executable file, see “LINK Output Files” on page 563.

UNKNOWN

Creates a segmented executable file but sets no bits in the header.

Windows Programming

The **WINDOWS** descriptor takes an optional version number. Windows reads this number to determine the minimum version of Windows needed to load the application or DLL. For example, if 3.0 is specified, the resulting application or DLL can run under Windows versions 3.0 and higher. If *version* is not specified, the default is 3.0. The syntax for *version* is:

```
number[.[number] ]
```

where each *number* is a decimal integer.

In Windows programming, use the **EXETYPE** statement with a **REALMODE** statement to specify an application or DLL that runs under either real-mode or protected-mode Windows.

Example

The following statement combination defines an application that runs under Windows 3.0 in any mode:

```
EXETYPE WINDOWS 3.0  
REALMODE
```

16.10 The **PROTMODE** Statement

The **PROTMODE** statement specifies that the application or DLL runs only under Windows in protected mode (either standard mode or 386 enhanced mode). **PROTMODE** lets **LINK** optimize to reduce both the size of the file on disk and

its loading time. **PROTMODE** is assumed by **EXETYPE WINDOWS**. To define a program that runs under any Windows mode, specify **REALMODE**.

Syntax**PROTMODE**

16.11 The REALMODE Statement

The **REALMODE** statement specifies that the application runs under Windows either in real mode or protected mode. By default, **EXETYPE WINDOWS** assumes **PROTMODE**.

Syntax**REALMODE**

16.12 The STACKSIZE Statement

The **STACKSIZE** statement specifies the size of the stack in bytes. It performs the same function as **LINK**'s **/STACK** option. If both of these are specified, the **STACKSIZE** statement overrides the **/STACK** option. Do not specify the **STACKSIZE** statement for a DLL.

Syntax**STACKSIZE** *number***Remarks**

The *number* must be a positive integer, in decimal or C-language notation, up to 64K–2. If an odd number is specified, **LINK** rounds up to the next even value.

Example

The following example allocates 4096 bytes of stack space:

```
STACKSIZE 4096
```

16.13 The HEAPSIZ Statement

The **HEAPSIZ** statement defines the size of the application or DLL's local heap in bytes. This value affects the size of the default data segment (**DGROUP**). The default without **HEAPSIZ** is that no local heap is created.

Syntax**HEAPSIZE** {*bytes* | **MAXVAL**}**Remarks**

The *bytes* field accepts a positive integer in decimal or C-language notation. The limit is **MAXVAL**; if *bytes* exceeds **MAXVAL**, the excess is not allocated.

MAXVAL is a keyword that sets the heap size to 64K minus the size of **DGROUP**.

Example

The following example sets the local heap to 4000 bytes:

```
HEAPSIZE 4000
```

16.14 The **CODE** Statement

The **CODE** statement defines the default attributes for all code segments within the application or DLL. The **SEGMENTS** statement can override this default for one or more specific segments.

Syntax**CODE** [[*attributes*]]**Remarks**

The *attributes* field accepts one or more optional attributes: *discard*, *executeonly*, *load*, *movable*, and *shared*. Each can appear once, in any order. These attributes are described in “CODE, DATA, and SEGMENTS Attributes” on page 620.

Example

The following example sets defaults for the program’s code segments:

```
CODE PRELOAD MOVABLE DISCARDABLE
```

16.15 The **DATA** Statement

The **DATA** statement defines the default attributes for all data segments within the application or DLL. The **SEGMENTS** statement can override this default for one or more specific segments.

Syntax**DATA** [[*attribute...*]]

Remarks

The *attributes* field accepts one or more optional attributes: *instance*, *load*, *movable*, *readonly*, and *shared*. Each can appear once, in any order. These attributes are described in “CODE, DATA, and SEGMENTS Attributes” on page 620. By default, all data segments have the following attributes:

```
SHARED LOADONCALL READWRITE FIXED
```

Example

The following example defines the application’s data segment so that it cannot be shared by multiple copies of the program and it cannot be written to. By default, the data segment can be read and written to, and a new DGROUP is created for each instance of the application.

```
DATA NONSHARED READONLY
```

16.16 The SEGMENTS Statement

The **SEGMENTS** statement defines the attributes of one or more individual segments in the application or DLL. The attributes specified for a specific segment override the defaults set in the **CODE** and **DATA** statements (except as noted). The total number of segment definitions cannot exceed the number set using **LINK**’s **/SEG** option. (The default without **/SEG** is 128.)

The **SEGMENTS** keyword marks the beginning of a section of segment definitions. Multiple definitions must be separated by one or more spaces, tabs, or new-line characters. The **SEGMENTS** statement must appear once before the first definition (on the same or preceding line) and can be repeated before each additional definition. **SEGMENTS** statements can appear more than once in the file.

Syntax**SEGMENTS**

```
[[ ' ]]segmentname[[ ' ]] [[CLASS 'classname' ]] [[attributes]]
```

or

```
[[ ' ]]segmentname[[ ' ]] [[CLASS 'classname' ]] [[OVL:overlaynumber]]
```

Remarks

Each segment definition begins with *segmentname* and is optionally enclosed in single or double quotation marks (' or "). The quotation marks are required if *segmentname* is a reserved word.

The **CLASS** keyword optionally specifies the class of the segment. Single or double quotation marks (' or ") are required around *classname*. If you do not use the **CLASS** argument, the linker assumes that the class is **CODE**.

The *attributes* field applies to segmented executable files. This field accepts one or more optional attributes: *discard*, *executeonly*, *load*, *movable*, *readonly*, and *shared*. Each can appear once, in any order. These attributes are described in the next section, "CODE, DATA, and SEGMENTS Attributes." LINK ignores *attributes* if **OVL** is specified.

The **OVL** keyword tells LINK to create a DOS program that contains overlays. If **OVL** is used, LINK assumes **EXETYPE DOS**. An alternate keyword is **OVERLAY**. The *overlaynumber* specifies the overlay in which the segment is to be placed. The value 0 represents the root, and positive decimal numbers through 65,535 represent overlays. By default, a segment is assigned to the root. For more information on overlays, see Chapter 15, "Creating Overlaid DOS Programs."

Example

The following example specifies segments named *cseg1*, *cseg2*, and *dseg*. The first segment is assigned the class *mycode*; the second is assigned **CODE** by default. Each segment is given different attributes.

```
SEGMENTS
  cseg1 CLASS 'mycode'
  cseg2          EXECUTEONLY PRELOAD
  dseg CLASS 'data'  LOADONCALL READONLY
```

16.17 CODE, DATA, and SEGMENTS Attributes

The following attribute fields apply to the **CODE**, **DATA**, and **SEGMENTS** statements described previously. Refer to "Remarks" in each of the previous sections for the attribute fields used by each statement. Most fields are used by all three statements; others are used as noted. Each field can appear once, in any order.

Listed with each attribute field are keywords that are legal values for the field. The fields and keywords are described, and the defaults are noted. If two segments with different attributes are combined into the same group, LINK makes decisions to resolve any conflicts and assumes a set of attributes.

discard

{**DISCARDABLE** | **NONDISCARDABLE**}

Used for **CODE** and **SEGMENTS** statements only. Determines whether a code segment can be discarded from memory to fill a different memory request. If the discarded segment is accessed later, it is reloaded from disk. The default is **NONDISCARDABLE**.

executeonly

{**EXECUTEONLY** | **EXECUTEREAD**}

Used for **CODE** and **SEGMENTS** statements only. Determines whether a code segment can be read as well as executed.

EXECUTEONLY specifies that the segment can only be executed. The keyword **EXECUTE-ONLY** is an alternate spelling.

EXECUTEREAD (the default) specifies that the segment is both executable and readable. This attribute is necessary for a program to run under the Microsoft CodeView debugger.

instance

{**NONE** | **SINGLE** | **MULTIPLE**}

Used for the **DATA** statement only. Affects the sharing attributes of the default data segment (DGROUP). This attribute interacts with the *shared* attribute.

NONE tells the loader not to allocate DGROUP. Use **NONE** when a DLL has no data and uses an application's DGROUP.

SINGLE (the default for DLLs) specifies that one DGROUP is shared by all instances of the DLL or application.

MULTIPLE (the default for applications) specifies that DGROUP is copied for each instance of the DLL or application.

load

{**PRELOAD** | **LOADONCALL**}

Used for **CODE**, **DATA**, and **SEGMENTS** statements. Determines when a segment is loaded.

PRELOAD specifies that the segment is loaded when the program starts.

LOADONCALL (the default) specifies that the segment is not loaded until accessed and only if not already loaded.

movable

{**MOVABLE** | **FIXED**}

Used for **CODE**, **DATA**, and **SEGMENTS** statements. Determines whether a segment can be moved in memory. This attribute is valid only for a Windows DLL or a real-mode Windows application. **FIXED** is the default. An alternative spelling for **MOVABLE** is **MOVEABLE**.

readonly

{**READONLY** | **READWRITE**}

Used for **DATA** and **SEGMENTS** statements only. Determines access rights to a data segment.

READONLY specifies that the segment can only be read.

READWRITE (the default) specifies that the segment is both readable and writeable.

shared

{**SHARED** | **NONSHARED**}

Used for real-mode Windows only. Determines whether all instances of the program can share **EXECUTEREAD** and **READWRITE** segments.

SHARED (the default for DLLs) specifies that one copy of the segment is loaded and shared among all processes that access the application or DLL. This attribute saves memory and can be used for code that is not self-modifying. An alternate keyword is **PURE**.

NONSHARED (the default for applications) specifies that the segment must be loaded separately for each process. An alternate keyword is **IMPURE**.

This attribute and the *instance* attribute interact for data segments. The *instance* attribute has the keywords **NONE**, **SINGLE**, and **MULTIPLE**. If **DATA SINGLE** is specified, **LINK** assumes **SHARED**; if **DATA MULTIPLE** is specified, **LINK** assumes **NONSHARED**. Similarly, **DATA SHARED** forces **SINGLE**, and **DATA NONSHARED** forces **MULTIPLE**.

16.18 The OLD Statement

The **OLD** statement directs the linker to search another DLL for export ordinals. This statement preserves ordinal values used from older versions of a DLL. For more information on ordinals, see the following sections on the **EXPORTS** and **IMPORTS** statements.

Exported names in the current DLL that match exported names in the old DLL are assigned ordinal values from the earlier DLL unless

- The name in the old module has no ordinal value assigned, or
- An ordinal value is explicitly assigned in the current DLL.

Only one DLL can be specified; ordinals can be preserved from only one DLL. If an export in the DLL was specified with the **NONAME** attribute, the exported name is not available and its ordinal cannot be preserved. The **OLD** statement has no effect on applications.

Syntax

OLD '*filename*'

Remarks

The *filename* specifies the DLL to be searched. It must be enclosed in single or double quotation marks (' or ').

16.19 The EXPORTS Statement

The **EXPORTS** statement defines the names and attributes of the functions and data made available to other applications and DLLs. It also defines the names and attributes of the functions that run with I/O privilege. By default, functions and data are hidden from other programs at run time. A definition is required for each function or data item being exported.

The **EXPORTS** keyword marks the beginning of a section of export definitions. Multiple definitions must be separated by one or more spaces, tabs, or newline characters. The **EXPORTS** keyword must appear once before the first definition (on the same or preceding line) and can be repeated before each additional definition. **EXPORTS** statements can appear more than once in the file.

Some languages offer a way to export without using an **EXPORTS** statement. For example, in C the `__export` keyword makes a function available from a DLL.

Syntax

EXPORTS

```
entryname[[=internalname]] [[@ord[[ nametable]] ]] [[NODATA]]
```

Remarks

The *entryname* defines the function or data-item name as it is known to other programs. If the function or data item is in a C++ module, the *entryname* must be specified as a decorated name. For specific information on decorated names, see Appendix B.

The optional *internalname* defines the actual name of the exported function or data item as it appears within the exporting program; by default, this name is the same as *entryname*.

The optional *ord* field defines a function's ordinal position within the module-definition table as an integer from 1 to 65,535. If *ord* is specified, the function can be called by either *entryname* or *ord*. The use of *ord* is faster and can save space.

The *nametable* is one of two optional keywords that determine what happens to *entryname*. By default, with or without *ord*, the *entryname* is placed in the nonresident names table. If the *ord* number is followed by **RESIDENTNAME**, the name is placed in the resident names table. If **NONAME** is specified after *ord*, the *entryname* is discarded from the DLL being created, and the item is exported only by ordinal.

The optional keyword **NODATA** specifies that there is no static data in the function.

Example

The following **EXPORTS** statement defines the three exported functions `SampleRead`, `StringIn`, and `CharTest`. The first two functions can be called either by their exported names or by an ordinal number. In the application or DLL where they are defined, these functions are named `read2bin` and `str1`, respectively.

```
EXPORTS
    SampleRead = read2bin @8
    StringIn   = str1     @4 RESIDENTNAME
    CharTest
```

16.20 The IMPORTS Statement

The **IMPORTS** statement defines the names and locations of functions and data items to be imported (usually from a DLL) for use in the application or DLL. A definition is required for each function or data item being imported. This statement is an alternative to resolving references through an import library created by the **IMPLIB** utility; functions and data items listed in an import library do not require an **IMPORTS** definition.

The **IMPORTS** keyword marks the beginning of a section of import definitions. Multiple definitions must be separated by one or more spaces, tabs, or newline characters. The **IMPORTS** keyword must appear once before the first definition on the same or preceding line and can be repeated before each additional definition. **IMPORTS** statements can appear more than once in the file.

Syntax

IMPORTS

```
[[internalname=]]modulename.entry
```

Remarks

The *internalname* specifies the function or data-item name as it is used in the importing application or DLL. Thus, *internalname* appears in the source code of the importing program, while the function may have a different name in the program where it is defined. By default, *internalname* is the same as the *entry* name. An *internalname* is required if *entry* is an ordinal value.

The *modulename* is the filename of the exporting application or DLL that contains the function or data item.

The *entry* field specifies the name or ordinal value of the function or data item as defined in the *modulename* application or DLL. If *entry* is an ordinal value, *internalname* must be specified. (Ordinal values are set in an **EXPORTS** statement.) If the function or data item is in a C++ module, *entry* must be specified as a decorated name. For information on decorated names, see Appendix B.

Note A given symbol (function or data item) has a name for each of three different contexts. The symbol has a name used by the exporting program (application or DLL) where it is defined, a name used as an entry point between programs, and a name used by the importing program where the symbol is used. If neither program uses the optional *internalname* field, the symbol has the same name in all three contexts. If either of the programs uses the *internalname* field, the symbol may have more than one distinct name.

Example

The **IMPORTS** statement that follows 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 `Sample`, `SampleA`, and `Read` applications or DLLs, respectively. The function from `Read` is referred to as `ReadChar` in the importing application or DLL. The original name of the function, as it is defined in `Read`, may or may not be known and is not included in the **IMPORTS** statement.

```
IMPORTS
        Sample.SampleRead
        SampleA.SampleWrite
ReadChar = Read.1
```

16.21 The **FUNCTIONS** Statement

The **FUNCTIONS** statement places functions in a specified physical order and assigns functions to segments or overlays. For more information on overlays, see Chapter 15, “Creating Overlaid DOS Programs.”

Syntax

```
FUNCTIONS[:{segmentname | overlaynumber}]
        functionname
```

Remarks

The **FUNCTIONS** keyword marks the beginning of a section of functions. **FUNCTIONS** statements can appear more than once in the `.DEF` file.

FUNCTIONS can be followed by a colon (`:`) and a destination specifier, which is either *segmentname* or *overlaynumber*.

The *segmentname* specifies a defined segment in which a function is to be placed. The *segmentname* does not have to be previously defined in a **SEGMENTS** statement. **LINK** assumes the segment definition, using the class **CODE**; a later **SEGMENTS** statement can redefine the segment.

The *overlaynumber* specifies the overlay in which a function is to be placed. Valid overlay numbers are from 0 through 65,535. The number 0 represents the root.

The *functionname* is the identifier for a “packaged function.” A packaged function is visible to the linker in the form of a COMDAT record. To compile a C function as a packaged function, use the /Gy option on the CL command line (or in PWB, choose Enable Function Level Linking in the Additional Global Options dialog box, which is available from the C or C++ Compiler Options dialog boxes.) Only packaged functions can be specified in a **FUNCTIONS** statement. You can specify one or more function names, separated by one or more spaces, tabs, or newline characters. If the function is in a C++ module, *functionname* must be specified as a decorated name. For specific information on decorated names, see Appendix B.

Ordering Functions

You can use **FUNCTIONS** to specify a list of ordered functions. LINK places ordered functions into a segment in the physical order that you specify before unordered functions in the same segment. You can let LINK choose the segment, or you can specify the segment. If LINK makes the decision, it places ordered functions in segments called COMDAT_SEG*n*, where *n* is one of a sequence of numbers beginning with 0. As LINK places ordered functions in these segments, it creates a new segment when the current one reaches 64K-36. You can specify the destination segment in one of two ways:

- Specify the segment using “explicit allocation.” In explicit allocation, a function is assigned to a segment at compile time, either in the source code or when compiling. In C source code, you can use the **__based** keyword (or its predecessor, the **alloc_text** pragma) to specify the segment where an individual function is to reside. When compiling with the CL compiler, you can use the /NT option to specify the segment where all functions in an object file are to reside. A function not explicitly allocated to a segment is sometimes referred to as an anonymous function.
- Specify the segment after the **FUNCTIONS** keyword. The segment must already have been defined, either in a **SEGMENTS** statement or at compile time. An explicitly allocated function cannot be placed in a different segment from the one to which it was allocated.

LINK accumulates multiple specifications and treats them as one list of ordered functions. If segments or overlays are specified, LINK accumulates the functions with other functions that have the same destination.

The following statement places three functions in a specified order within the segment called MySeg:

```
FUNCTIONS:MySeg
    Func1
    Func2
    Func3
```

Creating Overlays

You can use **FUNCTIONS** to place a packaged function in an overlay. By default, a function is assigned to the root.

If a function is explicitly allocated (see the previous section), it can be placed in an overlay only if its segment and any other functions in that segment are not also assigned to an overlay. In this case, the **FUNCTIONS** statement implicitly assigns the entire segment to the specified overlay. An explicitly allocated function cannot be placed in a different overlay from the segment to which it is allocated.

For examples of how to use the **FUNCTIONS** statement to create overlays, see “The **FUNCTIONS** and **SEGMENTS** Statements” in Chapter 15.

16.22 The **INCLUDE** Statement

The **INCLUDE** statement inserts the contents of a specified text file where it is specified in the .DEF file. The inserted file must contain module statements as they would appear in the .DEF file in which they are being inserted.

Syntax

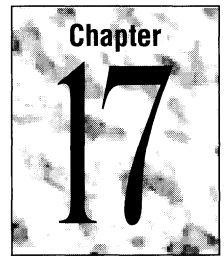
```
INCLUDE [[ ' ]]filename[[ ' ]]
```

Remarks

You can specify a path with the filename. Wildcards are not permitted. If *filename* contains a space (allowed under some installable systems), begins with a nonalphabetic character, or is a reserved word, enclose it in single or double quotation marks (' or ").

Multiple **INCLUDE** statements can appear in a .DEF file; each specifies a single insertion. **INCLUDE** statements can be nested up to 10 levels deep.

Using EXEHDR



The Microsoft EXE File Header Utility (EXEHDR) version 3.00 displays the contents of an executable-file header and can be used to alter some fields in the header. You can display or alter headers of DOS programs and segmented executable files (applications or DLLs). Some header fields have a different meaning in a Windows file; see your Windows documentation for more information. Examples of EXEHDR usage include:

- Determining whether a file is an application or a dynamic-link library (DLL).
- Viewing the attributes set by the module-definition file.
- Viewing the number and size of code and data segments.
- Setting a new stack allocation.

Many of the header fields contain information that was set in the module-definition file or as input options when the LINK utility created the file. This chapter assumes you are familiar with LINK and module-definition files. For information about LINK, see Chapter 14. For information about module-definition (.DEF) files, see Chapter 16. Many of the terms and keywords used in this section are explained in these chapters.

17.1 Running EXEHDR

This section describes the EXEHDR command line and the options available for controlling EXEHDR.

The EXEHDR Command Line

To run EXEHDR, use the following command line:

```
EXEHDR [[options]] filenames
```

The *options* field specifies options used to modify EXEHDR output or change the file header. Options are described in the next section.

The *filenames* field specifies one or more applications or DLLs. If you do not provide an extension, EXEHDR assumes .EXE. You can specify a path with the filename.

EXEHDR Options

Option names are not case sensitive and can be abbreviated to the shortest unique name. This section uses meaningful yet legal forms of the option names. Specify number arguments to options in decimal format or C-language notation. EXEHDR has the following options:

/HEA[[P]]:*number*

Sets the heap allocation field to *number* bytes. This option is only for segmented executable files.

/HEL[[P]]

Calls the QuickHelp utility. If EXEHDR cannot locate the Help file or QuickHelp, it displays a brief summary of EXEHDR command-line syntax.

/MA[[X]]:*number*

Sets the maximum memory allocation to *number* 16-byte paragraphs. The memory allocation fields tell DOS the maximum memory needed to load and run the program. The *number* must equal or exceed the minimum allocation. This option is equivalent to LINK's /CPARM option and applies only to DOS programs.

/MI[[N]]:*number*

Sets the minimum memory allocation to *number* 16-byte paragraphs. See the /MAX option for more information.

/NE[[WFILES]]

Sets a bit in the header to notify the loader that the program may be using an installable file system.

/NO[[LOGO]]

Suppresses the EXEHDR copyright message.

/P[[MTYPE]]:*type*

Sets the type of application. The *type* is one of the keywords for either LINK's /PM option or the NAME statement in a .DEF file. The keywords are **PM** (or **WINDOWAPI**), **VIO** (or **WINDOWCOMPAT**), and **NOVIO** (or **NOTWINDOWCOMPAT**).

/R[[ESETERROR]]

Clears the error bit in the header of a segmented executable file. LINK sets the error bit when it finds an unresolved reference or duplicate symbol definition. The operating system does not load a program if this bit is set. EXEHDR displays the message `Error in image` if it finds the error bit set. This option allows you to run a program that contains LINK errors and is useful during application development.

/S[[TACK]]:*number*

Sets the stack allocation to *number* bytes. The `/STACK` option is equivalent to LINK's `/STACK` option.

/V[[ERBOSE]]

Provides more information about segmented executable files. The additional information includes the default flags in the segment table, all run-time relocations, and additional fields from the header. For more information, see “EXEHDR Output: Verbose Output” on page 637.

/?

Displays a brief summary of EXEHDR command-line syntax.

17.2 Executable-File Format

DOS applications have a simple format, which consists of a single header followed by a relocation table and the load module. Segmented executable files have two headers. The first header, usually called the DOS header, has a simple format. The second header, sometimes called the new .EXE header, has a more detailed format. Figure 17.1 shows the arrangement of the headers in a segmented executable file. When the executable file runs under DOS, the operating system uses the old header to load the file. Otherwise, the system ignores the DOS (or “old”) header and uses the new header.

The listing generated by EXEHDR shows the contents of the file header and information about each segment in the file. The type of listing generated reflects the structure of the header for the kind of file being checked. (For more information about the structure of DOS applications and segmented executable files, see the *MS-DOS Encyclopedia*.)

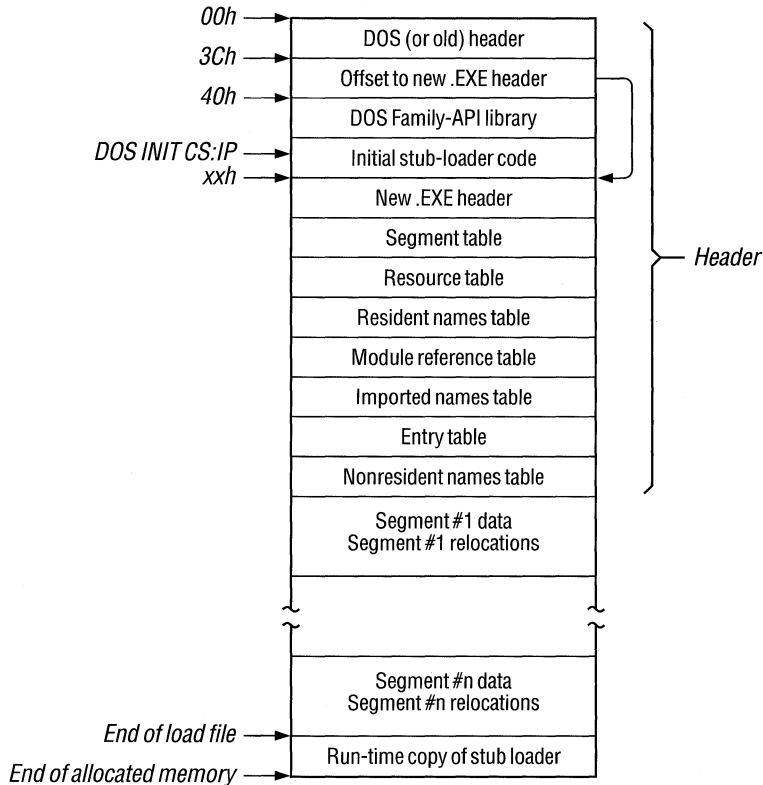


Figure 17.6 Format for a Segmented Executable File

17.3 EXEHDR Output: DOS Executable File

The EXEHDR output for a DOS executable file appears as follows:

```
.EXE size (bytes)
Packed .EXE file
Magic number:
Bytes on last page:
Pages in file:
Relocations:
Paragraphs in header:
Extra paragraphs needed:
Extra paragraphs wanted:
Initial stack location:
Word checksum:
Entry point:
Relocation table address:
Memory needed:
```

The meaning of each field is described in the following list:

.EXE size (bytes)

 Gives the size of the file on disk.

Packed .EXE file

 Is displayed only if the file is packed.

Magic number:

 Tells the operating-system loader the format of the header.

Bytes on last page:

 Tells the loader how much data exists in the last page on disk.

Pages in file:

 Gives the number of whole 512-byte pages in the file on disk. If the program contains overlays, this field shows the number of pages in the root.

Relocations:

 Tells the loader the number of entries in the relocation table.

Paragraphs in header:

 Gives the size of the header in 16-byte paragraphs. This represents the offset of the load image within the file.

Extra paragraphs needed:

 Tells the loader the required minimum number of paragraphs of memory in addition to the image size. The image size is equal to Bytes on last page + (Pages in file x 512).

Extra paragraphs wanted:

 Tells the loader the number of paragraphs of memory above the size on disk requested for loading the program. This value is set by LINK's /CPARM option.

Initial stack location:

 Gives the address (SS:SP) of the DOS program's stack.

Word checksum:

 Confirms for the loader that the file is a valid executable file.

Entry point:

 Gives the start address.

Relocation table address:

 Gives the location of the table of relocation pointers as an offset from the beginning of the file.

Memory needed:

 Tells the loader the total memory needed to load the application. The value in this field is equal to (Extra paragraphs needed x 16) + .EXE size (bytes).

17.4 EXEHDR Output: Segmented Executable File

The first part of the EXEHDR output for a segmented executable file appears as follows:

```
Module:  
Description:  
Data:  
Initial CS:IP:  
Initial SS:SP:  
Extra stack allocation:  
DGROUP:
```

The meaning of each field is described in the following list:

Module:

Gives the name of the application as specified in the **NAME** statement of the .DEF file used to create the file or the name assumed by default.

Description:

Gives the text of the **DESCRIPTION** statement of the .DEF file or the description assumed by default.

Data:

Indicates the program's default data segment (DGROUP) type: **SHARED**, **NONSHARED**, or **NONE**. This type can be specified in a .DEF file.

Initial CS:IP:

Gives the application's starting address.

Initial SS:SP:

Gives the value of the initial stack pointer, which gives the location of the initial stack.

Extra stack allocation:

Gives the size in bytes of the stack, specified in hexadecimal.

DGROUP:

Gives the segment number of DGROUP in the program's object files. Segment numbers start with the number 1.

At the end of the list of fields, EXEHDR displays any module flags that were set for every segment in the module. For example, `PROTMODE` may appear.

The message `Error in image` may appear at the end of the list of fields. If a **LINK** error (such as "unresolved external") occurs when the file is created, **LINK** sets the error bit in the header. This prevents the file from being loaded. You can clear the error bit with the `/RESET` option, described on page 631.

DLL Header Differences

For a DLL, the output differs slightly and appears as

```
Library:
Description:
Data:
Initialization:
Initial CS:IP:
Initial SS:SP:
DGROUP:
```

The meaning of each field is described in the following list:

Library:

Gives the name of the library as specified in the **LIBRARY** statement in the module-definition file (or the default name).

Description:

Data:

Same as for other segmented executable files.

Initialization:

Gives the type of initialization as specified in the **LIBRARY** statement in the module-definition file (or the default initialization).

Initial CS:IP:

Gives the address of the initialization routine. If the DLL has no initialization routine, the start address is zero.

Initial SS:SP:

May be zero for a DLL.

DGROUP:

May not appear for a DLL.

Segment Table

After the header fields for a segmented executable file, EXEHDR displays the segment table. All values appear in hexadecimal except for the segment index number. An example of this table is:

```
no. type address file mem flags
 1 CODE 00000400 00efb 00efb
 2 DATA 00001400 00031 0007d
 3 DATA 00001600 0003c 00040 SHARED
```


The following list describes each heading in the segment table:

no.

Segment index number (in decimal), starting with 1.

type

Identification of the segment as a code or data segment.

address

A seek offset for the segment within the file.

file

Size in bytes of the segment in the file on disk.

mem

Size in bytes of the segment in memory. If `mem` is greater than `file`, the operating system pads the extra space with zero values at load time.

flags

Segment attributes. If the `/V` option is not used, only nondefault attributes are listed. Attributes that are meaningful only to Windows are displayed in lower-case and in parentheses.

Exports Table

Following the segment table, EXEHDR displays a table of exports if they exist. An example of this table is:

Exports:

ord	seg	offset	name
1	3	0000	HELPWNDPROC exported
19	3	032e	ICONWNDPROC exported
21	35	0000	PATHWNDPROC exported
5	30	0264	ANNOUPDATEDLG exported
8	33	0000	BOOKMARKDLG exported

The following list describes each heading in the Exports table:

ord

The ordinal number as specified in the `@ord` field in an **EXPORTS** statement in a module-definition file. If `ord` was not specified, this column entry is blank.

seg

The index of the segment where the exported name is defined.

offset

The offset in the segment where the exported name is defined.

name

The exported name of the routine plus all flags applied to the exported routine, as specified in the **EXPORTS** statement in the module-definition file.

17.5 EXEHDR Output: Verbose Output

The `/V` option provides more extensive information about a segmented executable file. The verbose output more closely reflects the file's header structure. (For an illustration of this structure, see Figure 17.1, earlier in this chapter.)

DOS Header Information

EXEHDR begins by displaying the DOS fields described on page 632, with the addition of two fields:

Reserved words:

Displays the contents of space in a DOS header that is normally unused.

New .EXE header address:

Holds the starting location of the part of the header describing the segmented executable file.

New .EXE Header Information

EXEHDR then displays the header fields for the segmented executable file. In addition to the default fields described on page 634, the verbose output includes many other fields.

A field called `Operating system:` follows the `Description:` field. This field tells the system under which the program is to run.

The following fields are then displayed:

Linker version:
32-bit Checksum:
Segment Table:
Resource Table:
Resident Names Table:
Module Reference Table:
Imported Names Table:
Entry Table:
Non-resident Names Table:
Movable entry points:
Segment sector size:
Heap allocation:
Application type:
Other module flags:

The meaning of each field is described in the following list:

Linker version:

Tells which version of LINK was used to create the segmented executable file.

32-bit Checksum:

Confirms for the loader that the file is a valid executable file. (See the `word checksum: field` for DOS executable files.)

Segment Table:

Resource Table:

Resident Names Table:

Module Reference Table:

Imported Names Table:

Entry Table:

Non-resident Names Table:

Describe various tables in the segmented executable file. Each description gives the table name, its address within the file, and its length in hexadecimal and in decimal.

Movable entry points:

Gives the number of entries to segments that have the **MOVABLE** attribute. This field is used only by Windows.

Segment sector size:

Gives the alignment set by the `/ALIGN` option or the default of 512. This field equals the sector size on disk.

Heap allocation:

Gives the size of the heap. This field is displayed only if a **HEAPSIZE** statement appeared in the module-definition file.

Application type:

Gives the type as specified in the **PMTYPE** statement of the module-definition file used to create the file being examined, or as specified with LINK's `/PM` option, or assumed by default. For a DLL, a 0 is always displayed.

Other module flags:

Gives other attributes of the file; if none, this field is not displayed.

At the end of the list of fields, EXEHDR displays any module flags that were set for every segment in the module. For example, `PROTMODE` may appear.

Tables

Following the header fields, EXEHDR displays the segment table with complete attributes, not just the nondefault attributes. Attributes that are meaningful only to Windows are displayed in lowercase and in parentheses. In addition to the attributes specified in the module-definition file (described in “CODE, DATA, and SEGMENTS Attributes” on page 620) or assumed by default, the verbose output includes the following two 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.

EXEHDR then displays the Exports table if exports exist.

Relocations

Following the tables, EXEHDR displays descriptions of relocations. Each has a heading in the following form:

```

1 type  offset target
  BASE  eff4  seg   1 offset 0000
  BASE  f204  seg   2 offset 0000
  OFFSET eff1  seg   1 offset e968
  OFFSET 314e  seg   1 offset 32ea
  BASE  c0f1  seg   3 offset 0000
  OFFSET d397  seg   1 offset cf70
  PTR   cd3e  imp  DOSCALLS.137
  OFFSET b1a8  seg   1 offset ae7c
  PTR   f57c  imp  KBDCALLS.13

```

The following list describes each heading:

number

The segment number, as given earlier in the segments table.

type

Relocation type, which gives the kind of address information requested.

offset

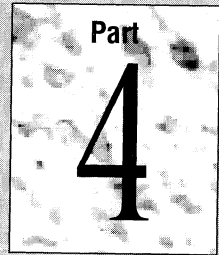
The location of the requested address change in the source segment.

target

The requested relocation address.

Each relocation table ends by stating the total number of relocations.

Utilities



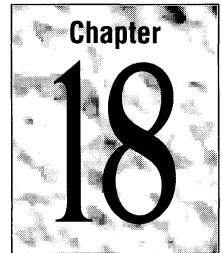
Chapter 18	Managing Projects with NMAKE.....	645
19	Managing Libraries with LIB.....	697
20	Creating Help Files with HELPMAKE.....	709
21	Browser Utilities.....	731
22	Using Other Utilities.....	743

Utilities

Microsoft C/C++ provides other tools for you to use in of building your program.

Chapter 18 describes how to use NMAKE to automate your project. Chapter 19 tells how to use LIB to manage standard libraries. Chapter 20 explains how to create and maintain help files with HELPMAKE. Chapter 21 discusses the tools for creating a browser database. Chapter 22 describes how to use other special purpose utilities.

Managing Projects with NMAKE



This chapter describes the Microsoft Program Maintenance Utility (NMAKE) version 1.20. NMAKE is a sophisticated command processor that saves time and simplifies project management. Once you specify which project files depend on others, NMAKE automatically builds your project without recompiling files that haven't changed since the last build.

If you are using the Microsoft Programmer's WorkBench (PWB) to build your project, PWB automatically creates a makefile and calls NMAKE to run the file. You may want to read this chapter if you intend to build your program outside of PWB, if you want to understand or modify a makefile created by PWB, or if you want to use a foreign makefile in PWB.

NMAKE can swap itself to expanded memory, extended memory, or disk to leave room for the commands it spawns. (For more information, see the description of the `/M` option on page 649.)

18.1 New Features

This version of NMAKE offers the following new features:

- New options: `/B`, `/K`, `/M`, `/V` (see pages 648–650)
- Addition of `.CPP` and `.CXX` to the `.SUFFIXES` list (see page 688)
- Predefined macros for C++ programs: `CPP`, `CXX`, `CPPFLAGS`, `CXXFLAGS` (see pages 676)
- Predefined inference rules for C++ programs (see page 684)
- The `!MESSAGE` directive (see page 689)
- Two preprocessing operators: `DEFINED`, `EXIST` (see page 691)
- Three keywords for use with the `!ELSE` directive: `IF`, `IFDEF`, `IFNDEF` (see page 688)
- New directives: `!ELSEIF`, `!ELSEIFDEF`, `!ELSEIFNDEF` (see page 688)

18.2 Overview

NMAKE works by looking at the “time stamp” of a file. A time stamp is the time and date the file was last modified. Time stamps are assigned by most operating systems in 2-second intervals. NMAKE compares the time stamps of a “target” file and its “dependent” files. A target is usually a file you want to create, such as an executable file, though it could be a label for a set of commands you wish to execute. A dependent is usually a file from which a target is created, such as a source file. A target is “out-of-date” if any of its dependents has a later time stamp than the target or if the target does not exist. (For information on how the 2-second interval affects your build, see the description of the /B option on page 648.)

Warning For NMAKE to work properly, the date and time setting on your system must be consistent relative to previous settings. If you set the date and time each time you start the system, be careful to set it accurately. If your system stores a setting, be certain that the battery is working.

NMAKE follows the instructions you specify in a makefile.

When you run NMAKE, it reads a “makefile” that you supply. A makefile (sometimes called a description file) is a text file containing a set of instructions that NMAKE uses to build your project. The instructions consist of description blocks, macros, directives, and inference rules. Each description block typically lists a target (or targets), the target’s dependents, and the commands that build the target. NMAKE compares the time stamp on the target file with the time stamp on the dependent files. If the time stamp of any dependent is the same as or later than the time stamp of the target, NMAKE updates the target by executing the commands listed in the description block.

It is possible to run NMAKE without a makefile. In this case, NMAKE uses predefined macros and inference rules along with instructions given on the command line or in TOOLS.INI.

NMAKE’s main purpose is to help you build programs quickly and easily. However, it is not limited to compiling and linking; NMAKE can run other types of programs and can execute operating system commands. You can use NMAKE to prepare backups, move files, and perform other project-management tasks that you ordinarily do at the operating-system prompt.

This chapter uses the term “build,” as in building a target, to mean evaluating the time stamps of a target and its dependent and, if the target is out-of-date, executing the commands associated with the target. The term “build” has this meaning whether or not the commands actually create or change the target file.

18.3 Running NMAKE

You invoke NMAKE with the following syntax:

```
NMAKE [[options]] [[macros]] [[targets]]
```

The *options* field lists NMAKE options, which are described in the following section, “Command-Line Options.”

The *macros* field lists macro definitions, which allow you to change text in the makefile. The syntax for macros is described in “User-Defined Macros” on page 668.

The *targets* field lists targets to build. NMAKE builds only the targets listed on the command line. If you don’t specify a target, NMAKE builds only the first target in the first dependency in the makefile. (You can use a pseudotarget to tell NMAKE to build more than one target. See “Pseudotargets” on page 658.)

NMAKE uses the following priorities to determine how to conduct the build:

1. If the /F option is used, NMAKE searches the current or specified directory for the specified makefile. NMAKE halts and displays an error message if the file does not exist.
2. If you do not use the /F option, NMAKE searches the current directory for a file named MAKEFILE.
3. If MAKEFILE does not exist, NMAKE checks the command line for target files and tries to build them using inference rules (either defined in TOOLS.INI or predefined). This feature lets you use NMAKE without a makefile as long as NMAKE has an inference rule for the target.
4. If a makefile is not used and the command line does not specify a target, NMAKE halts and displays an error message.

Example

The following command specifies an option (/S) and a macro definition ("program=sample") and tells NMAKE to build two targets (sort.exe and search.exe). The command does not specify a makefile, so NMAKE looks for MAKEFILE or uses inference rules.

```
NMAKE /S "program=sample" sort.exe search.exe
```

Command-Line Options

NMAKE accepts options for controlling the NMAKE session. Options are not case sensitive and can be preceded by either a slash (/) or a dash (-).

You can specify some options in the makefile or in `TOOLS.INI`. For information on `TOOLS.INI`, see page 652. For information on specifying options with the `!CMDSWITCHES` directive, see page 688.

`/A`

Forces NMAKE to build all evaluated targets, even if the targets are not out-of-date with respect to their dependents. This option does not force NMAKE to build unrelated targets.

`/B`

Tells NMAKE to execute a dependency even if time stamps are equal. Most operating systems assign time stamps with a resolution of 2 seconds. If your commands execute quickly, NMAKE may conclude that a file is up-to-date when in fact it is not. This option may result in some unnecessary build steps but is recommended when running NMAKE on very fast systems.

`/C`

Suppresses default NMAKE output, including nonfatal NMAKE error or warning messages, time stamps, and the NMAKE copyright message. If both `/C` and `/K` are specified, `/C` suppresses the warnings issued by `/K`.

`/D`

Displays information during the NMAKE session. The information is interspersed in NMAKE's default output to the screen. NMAKE displays the time stamp of each target and dependent evaluated in the build and issues a message when a target does not exist. Dependents for a target precede the target and are indented. The `/D` and `/P` options are useful for debugging a makefile.

To set or clear `/D` for part of a makefile, use the `!CMDSWITCHES` directive; see "Preprocessing Directives" on page 688.

`/E`

Causes environment variables to override macro definitions in the makefile. See "Macros" on page 667.

`/F filename`

Specifies *filename* as the name of the makefile. Zero or more spaces or tabs precede *filename*. If you supply a dash (`-`) instead of a filename, NMAKE gets makefile input from the standard input device. (End keyboard input with either `F6` or `CTRL+Z`.) NMAKE accepts more than one makefile; use a separate `/F` specification for each makefile input.

If you omit `/F`, NMAKE searches the current directory for a file called `MAKEFILE` (with no extension) and uses it as the makefile. If `MAKEFILE` doesn't exist, NMAKE uses inference rules for the command-line targets.

`/HELP`

Calls the QuickHelp utility. If NMAKE cannot locate the Help file or QuickHelp, it displays a brief summary of NMAKE command-line syntax.

/I

Ignores exit codes from all commands listed in the makefile. NMAKE processes the whole makefile even if errors occur. To ignore exit codes for part of a makefile, use the dash (-) command modifier or the **.IGNORE** directive; see “Command Modifiers” on page 661 and “Dot Directives” on page 687. To set or clear /I for part of a makefile, use the **!CMDSWITCHES** directive; see “Preprocessing Directives” on page 688. To ignore errors for unrelated parts of the build, use the /K option; /I overrides /K if both are specified.

/K

Continues the build for unrelated parts of the dependency tree if a command terminates with an error. By default, NMAKE halts if any command returns a non-zero exit code. If this option is specified and a command returns a nonzero exit code, NMAKE ceases to execute the block containing the command. It does not attempt to build the targets that depend on the results of that command; instead, it issues a warning and builds other targets. When /K is specified and the build is not complete, NMAKE returns an exit code of 1. This differs from the /I option, which ignores exit codes entirely; /I overrides /K if both are specified. The /C option suppresses the warnings issued by /K.

/M

Swaps NMAKE to disk to conserve extended or expanded memory under DOS. By default, NMAKE leaves room for commands to be executed in low memory by swapping itself to extended memory (if enough space exists there) or to expanded memory (if there is not sufficient extended memory but there is enough expanded memory) or to disk. The /M option tells NMAKE to ignore any extended or expanded memory and to swap only to disk.

/N

Displays but does not execute the commands that would be executed by the build. Preprocessing commands are executed. This option is useful for debugging makefiles and checking which targets are out-of-date. To set or clear /N for part of a makefile, use the **!CMDSWITCHES** directive; see “Preprocessing Directives” on page 688.

/NOLOGO

Suppresses the NMAKE copyright message.

/P

Displays NMAKE information to the standard output device, including all macro definitions, inference rules, target descriptions, and the **.SUFFIXES** list, before running the NMAKE session. If /P is specified without a makefile or command-line target, NMAKE displays the information and does not issue an error. The /P and /D options are useful for debugging a makefile.

/Q

Checks time stamps of targets that would be updated by the build but does not run the build. NMAKE returns a zero exit code if the targets are up-to-date and a nonzero exit code if any target is out-of-date. Only preprocessing commands

in the makefile are executed. This option is useful when running NMAKE from a batch file.

/R

Clears the **.SUFFIXES** list and ignores inference rules and macros that are defined in the **TOOLS.INI** file or that are predefined.

/S

Suppresses the display of all executed commands. To suppress the display of commands in part of a makefile, use the **@** command modifier or the **.SILENT** directive; see “Command Modifiers” on page 661 and “Dot Directives” on page 687. To set or clear **/S** for part of a makefile, use the **!CMDSWITCHES** directive; see “Preprocessing Directives” on page 688.

/T

Changes time stamps of command-line targets (or first target in the makefile if no command-line targets are specified) to the current time and executes preprocessing commands but does not run the build. Contents of target files are not modified.

/V

Causes all macros to be inherited when recursing. By default, only macros defined on the command line and environment-variable macros are inherited when NMAKE is called recursively. This option makes all macros available to the recursively called NMAKE session. See “Inherited Macros” on page 679.

/X filename

Sends all error output (from both NMAKE and the executed commands) to *filename*, which can be a file or a device. Zero or more spaces or tabs precede *filename*. If you supply a dash (–) instead of a filename, error output is sent to the standard output device. By default, NMAKE sends errors to standard error.

/?

Displays a brief summary of NMAKE command-line syntax and exits to the operating system.

Example

The following command line specifies two NMAKE options:

```
NMAKE /F sample.mak /C targ1 targ2
```

The **/F** option tells NMAKE to read the makefile **SAMPLE.MAK**. The **/C** option tells NMAKE not to display nonfatal error messages and warnings. The command specifies two targets (**targ1** and **targ2**) to update.

NMAKE Command File

You can place a sequence of command-line arguments in a text file and pass the file’s name as a command-line argument to NMAKE. NMAKE opens the command file and reads the arguments. You can use a command file to overcome the

limit on the length of a command line in the operating system (128 characters in DOS).

To provide input to NMAKE with a command file, type

```
NMAKE @commandfile
```

The *commandfile* is the name of a text file containing the information NMAKE expects on the command line. Precede the name of the command file with an at sign (@). You can specify a path with the filename.

NMAKE treats the file as if it were a single set of arguments. It replaces each line break with a space. Macro definitions that contain spaces must be enclosed in quotation marks; see “Where to Define Macros” on page 669.

You can split input between the command line and a command file. Specify *@commandfile* on the command line at the place where the file’s information is expected. Command-line input can precede and/or follow the command file. You can specify more than one command file.

Example 1

If a file named UPDATE contains the line

```
/S "program = sample" sort.exe search.exe
```

you can start NMAKE with the command

```
NMAKE @update
```

The effect is the same as if you entered the following command line:

```
NMAKE /S "program = sample" sort.exe search.exe
```

Example 2

The following is another version of the UPDATE file:

```
/S "program \  
= sample" sort.exe search.exe
```

The backslash (\) allows the macro definition to span two lines.

Example 3

If the command file called UPDATE contains the line

```
/S "program = sample" sort.exe
```

you can start NMAKE with the command

```
NMAKE @update search.exe
```

The TOOLS.INI File

You can customize NMAKE by placing commonly used information in the TOOLS.INI initialization file. Settings for NMAKE must follow a line that begins with the tag `[NMAKE]`. The tag is not case sensitive. This section of the initialization file can contain any makefile information. NMAKE uses this information in every session, unless you run NMAKE with the `/R` option. NMAKE looks for TOOLS.INI first in the current directory and then in the directory specified by the `INIT` environment variable.

You can use the `!CMDSWITCHES` directive to specify command-line options in TOOLS.INI. An option specified this way is in effect for every NMAKE session. This serves the same purpose as does an environment variable, which is a feature available in other utilities. For more information on `!CMDSWITCHES`, see page 688.

Macros and inference rules appearing in TOOLS.INI can be overridden. See “Precedence among Macro Definitions” on page 680 and “Precedence among Inference Rules” on page 686.

NMAKE reads information in TOOLS.INI before it reads makefile information. Thus, for example, a description block appearing in TOOLS.INI acts as the first description block in the makefile; this is true for every NMAKE session, unless `/R` is specified.

To place a comment in TOOLS.INI, specify the comment on a separate line beginning with a semicolon (`;`). You can also specify comments with a number sign (`#`) as you can in a makefile; for more information, see “Comments” on page 654.

Example

The following is an example of text in a TOOLS.INI file:

```
[NMAKE]
; macros
CC      = qc1
CFLAGS = /Gc /Gs /W3 /Oat
; inference rule
.c.obj:
    $(CC) /Zi /c $(CFLAGS) $*.c
```

NMAKE reads and applies the lines following `[NMAKE]`. The example redefines the macro `CC` to invoke the Microsoft QuickC Compiler, defines the macro `CFLAGS`, and redefines the inference rule for making `.OBJ` files from `.C` sources. These NMAKE features are explained throughout this chapter.

18.4 Contents of a Makefile

An NMAKE makefile contains description blocks, macros, inference rules, and directives. This section presents general information about writing makefiles. The rest of the chapter describes each of the elements of makefiles in detail.

Using Special Characters as Literals

You may need to specify a literal character one of the characters that NMAKE uses for a special purpose. These characters are:

: ; # () \$ ^ \ { } ! @ -

To use one of these characters without its special meaning, place a caret (^) in front of it. NMAKE ignores carets that precede characters other than the special characters listed previously. A caret within a quoted string is treated as a literal caret character.

You can also use a caret at the end of a line to insert a literal newline character in a string or macro. The caret tells NMAKE to interpret the newline character as part of the macro, not a line break. Note that this effect differs from using a backslash (\) to continue a line in a macro definition. A newline character that follows a backslash is replaced with a space. For more information, see “User-Defined Macros” on page 668.

In a command, a percent symbol (%) can represent the beginning of a file specifier. (See “Filename-Parts Syntax” on page 663.) NMAKE interprets %s as a filename, and it interprets the character sequence of %l followed by d, e, f, p, or F as part or all of a filename or path. If you need to represent these characters literally in a command, specify a double percent sign (%%) in place of a single one. In all other situations, NMAKE interprets a single % literally. However, NMAKE always interprets a double %% as a single %. Therefore, to represent a literal %%, you can specify either three percent signs, %%%, or four percent signs, %%%%.

To use the dollar sign (\$) as a literal character in a command, you must specify two dollar signs (\$\$); this method can also be used in other situations where ^\$ also works.

For information on literal characters in macro definitions, see “Special Characters in Macros” on page 668.

Wildcards

You can use DOS wildcards (* and ?) to specify target and dependent names. NMAKE expands wildcards that appear on dependency lines. A wildcard specified in a command is passed to the command; NMAKE does not expand it.

Example

In the following description block, the wildcard `*` is used twice:

```
project.exe : *.c
    cl *.c /Foproject.exe
```

NMAKE expands the `*.c` in the dependency line and looks at all files having the `.C` extension in the current directory. If any `.C` file is out-of-date, the CL command expands the `*.c` and compiles and links all files.

Comments

To place a comment in a makefile, precede it with a number sign (`#`). NMAKE ignores all text from the number sign to the next newline character. You can use comments in the following situations:

```
# Comment on line by itself

OPTIONS = /MAP # Comment on macro definition line

all.exe : one.obj two.obj # Comment on dependency line
    link one.obj two.obj;
# Comment in commands block
    copy one.exe \release

.obj.exe: # Comment in inference rule
```

Command lines cannot contain comments; this is true even for a command that is specified on the same line as a dependency line or inference rule. However, a comment can appear between lines in a commands block; the `#` must appear at the beginning of the line.

To specify a literal `#`, precede it with a caret (`^`), as in the following:

```
DEF = ^#define #Macro representing a C preprocessing directive
```

Comments can also appear in a TOOLS.INI file. TOOLS.INI allows an additional form of comment using a semicolon (`;`). See “The TOOLS.INI File” on page 652.

Long Filenames

You can use long filenames if they are supported by your file system. However, you must enclose the name in double quotation marks, as in the following dependency line:

```
all : "VeryLongFileName.exe"
```

18.5 Description Blocks

Description blocks form the heart of the makefile. Figure 18.1 illustrates a typical NMAKE description block. The following sections discuss dependencies, targets, and dependents. The contents of a commands block are described in “Commands” on page 660.

```

Targets          Dependents
-----
myapp.exe : myapp.obj another.obj myapp.def } Dependency line
  link myapp another, , NUL, mylib, myapp }
  copy myapp.exe c:\project } Commands

```

Figure 18.1 NMAKE Description Block

Dependency Line

A description block begins with a “dependency line.” A dependency line specifies one or more “target” files and then lists zero or more “dependent” files. If a target does not exist, or if its time stamp is earlier than that of any dependent, NMAKE executes the commands block for that target. Figure 18.1 illustrates a dependency line.

The dependency line must not be indented (it cannot start with a space or tab). The first target must be specified at the beginning of the line. Targets are separated from dependents by a single colon, except as described in “Using Targets in Multiple Description Blocks” on page 656. The colon can be preceded or followed by zero or more spaces or tabs. The entire dependency must appear on one line; however, you can extend the line by placing a backslash (\) after a target or dependent and continuing the dependency on the next line.

Before executing any commands, NMAKE moves through all dependencies and applicable inference rules to build a “dependency tree” that specifies all the steps required to fully update the target. NMAKE checks to see if dependents themselves are targets in other dependency lists, if any dependents in those lists are targets elsewhere, and so on. After it builds the dependency tree, NMAKE checks time stamps. If it finds any dependents in the tree that are newer than the target, NMAKE builds the target.

The dependency line in Figure 18.1 tells NMAKE to rebuild the MYAPP.EXE target whenever MYAPP.OBJ, ANOTHER.OBJ, or MYAPP.DEF has changed more recently than MYAPP.EXE.

Targets

The targets section of the dependency line lists one or more target names. At least one target must be specified. Separate multiple target names with one or more spaces or tabs. You can specify a path with the filename. Targets are not case sensitive. A target (including path) cannot exceed 256 characters.

If the name of the last target before the colon (:) is a single character, you must put a space between the name and the colon; otherwise, NMAKE interprets the letter-colon combination as a drive specifier.

Usually a target is the name of a file to be built using the commands in the description block. However, a target can be any valid filename, or it can be a pseudotarget. (For more information, see “Pseudotargets” on page 658.)

NMAKE builds targets specified on the NMAKE command line. If a command-line target is not specified, NMAKE builds the first target in the first dependency in the makefile.

The example in Figure 18.1 tells NMAKE how to build a single target file called MYAPP.EXE if it is missing or out-of-date.

Using Targets in Multiple Description Blocks

A target can appear in only one description block when specified as shown in Figure 18.1. To update a target using more than one description block, specify two consecutive colons (::) between targets and dependents. One use for this feature is for building a complex target that contains components created with different commands.

Example

The following makefile updates a library:

```
target.lib :: one.asm two.asm three.asm
    ML one.asm two.asm three.asm
    LIB target ++one.obj ++two.obj ++three.obj;
target.lib :: four.c five.c
    CL /c four.c five.c
    LIB target ++four.obj ++five.obj;
```

If any of the assembly-language files have changed more recently than the library, NMAKE assembles the source files and updates the library. Similarly, if any of the C-language files have changed, NMAKE compiles the C files and updates the library.

Accumulating Targets in Dependencies

Dependency lines are cumulative when the same target appears more than once in a single description block. For example,

```
bounce.exe : jump.obj
bounce.exe : up.obj
    echo Building bounce.exe...
```

is evaluated by NMAKE as

```
bounce.exe : jump.obj up.obj
    echo Building bounce.exe...
```

This evaluation has several effects. Since NMAKE builds the dependency tree based on one target at a time, the lines can contain other targets, as in:

```
bounce.exe leap.exe : jump.obj
bounce.exe climb.exe : up.obj
    echo Building bounce.exe...
```

NMAKE evaluates a dependency for each of the three targets as if each were specified in a separate description block. If `bounce.exe` or `climb.exe` is out-of-date, NMAKE runs the given command. If `leap.exe` is out-of-date, the given command does not apply, and NMAKE tries to use an inference rule.

Be careful when specifying the same target in different dependencies in the makefile.

If the same target is specified in two single-colon dependency lines in different locations in the makefile, and if commands appear after only one of the lines, NMAKE interprets the dependency lines as if they were adjacent or combined. This can cause an unwanted side effect: NMAKE does not invoke an inference rule for the dependency that has no commands (see “Inference Rules” on page 680). Rather, it assumes that the dependencies belong to one description block and executes the commands specified with the other dependency.

The following makefile is interpreted in the same way as the preceding examples:

```
bounce.exe : jump.obj
    echo Building bounce.exe...
.
.
.
bounce.exe : up.obj
```

This effect does not occur if the colons are doubled (`::`) after the duplicate targets. A double-colon dependency with no commands block invokes an inference rule, even if another double-colon dependency containing the same target is followed by a commands block.

Pseudotargets

A “pseudotarget” is a target that doesn’t specify a file but instead names a label for use in executing a group of commands. NMAKE interprets the pseudotarget as a file that does not exist and thus is always out-of-date. When NMAKE evaluates a pseudotarget, it always executes its commands block. Be sure that the current directory does not contain a file with a name that matches the pseudotarget.

A pseudotarget name must follow the syntax rules for filenames. Like a filename target, a pseudotarget name is not case sensitive. However, if the name does not have an extension (that is, it does not contain a period), it can exceed the 8-character limit for filenames and can be up to 256 characters long.

A pseudotarget can be listed as a dependent. A pseudotarget used this way must appear as a target in another dependency; however, that dependency does not need to have a commands block.

A pseudotarget used as a target has an assumed time stamp that is the most recent time stamp of all its dependents. If a pseudotarget has no dependents, the assumed time stamp is the current time. NMAKE uses the assumed time stamp if the pseudotarget appears as a dependent elsewhere in the makefile.

Pseudotargets are useful when you want NMAKE to build more than one target automatically. NMAKE builds only those targets specified on the NMAKE command line, or, when no command-line target is specified, it builds only the first target in the first dependency in the makefile. To tell NMAKE to build multiple targets without having to list them on the command line, write a description block with a dependency containing a pseudotarget and list as its dependents the targets you want to build. Either place this description block first in the makefile or specify the pseudotarget on the NMAKE command line.

Example 1

In the following example, UPDATE is a pseudotarget.

```
UPDATE : *.*
    !COPY *** a:\product
```

If UPDATE is evaluated, NMAKE copies all files in the current directory to the specified drive and directory.

Example 2

In the following makefile, the pseudotarget all builds both PROJECT1.EXE and PROJECT2.EXE if either all or no target is specified on the command line. The pseudotarget setenv changes the LIB environment variable before the .EXE files are updated:

```
all : setenv project1.exe project2.exe

project1.exe : project1.obj
              LINK project1;

project2.exe : project2.obj
              LINK project2;

setenv :
        set LIB=\project\lib
```

Dependents

The dependents section of the dependency line lists zero or more dependent names. Usually a dependent is a file used to build the target. However, a dependent can be any valid filename, or it can be a pseudotarget. You can specify a path with the filename. Dependents are not case sensitive. Separate each dependent name with one or more spaces or tabs. A single or double colon (: or ::) separates it from the targets section.

Along with dependents you explicitly list in the dependency line, NMAKE can assume an “inferred dependent.” An inferred dependent is derived from an inference rule. (For more information, see “Inference Rules” on page 680.) NMAKE considers an inferred dependent to appear earlier in a dependents list than explicit dependents. It builds inferred dependents into the dependency tree. It is important to note that when an inferred dependent in a dependency is out-of-date with respect to a target, NMAKE invokes the commands block associated with the dependency, just as it does with an explicit dependent.

NMAKE uses the dependency tree to make sure that dependents themselves are updated before it updates their targets. If a dependent file doesn’t exist, NMAKE looks for a way to build it; if it already exists, NMAKE looks for a way to make sure it is up-to-date. If the dependent is listed as a target in another dependency, or if it is implied as a target in an inference rule, NMAKE checks that the dependent is up-to-date with respect to its own dependents; if the dependent file is out-of-date or doesn’t exist, NMAKE executes the commands block for that dependency.

The example in Figure 18.1 lists three dependents after MYAPP.EXE:

```
myapp.exe : myapp.obj another.obj myapp.def
```

Specifying Search Paths for Dependents

You can specify the directories in which NMAKE should search for a dependent. The syntax for a directory specification is:

```
{directory[[;directory...]]}dependent
```

Enclose one or more directory names in braces ({ }). Separate multiple directories with a semicolon (;). No spaces are allowed. You can use a macro to specify part or all of a search path. NMAKE searches the current directory first, then the directories in the order specified. A search path applies only to a single dependent.

Example

The following dependency line contains a directory specification:

```
forward.exe : {\src\alpha;d:\proj}pass.obj
```

The target FORWARD.EXE has one dependent, PASS.OBJ. The directory list specifies two directories. NMAKE first searches for PASS.OBJ in the current directory. If PASS.OBJ isn't there, NMAKE searches the \SRC \ALPHA directory, then the D:\PROJ directory.

18.6 Commands

The commands section of a description block or inference rule lists the commands that NMAKE must run if the dependency is out-of-date. You can specify any command or program that can be executed from a DOS command line (with a few exceptions, such as PATH). Multiple commands can appear in a command block. Each appears on its own line (except as noted in the next section). If a description block doesn't contain any commands, NMAKE looks for an inference rule that matches the dependency. (See "Inference Rules" on page 680.) The example in Figure 18.1 shows a single command following a dependency line.

NMAKE displays each command line before it executes it, unless you specify the /S option (described on page 650), the **.SILENT** directive (described on page 687), the **!CMDSWITCHES** directive (described on page 688), or the @ modifier (described on the following page).

Command Syntax

A command line must begin with one or more spaces or tabs. NMAKE uses this indentation to distinguish between a dependency line and a command line.

Blank lines cannot appear between the dependency line and the commands block. However, a line containing only spaces or tabs can appear; this line is interpreted as a null command, and no error occurs. Blank lines can appear between command lines.

A command can be continued over more than one line.

A long command can span several lines if each line ends with a backslash (\). A backslash at the end of a line is interpreted as a space on the command line. For example, the command shown in Figure 18.1 can be expressed as:

```
    link myapp\  
another, , NUL, mylib, myapp
```

NMAKE passes the continued lines to the operating system as one long command. A command continued with a backslash must still be within the operating system's limit on the length of a command line. If any other character, such as a space or tab, follows the backslash, NMAKE interprets the backslash and the trailing characters literally.

A command can appear on a dependency line.

You can also place a single command at the end of a dependency line, whether or not other commands follow in the indented commands block. Use a semicolon (;) to separate the command from the rightmost dependent, as in:

```
project.obj : project.c project.h ; cl /c project.c
```

Command Modifiers

Command modifiers provide extra control over the commands in a description block. You can use more than one modifier for a single command. Specify a command modifier preceding the command being modified, optionally separated by spaces or tabs. Like a command, a modifier cannot appear at the beginning of a line. It must be preceded by one or more spaces or tabs.

The following describes the three NMAKE command modifiers.

@command

Prevents NMAKE from displaying the command. Any results displayed by commands are not suppressed. Spaces and tabs can appear before the command. By default, NMAKE echoes all makefile commands that it executes.

The /S option (described on page 650) suppresses display for the entire makefile; the **.SILENT** directive (described on page 687) suppresses display for part of the makefile.

-[[number]] command

Turns off error checking for the command. Spaces and tabs can appear before the command. By default, NMAKE halts when any command returns an error in the form of a nonzero exit code. This modifier tells NMAKE to ignore errors from the specified command. If the dash is followed by a number, NMAKE stops if the exit code returned by the command is greater than that number. No spaces or tabs can appear between the dash and the number; they must appear between the number and the command. (For more information on using this number, see "Exit Codes from Commands" on page 662.) The /I option (described on page 649) turns off error checking for the entire makefile; the **.IGNORE** directive (described on page 687) turns off error checking for part of the makefile.

!command

Executes the command for each dependent file if the command preceded by the exclamation point uses the predefined macros `$$$` or `$$?`. (See “Filename Macros” on page 672.) Spaces and tabs can appear before the command. The `$$$` macro represents all dependent files in the dependency line. The `$$?` macro refers to all dependent files in the dependency line that have a later time stamp than the target.

Example 1

In the following example, the at sign (`@`) suppresses display of the ECHO command line:

```
sort.exe : sort.obj
    @ECHO Now sorting...
```

The output of the ECHO command is not suppressed.

Example 2

In the following description block, if the program `sample` returns a nonzero exit code, NMAKE does not halt; if `sort` returns an exit code that is greater than 5, NMAKE stops:

```
light.lst : light.txt
    -sample light.txt
    -5 sort light.txt
```

Example 3

The description block

```
print : one.txt two.txt three.txt
    !print $$$ lpt1:
```

generates the following commands:

```
print one.txt lpt1:
print two.txt lpt1:
print three.txt lpt1:
```

Exit Codes from Commands

NMAKE stops execution if a command or program executed in the makefile encounters an error and returns a nonzero exit code. The exit code is displayed in an NMAKE error message.

You can control how NMAKE behaves when a nonzero exit code occurs by using the `/I` or `/K` option, the `.IGNORE` directive, the `!CMDSWITCHES` directive, or the dash (`-`) command modifier.

Another way to use exit codes is during preprocessing. You can run a command or program and test its exit code using the **!IF** preprocessing directive. For more information, see “Executing a Program in Preprocessing” on page 692.

Filename-Parts Syntax

NMAKE provides a syntax that you can use in commands to represent components of the name of the first dependent file. This file is generally the first file listed to the right of the colon in a dependency line. However, if a dependent is implied from an inference rule, NMAKE considers the inferred dependent to be the first dependent file, ahead of any explicit dependents. If more than one inference rule applies, the **.SUFFIXES** list determines which dependent is first. The filename components are the file’s drive, path, base name, and extension as you have specified it, not as it exists on disk.

You can represent the complete filename with the following syntax:

```
%s
```

For example, if a description block contains

```
sample.exe : c:\project\sample.obj
    LINK %s;
```

NMAKE interprets the command as

```
LINK c:\project\sample.obj;
```

You can represent parts of the complete filename with the following syntax:

```
%l [[parts]]F
```

where *parts* can be zero or more of the following letters, in any order:

Letter	Description
No letter	Complete name
d	Drive
p	Path
f	File base name
e	File extension

Using this syntax, you can represent the full filename specification by `%|F` or by `%|dpfeF`, as well as by `%s`.

Example

The following description block uses filename-parts syntax:

```
sample.exe : c:\project\sample.obj  
    LINK %s, a:%|pff.exe;
```

NMAKE interprets the first representation as the complete filename of the dependent. It interprets the second representation as a filename with the same path and base name as the dependent but on the specified drive and with the specified extension. It executes the following command:

```
LINK c:\project\sample.obj, a:\project\sample.exe;
```

Note For another way to represent components of a filename, see “Modifying Filename Macros” on page 672.

Inline Files

NMAKE can create “inline files” in the commands section of a description block or inference rule. An inline file is created on disk by NMAKE and contains text you specify in the makefile. The name of the inline file can be used in commands in the same way as any filename. NMAKE creates the inline file only when it executes the command in which the file is created.

One way to use an inline file is as a response file for another utility such as LINK or LIB. Response files avoid the operating system limit on the maximum length of a command line and automate the specification of input to a utility. Inline files eliminate the need to maintain a separate response file. They can also be used to pass a list of commands to the operating system.

Specifying an Inline File

The syntax for specifying an inline file in a command is:

```
<<[[filename]]
```

Specify the double angle brackets (<<) on the command line at the location where you want a filename to appear. Because command lines must be indented (see page 660), the angle brackets cannot appear at the beginning of a line. The angle bracket syntax must be specified literally; it cannot be represented by a macro expansion.

When NMAKE executes the description block, it replaces the inline file specification with the name of the inline file being created. The effect is the same as if a filename was literally specified in the commands section.

The *filename* supplies a name for the inline file. It must immediately follow the angle brackets; no space is permitted. You can specify a path with the filename. No extension is required or assumed. If a file by the same name already exists, NMAKE overwrites it; such a file is deleted if the inline file is temporary. (Temporary inline files are discussed in the next section.)

A name is optional; if you don't specify *filename*, NMAKE gives the inline file a unique name. If *filename* is specified, NMAKE places the file in the directory specified with the name or in the current directory if no path is specified. If *filename* is not specified, NMAKE places the inline file in the directory specified by the TMP environment variable or in the current directory if TMP is not defined. You can reuse a previous inline *filename*; NMAKE overwrites the previous file.

Creating an Inline File

The instructions for creating the inline file begin on the first line after the command. The syntax to create the inline file is:

inlinetext

.
. .
.

<<[[**KEEP** | **NOKEEP**]]

The set of angle brackets marking the end of the inline file must appear at the beginning of a separate line in the makefile. All *inlinetext* before the delimiting angle brackets is placed in the inline file. The text can contain macro expansions and substitutions. Directives and comments are not permitted in an inline file; NMAKE treats them as literal text. Spaces, tabs, and newline characters are treated literally.

The inline file can be temporary or permanent. To retain the file after the end of the NMAKE session, specify **KEEP** immediately after the closing set of angle brackets. If you don't specify a preference, or if you specify **NOKEEP** (the default), the file is temporary. **KEEP** and **NOKEEP** are not case sensitive. The temporary file exists for the duration of the NMAKE session.

It is possible to specify **KEEP** for a file that you do not name; in this case, the NMAKE-generated filename appears in the appropriate directory after the NMAKE session.

Example

The following makefile uses a temporary inline file to clear the screen and then display the contents of the current directory:

```
COMMANDS = cls ^
dir
showdir :
    <<showdir.bat
$(COMMANDS)
<<
```

In this example, the name of the inline file serves as the only command in the description block. This command has the same effect as running a batch file named SHOWDIR.BAT that contains the same commands as those listed in the macro definition.

Reusing an Inline File

After an inline file is created, you can use it more than once. To reuse an inline file in the command in which it is created, you must supply a *filename* for the file where it is defined and first used. You can then reuse the name later in the same command.

You can also reuse an inline file in subsequent commands in the same description block or elsewhere in the makefile. Be sure that the command that creates the inline file executes before all commands that use the file. Regardless of whether you specify **KEEP** or **NOKEEP**, NMAKE keeps the file for the duration of the NMAKE session.

Example

The following makefile creates a temporary LIB response file named LIB.LRF:

```
OBJECTS = add.obj sub.obj mul.obj div.obj
math.lib : $(OBJECTS)
    LIB math.lib @<<lib.lrf
--$(?: = &^
--+)
listing;
<<
    copy lib.lrf \projinfo\lib.lrf
```

The resulting response file tells LIB which library to use, the commands to execute, and the name of the listing file to produce:

```
--add.obj &
--sub.obj &
--mul.obj &
--div.obj
listing;
```

The second command in the descripton block tells NMAKE to copy the response file to another directory.

Using Multiple Inline Files

The same command can use more than one inline file.

You can specify more than one inline file in a single command line. For each inline specification, specify one or more lines of inline text followed by a closing line containing the delimiter. Begin the second file's text on the line following the delimiting line for the first file.

Example

The following example creates two inline files:

```
target.abc : depend.xyz
    copy <<file1 + <<file2 both.txt
I am the contents of file1.
<<
I am the contents of file2.
<<KEEP
```

This is equivalent to specifying

```
copy file1 + file2 both.txt
```

to concatenate two files, where FILE1 contains

```
I am the contents of file1.
```

and FILE2 contains

```
I am the contents of file2.
```

The **KEEP** keyword tells NMAKE not to delete FILE2. After the NMAKE session, the files FILE2 and BOTH.TXT exist in the current directory.

18.7 Macros

Macros offer a convenient way to replace a particular string in the makefile with another string. You can define your own macros or use predefined macros. Macros are useful for a variety of tasks, such as:

- Creating a single makefile that works for several projects. You can define a macro that replaces a dummy filename in the makefile with the specific filename for a particular project.
- Controlling the options NMAKE passes to the compiler or linker. When you specify options in a macro, you can change options throughout the makefile in a single step.

- Specifying paths in an inference rule. (For an example, see Example 3 in “User-Defined Inference Rules” on page 682.)

This section describes user-defined macros, shows how to use a macro, and discusses the macros that have special meaning for NMAKE. It ends by discussing macro substitutions, inherited macros, and precedence rules.

User-Defined Macros

To define a macro, use the following syntax:

```
macroname=string
```

The *macroname* can be any combination of letters, digits, and the underscore (`_`) character, up to 1024 characters. Macro names are case sensitive; NMAKE interprets `MyMacro` and `MYMACRO` as different macro names. The *macroname* can contain a macro invocation. If *macroname* consists entirely of an invoked macro, the macro being invoked cannot be null or undefined.

The *string* can be any sequence of zero or more characters up to 64K–25 (65,510 bytes). A string of zero characters is called a “null string.” A string consisting only of spaces, tabs, or both is also considered a null string.

Other syntax rules, such as the use of spaces, apply depending on where you specify the macro; see “Where to Define Macros” on page 669. The *string* can contain a macro invocation.

Example

The following specification defines a macro named `DIR` and assigns to it a string that represents a directory.

```
DIR=c:\objects
```

Special Characters in Macros

Certain characters have special meaning within a macro definition. You use these characters to perform specific tasks. If you want one of these characters to have a literal meaning, you must specify it using a special syntax.

- To specify a comment with a macro definition, place a number sign (`#`) and the comment after the definition, as in:

```
LINKCMD = link /CO # Prepare for debugging
```

NMAKE ignores the number sign and all characters up to the next newline character. To specify a literal number sign in a macro, use a caret (`^`), as in `^#`.

- To extend a macro definition to a new line, end the line with a backslash (`\`). The newline character that follows the backslash is replaced with a space when the macro is expanded, as in the following example:

```
LINKCMD = link myapp\  
another, , NUL, mylib, myapp
```

When this macro is expanded, a space separates `myapp` and `another`.

To specify a literal backslash at the end of the line, precede it with a caret (`^`), as in:

```
exepath = c:\bin^\
```

You can also make a backslash literal by following it with a comment specifier (`#`). NMAKE interprets a backslash as literal if it is followed by any other character.

- To insert a literal newline character into a macro, end the line with a caret (`^`). The caret tells NMAKE to interpret the newline character as part of the macro, not as a line break ending the macro definition. The following example defines a macro composed of two operating-system commands separated by a newline character:

```
CMDS = cls^  
dir
```

For an illustration of how this macro can be used, see the first example under “Inline Files” on page 664.

- To specify a literal dollar sign (`$`) in a macro definition, use two dollar signs (`$$`). NMAKE interprets a single dollar sign as the specifier for invoking a macro; see “Using Macros” on page 671.

For information on how to handle other special characters literally, regardless of whether they appear in a macro, see “Using Special Characters as Literals” on page 653.

Where to Define Macros

You can define macros in the makefile, on the command line, in a command file, or in `TOOLS.INI`. (For more information, see “Precedence among Macro Definitions” on page 680.) Each macro defined in the makefile or in `TOOLS.INI` must appear on a separate line. The line cannot start with a space or tab.

When you define a macro in the makefile or in `TOOLS.INI`, NMAKE ignores any spaces or tabs on either side of the equal sign. The *string* itself can contain embedded spaces. You do not need to enclose *string* in quotation marks (if you do, they become part of the string). The macro name being defined must appear at the

beginning of the line. Only one macro can be defined per line. For example, the following macro definition can appear in a makefile or TOOLS.INI:

```
LINKCMD = LINK /MAP
```

...are not the same as spaces on the command line.

Slightly different rules apply when you define a macro on the NMAKE command line or in a command file. The command-line parser treats spaces and tabs as argument delimiters. Therefore, spaces must not precede or follow the equal sign. If *string* contains embedded spaces or tabs, either the string itself or the entire macro must be enclosed in double quotation marks ("). For example, either form of the following command-line macro is allowed:

```
NMAKE "LINKCMD = LINK /MAP"  
NMAKE LINKCMD="LINK /MAP"
```

However, the following form of the same macro is not permitted. It contains spaces that are not enclosed by quotation marks:

```
NMAKE LINKCMD = "LINK /MAP"
```

Null Macros and Undefined Macros

An undefined macro is not the same thing as a macro defined to be null. Both kinds of macros expand to a null string. However, a macro defined to be null is still considered to be defined when used with preprocessing directives such as **!IFDEF**. (See “Preprocessing Directives” on page 688). A macro name can be “undefined” in a makefile by using the **!UNDEF** preprocessing directive.

To define a macro to be null:

- In a makefile or TOOLS.INI, specify zero or more spaces between the equal sign (=) and the end of the line, as in the following:

```
LINKOPTIONS =
```

- On the command line or in a command file, specify zero or more spaces enclosed in double quotation marks (""), or specify the entire null definition enclosed in double quotation marks, as in either of the following:

```
LINKOPTIONS=""  
"LINKOPTIONS ="
```

To undefine a macro, use **!UNDEF**, as in:

```
!UNDEF LINKOPTIONS
```

Using Macros

To use a macro (defined or not), enclose its name in parentheses preceded by a dollar sign (\$), as follows:

```
$(macroname)
```

No spaces are allowed. For example, you can use the `LINKCMD` macro defined as

```
LINKCMD = LINK /map
```

by specifying

```
$(LINKCMD)
```

NMAKE replaces the specification `$(LINKCMD)` with `LINK /map`.

An undefined macro is replaced by a null string.

If the name you use as a macro has never been defined, or was previously defined but is now undefined, NMAKE treats that name as a null string. No error occurs.

The parentheses are optional if *macroname* is a single character. For example, `$L` is equivalent to `$(L)`. However, parentheses are recommended for consistency and to avoid possible errors.

Example

The following makefile defines and uses three macros:

```
program = sample
L       = LINK
OPTIONS =

$(program).exe : $(program).obj
              $(L) $(OPTIONS) $(program).obj;
```

NMAKE interprets the description block as

```
sample.exe : sample.obj
            LINK sample.obj;
```

NMAKE replaces every occurrence of `$(program)` with `sample`, every instance of `$(L)` with `LINK`, and every instance of `$(OPTIONS)` with a null string.

Special Macros

NMAKE provides several special macros to represent various filenames and commands. One use for these macros is in the predefined inference rules. (For more information, see “Predefined Inference Rules” on page 684.) Like user-defined macro names, special macro names are case sensitive. For example, NMAKE interprets `CC` and `cc` as different macro names.

The following sections describe the four categories of special macros. The filename macros offer a convenient representation of filenames from a dependency line. The recursion macros allow you to call NMAKE from within your makefile. The command macros and options macros make it convenient for you to invoke the Microsoft language compilers.

Filename Macros

Filename macros conveniently represent filenames from the dependency line.

NMAKE provides macros that are predefined to represent filenames. The filenames are as you have specified them in the dependency line and not the full specification of the filenames as they exist on disk. As with all one-character macros, these do not need to be enclosed in parentheses. (The `$$@` and `$$$` macros are exceptions to the parentheses rule for macros; they do not require parentheses even though they contain two characters.)

`$@`

The current target's full name (path, base name, and extension), as currently specified.

`$$@`

The current target's full name (path, base name, and extension), as currently specified. This macro is valid only for specifying a dependent in a dependency line.

`$*`

The current target's path and base name minus the file extension.

`$$$`

All dependents of the current target.

`$?`

All dependents that have a later time stamp than the current target.

`$<`

The dependent file that has a later time stamp than the current target. You can use this macro only in commands in inference rules.

Example 1

The following example uses the `$?` macro, which represents all dependents that have changed more recently than the target. The `!` command modifier causes NMAKE to execute a command once for each dependent in the list. As a result, the `LIB` command is executed up to three times, each time replacing a module with a newer version.

```
trig.lib : sin.obj cos.obj arctan.obj
    !LIB trig.lib -+ $?;
```

Example 2

In the next example, NMAKE updates a file in another directory by replacing it with a file of the same name from the current directory. The `$$@` macro is used to represent the current target's full name.

```
# File in objects directory depends on version in current directory
DIR = c:\objects
$(DIR)\a.obj : a.obj
    COPY a.obj $$@
```

Modifying Filename Macros

You can append one of the modifiers in the following table to any of the filename macros to extract part of a filename. If you add one of these modifiers to the macro, you must enclose the macro name and the modifier in parentheses.

Macro modifiers specify parts of the predefined filename macros.

Modifier	Resulting Filename Part
D	Drive plus directory
B	Base name
F	Base name plus extension
R	Drive plus directory plus base name

Example 1

Assume that `$$@` represents the target `C:\SOURCE\PROG\SORT.OBJ`. The following table shows the effect of combining each modifier with `$$@`:

Macro Reference	Value
<code>\$\$(@D)</code>	<code>C:\SOURCE\PROG</code>
<code>\$\$(@F)</code>	<code>SORT.OBJ</code>
<code>\$\$(@B)</code>	<code>SORT</code>
<code>\$\$(@R)</code>	<code>C:\SOURCE\PROG\SORT</code>

If `$$@` has the value `SORT.OBJ` without a preceding directory, the value of `$$(@R)` is `SORT`, and the value of `$$(@D)` is a period (`.`) to represent the current directory.

Example 2

The following example uses the **F** modifier to specify a file of the same name in the current directory:

```
# Files in objects directory depend on versions in current directory
DIR = c:\objects
$(DIR)\a.obj $(DIR)\b.obj $(DIR)\c.obj : $$(@F)
    COPY $$(@F) $$@
```

Note For another way to represent components of a filename, see “Filename-Parts Syntax” on page 663.

Recursion macros let you use NMAKE to call NMAKE.

Recursion Macros

There are three macros that you can use when you want to call NMAKE recursively from within a makefile. These macros can make recursion more efficient.

MAKE

Defined as the name which you specified to the operating system when you ran NMAKE; this name is `NMAKE` unless you have renamed the utility file. Use this macro to call NMAKE recursively. The `/N` command-line option to prevent execution of commands does not prevent this command from executing. It is recommended that you do not redefine **MAKE**.

MAKEDIR

Defined as the current directory when NMAKE was called.

MAKEFLAGS

Defined as the NMAKE options currently in effect. This macro is passed automatically when you call NMAKE recursively. However, specification of this macro when invoking recursion is harmless; thus, you can use older makefiles that specify this macro. You cannot redefine **MAKEFLAGS**. To change the `/D`, `/I`, `/N`, and `/S` options within a makefile, use the preprocessing directive **!CMDSWITCHES**. (See “Preprocessing Directives” on page 688.) To add other options to the ones already in effect for NMAKE when recursing, specify them as part of the recursion command.

Calling NMAKE Recursively

In a commands block, you can specify a call to NMAKE itself. Either invoke the **MAKE** macro or specify NMAKE literally. When you call NMAKE recursively by macro rather than by literally specifying the command `NMAKE`, NMAKE does not run a new copy of itself. Instead, it uses its own stack for the recursive parts of the build. This saves space in memory.

The following NMAKE information is available to the called NMAKE session during recursion:

- Environment-variable macros (see “Inherited Macros” on page 679). To cause all macros to be inherited, specify the `/V` option.
- The **MAKEFLAGS** macro. If `.IGNORE` (or `!CMDSWITCHES +I`) is set, **MAKEFLAGS** contains an `I` when it is passed to the recursive call. Likewise, if `.SILENT` (or `!CMDSWITCHES +S`) is set, **MAKEFLAGS** contains an `S` when passed to the call.
- Macros specified on the command line for the recursive call.
- All information in `TOOLS.INI`.

Inference rules defined in the makefile are not passed to the called NMAKE session. Settings for `.SUFFIXES` and `.PRECIOUS` are also not inherited. However, you can make `.SUFFIXES`, `.PRECIOUS`, and all inference rules available to the recursive call either by specifying them in `TOOLS.INI` or by placing them in a file

that is specified in an **!INCLUDE** directive in the makefile for each NMAKE session.

Example

The **MAKE** macro is useful for building different versions of a program. The following makefile calls NMAKE recursively to build targets in the \VERS1 and \VERS2 directories.

```
all : vers1 vers2

vers1 :
    cd \vers1
    $(MAKE)
    cd ..

vers2 :
    cd \vers2
    $(MAKE) /F vers2.mak
    cd ..
```

If the dependency containing `vers1` as a target is executed, NMAKE performs the commands to change to the \VERS1 directory and call itself recursively using the MAKEFILE in that directory. If the dependency containing `vers2` as a target is executed, NMAKE changes to the \VERS2 directory and calls itself using the file VERS2.MAK in that directory.

Note Deeply recursive builds can exhaust NMAKE's run-time stack, causing an error. If this occurs, use the EXEHDR utility to increase NMAKE's stack. See Chapter 17, "Using EXEHDR."

Command Macros

NMAKE predefines several macros to represent commands for Microsoft products. You can use these macros as commands in either a description block or an inference rule; they are automatically used in NMAKE's predefined inference rules. (See "Inference Rules" on page 680.) You can redefine these macros to represent part or all of a command line, including options.

AS

Defined as `m1`, the command to run the Microsoft Macro Assembler

BC

Defined as `bc`, the command to run the Microsoft Basic Compiler

CC

Defined as `c1`, the command to run the Microsoft C Compiler

COBOL

Defined as `cobol`, the command to run the Microsoft COBOL Compiler

Command macros are shortcut calls to Microsoft compilers.

CPP

Defined as `c1`, the command to run the Microsoft C++ Compiler

CXX

Defined as `c1`, the command to run the Microsoft C++ Compiler

FOR

Defined as `f1`, the command to run the Microsoft FORTRAN Compiler

PASCAL

Defined as `p1`, the command to run the Microsoft Pascal Compiler

RC

Defined as `rc`, the command to run the Microsoft Resource Compiler

Options Macros

Options macros pass preset options to Microsoft compilers.

The following macros represent options to be passed to the commands for invoking the Microsoft language compilers. These macros are used automatically in the predefined inference rules. (See “Predefined Inference Rules” on page 684.) By default, these macros are undefined. You can define them to mean the options you want to pass to the compilers, and you can use these macros in commands in description blocks and inference rules. As with all macros, the options macros can be used even if they are undefined; a macro that is undefined or defined to be a null string generates a null string where it is used.

AFLAGS

Passes options to the Microsoft Macro Assembler

BFLAGS

Passes options to the Microsoft Basic Compiler

CFLAGS

Passes options to the Microsoft C Compiler

COBFLAGS

Passes options to the Microsoft COBOL Compiler

CPPFLAGS

Passes options to the Microsoft C++ Compiler

CXXFLAGS

Passes options to the Microsoft C++ Compiler

FFLAGS

Passes options to the Microsoft FORTRAN Compiler

PFLAGS

Passes options to the Microsoft Pascal Compiler

RFLAGS

Passes options to the Microsoft Resource Compiler

Substitution Within Macros

Just as macros allow you to substitute text in a makefile, you can also substitute text within a macro itself. The substitution applies only to the current use of the macro and does not modify the original macro definition. To substitute text within a macro, use the following syntax:

```
$(macroname:string1=string2)
```

Every occurrence of *string1* is replaced by *string2* in the macro *macroname*. Do not put any spaces or tabs before the colon. Spaces that appear after the colon are interpreted as part of the string in which they occur. If *string2* is a null string, all occurrences of *string1* are deleted from the *macroname* macro.

Macro substitution is literal and case sensitive. This means that the case as well as the characters in *string1* must match the target string in the macro exactly, or the substitution is not performed. This also means that *string2* is substituted exactly as it is specified. Because substitution is literal, the strings cannot contain macro expansions.

Example 1

The following makefile illustrates macro substitution:

```
SOURCES = project.c one.c two.c

project.exe : $(SOURCES:.c=.obj)
    LINK **;
```

The predefined macro `**` stands for the names of all the dependent files (See “Filename Macros” on page 672.) When this makefile is run, NMAKE executes the following command:

```
LINK project.obj one.obj two.obj;
```

The macro substitution does not alter the `SOURCES` macro definition; if it is used again elsewhere in the makefile, `SOURCES` has its original value as it was defined.

Example 2

If the macro `OBJS` is defined as

```
OBJS = ONE.OBJ TWO.OBJ THREE.OBJ
```

you can replace each space in the defined value of `OBJS` with a space, followed by a plus sign, followed by a newline character, by using

```
$(OBJS: = +^
)
```


The caret (`^`) tells NMAKE to treat the end of the line as a literal newline character. The expanded macro after substitution is:

```
ONE.OBJ +  
TWO.OBJ +  
THREE.OBJ
```

This example is useful for creating response files.

Substitution Within Predefined Macros

You can also substitute text in any predefined macro (except `$$@`) using the same syntax as for other macros.

The command in the following description block makes a substitution within the predefined macro `$@`, which represents the full name of the current target. Note that although `$@` is a single-character macro, when it is used in a substitution, it must be enclosed in parentheses.

```
target.abc : depend.xyz  
    echo $(@:targ=blank)
```

NMAKE substitutes `blank` for `targ` in the target, resulting in the string `blanket.abc`. If dependent `depend.xyz` has a later time stamp than target `target.abc`, then NMAKE executes the command

```
echo blanket.abc
```

Environment-Variable Macros

When NMAKE executes, it inherits macro definitions equivalent to every environment variable that existed before the start of the NMAKE session. If a variable such as `LIB` or `INCLUDE` has been set in the operating-system environment, you can use its value as if you had specified an NMAKE macro with the same name and value. The inherited macro names are converted to uppercase. Inheritance occurs before preprocessing. The `/E` option causes macros inherited from environment variables to override any macros with the same name in the makefile.

You can redefine environment-variable macros the same way that you define or redefine other macros. Changing a macro does not change the corresponding environment variable; to change the variable, use a `SET` command. Also, using the `SET` command to change an environment variable in an NMAKE session does not change the corresponding macro; to change the macro, use a macro definition.

If an environment variable has not been set in the operating-system environment, it cannot be set using a macro definition. However, you can use a `SET` command in the NMAKE session to set the variable. The variable is then in effect for the rest

of the NMAKE session unless redefined or cleared by a later SET command. A SET definition that appears in a makefile does not create a corresponding macro for that variable name; if you want a macro for an environment variable that is created during an NMAKE session, you must explicitly define the macro in addition to setting the variable.

Warning If an environment variable contains a dollar sign (\$), NMAKE interprets it as the beginning of a macro invocation. The resulting macro expansion can cause unexpected behavior and possibly an error.

Example

The following makefile redefines the environment-variable macro called LIB:

```
LIB = c:\tools\lib

sample.exe : sample.obj
    LINK sample;
```

No matter what value the environment variable LIB had before, it has the value `c:\tools\lib` when NMAKE executes the LINK command in this description block. Redefining the inherited macro does not affect the original environment variable; when NMAKE terminates, LIB still has its original value.

If LIB is not defined before the NMAKE session, the LIB macro definition in the preceding example does not set a LIB environment variable for the LINK command. To do this, use the following makefile:

```
sample.exe : sample.obj
    SET LIB=c:\tools.lib
    LINK sample;
```

Inherited Macros

When NMAKE is called recursively, the only macros that are inherited by the called NMAKE are those defined on the command line or in environment variables. Macros defined in the makefile are not inherited when NMAKE is called recursively. There are several ways to pass macros to a recursive NMAKE session:

- Run NMAKE with the /V option. This option causes all macros to be inherited by the recursively called NMAKE. You can use this option on the NMAKE command for the entire session, or you can specify it in a command for a recursive NMAKE call to affect just the specified recursive session.
- Use the SET command before the recursive call to set an environment variable before the called NMAKE session.

- Define a macro on the command line for the recursive call.
- Define a macro in the `TOOLS.INI` file. Each time `NMAKE` is recursively called, it reads `TOOLS.INI`.

Precedence Among Macro Definitions

If you define the same macro name in more than one place, `NMAKE` uses the macro with the highest precedence. The precedence from highest to lowest is as follows:

1. A macro defined on the command line
2. A macro defined in a makefile or include file
3. An inherited environment-variable macro
4. A macro defined in the `TOOLS.INI` file
5. A predefined macro, such as `CC` and `AS`

The `/E` option causes macros inherited from environment variables to override any macros with the same name in the makefile. The `!UNDEF` directive in a makefile overrides a macro defined on the command line.

18.8 Inference Rules

Inference rules are templates that define how a file with one extension is created from a file with another extension. `NMAKE` uses inference rules to supply commands for updating targets and to infer dependents for targets. In the dependency tree, inference rules cause targets to have inferred dependents as well as explicitly specified dependents; see “Inferred Dependents” on page 685. The `.SUFFIXES` list determines priorities for applying inference rules; see “Dot Directives” on page 687.

Inference rules provide a convenient shorthand for common operations. For instance, you can use an inference rule to avoid repeating the same command in several description blocks. You can define your own inference rules or use predefined inference rules. Inference rules can be specified in the makefile or in `TOOLS.INI`.

Inference rules can be used in the following situations:

- If `NMAKE` encounters a description block that has no commands, it checks the `.SUFFIXES` list and the files in the current or specified directory and then searches for an inference rule that matches the extensions of the target and an existing dependent file with the highest possible `.SUFFIXES` priority.

- If a dependent file doesn't exist and is not listed as a target in another description block, NMAKE looks for an inference rule that shows how to create the missing dependent from another file with the same base name.
- If a target has no dependents and its description block has no commands, NMAKE can use an inference rule to create the target.
- If a target is specified on the command line and there is no makefile (or no mention of the target in the makefile), inference rules are used to build the target.

If a target is used in more than one single-colon dependency, an inference rule might not be applied as expected; see “Accumulating Targets in Dependencies” on page 657.

Inference Rule Syntax

To define an inference rule, use the following syntax:

```
.fromext.toext:  
  commands
```

The first line lists two extensions: *fromext* represents the extension of a dependent file, and *toext* represents the extension of a target file. Extensions are not case sensitive. Macros can be invoked to represent *fromext* and *toext*; the macros are expanded during preprocessing.

The period (.) preceding *fromext* must appear at the beginning of the line. The colon (:) can be preceded by zero or more spaces or tabs; it can be followed only by spaces or tabs, a semicolon (;) to specify a command, a number sign (#) to specify a comment, or a newline character. No other spaces are allowed.

The rest of the inference rule gives the commands to be run if the dependency is out-of-date. Use the same rules for commands in inference rules as in description blocks. (See “Commands” on page 660.)

An inference rule applies to a single target and dependent.

An inference rule can be used only when a target and dependent have the same base name. You cannot use a rule to match multiple targets or dependents. For example, you cannot define an inference rule that replaces several modules in a library because all but one of the modules must have a different base name from the target library.

Inference rules apply only to files with extensions listed in .SUFFIXES.

Inference rules can exist only for dependents with extensions that are listed in the **.SUFFIXES** directive. (For information on **.SUFFIXES**, see “Dot Directives” on page 687.) If an out-of-date dependency does not have a commands block, and if the **.SUFFIXES** list contains the extension of the dependent, NMAKE looks for an inference rule matching the extensions of the target and of an existing file in the current or specified directory. If more than one rule matches existing dependent files, NMAKE uses the order of the **.SUFFIXES** list to determine which rule to

invoke. Priority in the list descends from left to right. NMAKE may invoke a rule for an inferred dependent even if an explicit dependent is specified; for more information, see “Inferred Dependents” on page 685.

Inference rules can make a makefile unnecessary.

Inference rules tell NMAKE how to build a target specified on the command line if no makefile is provided or if the makefile does not have a dependency containing the specified target. When a target is specified on the command line and NMAKE cannot find a description block to run, it looks for an inference rule to tell it how to build the target. You can run NMAKE without a makefile if the inference rules that are predefined or defined in TOOLS.INI are all you need for your build.

Inference Rule Search Paths

The inference-rule syntax described previously tells NMAKE to look for the specified files in the current directory. You can also specify directories to be searched by NMAKE when it looks for files. An inference rule that specifies paths has the following syntax:

```
{frompath},fromext {topath}.toext:  
  commands
```

No spaces are allowed. The *frompath* directory must match the directory specified for the dependent file; similarly, *topath* must match the target’s directory specification. For NMAKE to apply an inference rule to a dependency, the paths in the dependency line must match the paths specified in the inference rule exactly. For example, if the current directory is called PROJ, the inference rule

```
{..\proj}.exe{..\proj}.obj:
```

does not apply to the dependency

```
project1.exe : project1.obj
```

If you use a path on one extension in the inference rule, you must use paths on both. You can specify the current directory by either a period (.) or an empty pair of braces ({}).

You can specify only one path for each extension in an inference rule. To specify more than one path, you must create a separate inference rule for each path.

Macros can be invoked to represent *frompath* and *topath*; the macros are expanded during preprocessing.

User-Defined Inference Rules

The following examples illustrate several ways to write inference rules.

Example 1

The following makefile contains an inference rule and a minimal description block:

```
.c.obj:
    cl /c $<

sample.obj :
```

The inference rule tells NMAKE how to build a .OBJ file from a .C file. The predefined macro \$< represents the name of a dependent that has a later time stamp than the target. The description block lists only a target, SAMPLE.OBJ; there is no dependent or command. However, given the target's base name and extension, plus the inference rule, NMAKE has enough information to build the target.

After checking to be sure that .c is one of the extensions in the .SUFFIXES list, NMAKE looks for a file with the same base name as the target and with the .C extension. If SAMPLE.C exists (and no files with higher-priority extensions exist), NMAKE compares its time to that of SAMPLE.OBJ. If SAMPLE.C has changed more recently, NMAKE compiles it using the CL command listed in the inference rule:

```
cl /c sample.c
```

Example 2

The following inference rule compares a .C file in the current directory with the corresponding .OBJ file in another directory:

```
{.}.c{c:\objects}.obj:
    cl /c $<;
```

The path for the .C file is represented by a period. A path for the dependent extension is required because one is specified for the target extension.

This inference rule matches a dependency line containing the same combination of paths, such as:

```
c:\objects\test.obj : test.c
```

This rule does not match a dependency line such as:

```
test.obj : test.c
```

In this case, NMAKE uses the predefined inference rule for .c.obj when building the target.

Example 3

The following inference rule uses macros to specify paths in an inference rule:

```

C_DIR = proj1src
OBJ_DIR = proj1obj
{$(C_DIR)}.c{$(OBJ_DIR)}.obj:
    cl /c $

```

If the macros are redefined, NMAKE uses the definition that is current at that point during preprocessing. To reuse an inference rule with different macro definitions, you must repeat the rule after the new definition:

```

C_DIR = proj1src
OBJ_DIR = proj1obj
{$(C_DIR)}.c{$(OBJ_DIR)}.obj:
    cl /c $<
C_DIR = proj2src
OBJ_DIR = proj2obj
{$(C_DIR)}.c{$(OBJ_DIR)}.obj:
    cl /c $<

```

Predefined Inference Rules

NMAKE provides predefined inference rules containing commands for creating object, executable, and resource files. Table 18.1 describes the predefined inference rules.

Table 18.1 Predefined Inference Rules

Rule	Command	Default Action
.asm.exe	\$(AS) \$(AFLAGS) \$*.asm	ML \$*.ASM
.asm.obj	\$(AS) \$(AFLAGS) /c \$*.asm	ML /c \$*.ASM
.c.exe	\$(CC) \$(CFLAGS) \$*.c	CL \$*.C
.c.obj	\$(CC) \$(CFLAGS) /c \$*.c	CL /c \$*.C
.cpp.exe	\$(CPP) \$(CPPFLAGS) \$*.cpp	CL \$*.CPP
.cpp.obj	\$(CPP) \$(CPPFLAGS) /c \$*.cpp	CL /c \$*.CPP
.cxx.exe	\$(CXX) \$(CXXFLAGS) \$*.cxx	CL \$*.CXX
.cxx.obj	\$(CXX) \$(CXXFLAGS) /c \$*.cxx	CL /c \$*.CXX
.bas.obj	\$(BC) \$(BFLAGS) \$*.bas;	BC \$*.BAS;
.cbl.exe	\$(COBOL) \$(COBFLAGS) \$*.cbl, \$*.exe;	COBOL \$*.CBL, \$*.EXE;
.cbl.obj	\$(COBOL) \$(COBFLAGS) \$*.cbl;	COBOL \$*.CBL;
.for.exe	\$(FOR) \$(FFLAGS) \$*.for	FL \$*.FOR
.for.obj	\$(FOR) /c \$(FFLAGS) \$*.for	FL /c \$*.FOR
.pas.exe	\$(PASCAL) \$(PFLAGS) \$*.pas	PL \$*.PAS
.pas.obj	\$(PASCAL) /c \$(PFLAGS) \$*.pas	PL /c \$*.PAS
.rc.res	\$(RC) \$(RFLAGS) /r \$*	RC /r \$*

For example, assume you have the following makefile:

```
sample.exe :
```

This description block lists a target without any dependents or commands. NMAKE looks at the target's extension (.EXE) and searches for an inference rule that describes how to create an .EXE file. Table 18.1 shows that more than one inference rule exists for building an .EXE file. NMAKE uses the order of the extensions appearing in the **.SUFFIXES** list to determine which rule to invoke. It then looks in the current or specified directory for a file that has the same base name as the target `sample` and one of the extensions in the **.SUFFIXES** list; it checks the extensions one by one until it finds a matching dependent file in the directory.

For example, if a file called `SAMPLE.FOR` exists, NMAKE applies the `.for.exe` inference rule. If both `SAMPLE.C` and `SAMPLE.FOR` exist, and if `.c` appears before `.for` in the **.SUFFIXES** list, NMAKE instead uses the `.c.exe` inference rule to compile `SAMPLE.C` and links the resulting file `SAMPLE.OBJ` to create `SAMPLE.EXE`.

Note By default, the options macros such as **CFLAGS** are undefined. As explained in “Using Macros” on page 671, this causes no problem; NMAKE replaces an undefined macro with a null string. Because the predefined options macros are included in the inference rules, you can define these macros and have their assigned values passed automatically to the predefined inference rules.

Inferred Dependents

NMAKE can assume an “inferred dependent” for a target if there is an applicable inference rule. An inference rule is applicable if:

- The *toext* in the rule matches the extension of the target being evaluated.
- The *fromext* in the rule matches the extension of a file that has the same base name as the target and that exists in the current or specified directory.
- The *fromext* is in the **.SUFFIXES** list.
- No other *fromext* in a matching rule is listed in **.SUFFIXES** with a higher priority.
- No explicitly specified dependent has a higher priority extension.

If an existing dependent matches an inference rule and has an extension with a higher **.SUFFIXES** priority, NMAKE does not infer a dependent.

NMAKE does not necessarily execute the commands block in an inference rule for an inferred dependent. If the target's description block contains commands, NMAKE executes the description block's commands and not the commands in the

inference rule. The effect of an inferred dependent is illustrated in the following example:

```
project.obj :  
    cl /Zi /c project.c
```

If a makefile contains this description block and if the current directory contains a file named `PROJECT.C` and no other files, `NMAKE` uses the predefined inference rule for `.c.obj` to infer the dependent `project.c`. It does not execute the predefined rule's command, `cl /c project.c`. Instead, it runs the command specified in the makefile.

Inferred dependents can cause unexpected side effects. In the following examples, assume that both `PROJECT.ASM` and `PROJECT.C` exist and that `.SUFFIXES` contains the default setting. If the makefile contains

```
project.obj : project.c
```

`NMAKE` infers the dependent `project.asm` ahead of `project.c` because `.SUFFIXES` lists `.asm` before `.c` and because a rule for `.asm.obj` exists. If either `PROJECT.ASM` or `PROJECT.C` is out-of-date, `NMAKE` executes the commands in the rule for `.asm.obj`.

However, if the dependency in the preceding example is followed by a commands block, `NMAKE` executes those commands and not the commands in the inference rule for the inferred dependent.

Another side effect occurs because `NMAKE` builds a target if it is out-of-date with respect to any of its dependents, whether explicitly specified or inferred. For example, if `PROJECT.OBJ` is up-to-date with respect to `PROJECT.C` but not with respect to `PROJECT.ASM`, and if the makefile contains

```
project.obj : project.c  
    cl /Zi /c project.c
```

`NMAKE` infers the dependent `project.asm` and updates the target using the command specified in this description block.

Precedence Among Inference Rules

If the same inference rule is defined in more than one place, `NMAKE` uses the rule with the highest precedence. The precedence from highest to lowest is as follows:

1. An inference rule defined in the makefile. If more than one rule is defined, the last rule applies.
2. An inference rule defined in the `TOOLS.INI` file. If more than one rule is defined, the last rule applies.

3. A predefined inference rule.

User-defined inference rules always override predefined inference rules. NMAKE uses a predefined inference rule only if no user-defined inference rule exists for a given target and dependent.

If two inference rules match a target's extension and a dependent is not specified, NMAKE uses the inference rule whose dependent's extension appears first in the `.SUFFIXES` list.

18.9 Directives

NMAKE provides several ways to control the NMAKE session through dot directives and preprocessing directives. Directives are instructions to NMAKE that are placed in the makefile or in `TOOLS.INI`. NMAKE interprets dot directives and preprocessing directives and applies the results to the makefile before processing dependencies and commands.

Dot Directives

Dot directives must appear outside a description block and must appear at the beginning of a line. Dot directives begin with a period (`.`) and are followed by a colon (`:`). Spaces and tabs can precede and follow the colon. These directive names are case sensitive and must be uppercase.

.IGNORE :

Ignores nonzero exit codes returned by programs called from the makefile. By default, NMAKE halts if a command returns a nonzero exit code. This directive affects the makefile from the place it is specified to the end of the file. To turn it off again, use the `!CMDSWITCHES` preprocessing directive. To ignore the exit code for a single command, use the dash (`-`) command modifier. To ignore exit codes for an entire file, invoke NMAKE with the `/I` option.

.PRECIOUS : *targets*

Tells NMAKE not to delete *targets* if the commands that build them are interrupted. This directive has no effect if a command is interrupted and handles the interrupt by deleting the file. Separate the target names with one or more spaces or tabs. By default, NMAKE deletes the target if building was interrupted by `CTRL+C` or `CTRL+BREAK`. Multiple specifications are cumulative; each use of `.PRECIOUS` applies to the entire makefile.

.SILENT :

Suppresses display of the command lines as they are executed. By default, NMAKE displays the commands it invokes. This directive affects the makefile from the place it is specified to the end of the file. To turn it off again, use the `!CMDSWITCHES` preprocessing directive. To suppress display of a single

command line, use the @ command modifier. To suppress the command display for an entire file, invoke NMAKE with the /S option.

.SUFFIXES : *list*

Lists file suffixes (extensions) for NMAKE to try to match when it attempts to apply an inference rule. (For details about using **.SUFFIXES**, see “Inference Rules” on page 680.) The list is predefined as follows:

```
.SUFFIXES : .exe .obj .asm .c .cpp .cxx .bas .cbl .for .pas .res .rc
```

To add additional suffixes to the end of the list, specify

.SUFFIXES : *suffixlist*

where *suffixlist* is a list of the additional suffixes, separated by one or more spaces or tabs. To clear the list, specify

```
.SUFFIXES :
```

without extensions. To change the list order or to specify an entirely new list, you must clear the list and specify a new setting. To see the current setting, run NMAKE with the /P option.

Preprocessing Directives

NMAKE preprocessing directives are similar to compiler preprocessing directives. You can use several of the directives to conditionally process the makefile. With other preprocessing directives you can display error messages, include other files, undefine a macro, and turn certain options on or off. NMAKE reads and executes the preprocessing directives before processing the makefile as a whole.

Preprocessing directives begin with an exclamation point (!), which must appear at the beginning of the line. Zero or more spaces or tabs can appear between the exclamation point and the directive keyword; this allows indentation for readability. These directives (and their keywords and operators) are not case sensitive.

!CMDSWITCHES {+|-}*opt...*

Turns on or off one or more options. (For descriptions of options, see page 647.) Specify an operator, either a plus sign (+) to turn options on or a minus sign (-) to turn options off, followed by one or more letters representing options. Letters are not case sensitive. Do not specify the slash (/). Separate the directive from the operator by one or more spaces or tabs; no space can appear between the operator and the options. To turn on some options and turn off other options, use separate specifications of the **!CMDSWITCHES** directives.

All options with the exception of /F, /HELP, /NOLOGO, /X, and /? can appear in **!CMDSWITCHES** specifications in TOOLS.INI. In a makefile, only the letters D, I, N, and S can be specified. If **!CMDSWITCHES** is specified within a description block, the changes do not take effect until the next description block. This directive updates the **MAKEFLAGS** macro; the changes are inherited during recursion.

!ERROR *text*

Displays *text* to standard error in the message for error U1050, then stops the NMAKE session. This directive stops the build even if /K, /I, **.IGNORE**, **!CMDSWITCHES**, or the dash (-) command modifier is used. Spaces or tabs before *text* are ignored.

!MESSAGE *text*

Displays *text* to standard output, then continues the NMAKE session. Spaces or tabs before *text* are ignored.

!INCLUDE [[<]]*filename*[[>]]

Reads and evaluates the file *filename* as a makefile before continuing with the current makefile. NMAKE first looks for *filename* in the current directory if *filename* is specified without a path; if a path is specified, NMAKE looks in the specified directory. Next, if the **!INCLUDE** directive is itself contained in a file that is included, NMAKE looks for *filename* in the parent file's directory; this search is recursive, ending with the original makefile's directory. Finally, if *filename* is enclosed by angle brackets (< >), NMAKE searches in the directories specified by the **INCLUDE** macro. The **INCLUDE** macro is initially set to the value of the **INCLUDE** environment variable.

!IF *constantexpression*

Processes the statements between the **!IF** and the next **!ELSE** or **!ENDIF** if *constantexpression* evaluates to a nonzero value.

!IFDEF *macroname*

Processes the statements between the **!IFDEF** and the next **!ELSE** or **!ENDIF** if *macroname* is defined. NMAKE considers a macro with a null value to be defined.

!IFNDEF *macroname*

Processes the statements between the **!IFNDEF** and the next **!ELSE** or **!ENDIF** if *macroname* is not defined.

!ELSE [[**!IF** *constantexpression*||**!IFDEF** *macroname*||**!IFNDEF** *macroname*]]

Processes the statements between the **!ELSE** and the next **!ENDIF** if the preceding **!IF**, **!IFDEF**, or **!IFNDEF** statement evaluated to zero. The optional keywords give further control of preprocessing.

!ELSEIF

Synonym for **!ELSE IF**.

!ELSEIFDEF

Synonym for **!ELSE IFDEF**.

!ELSEIFNDEF

Synonym for **!ELSE IFNDEF**.

!ENDIF

Marks the end of an **!IF**, **!IFDEF**, or **!IFNDEF** block. Anything following **!ENDIF** on the same line is ignored.

!UNDEF *macroname*

Undefines a macro by removing *macroname* from NMAKE's symbol table. (For more information, see "Null Macros and Undefined Macros" on page 670.)

Example

The following set of directives

```
!IF
!ELSE
! IF
! ENDIF
!ENDIF
```

is equivalent to the set of directives

```
!IF
!ELSE IF
!ENDIF
```

Expressions in Preprocessing

The *constantexpression* used with the **!IF** or **!ELSE IF** directives can consist of integer constants, string constants, or program invocations. You can group expressions by enclosing them in parentheses. NMAKE treats numbers as decimals unless they start with 0 (octal) or 0x (hexadecimal).

Expressions in NMAKE use C-style signed long integer arithmetic; numbers are represented in 32-bit two's-complement form and are in the range -2147483648 to 2147483647.

Two unary operators evaluate a condition and return a logical value of true (1) or false (0):

DEFINED (*macroname*)

Evaluates to true if *macroname* is defined. In combination with the **!IF** or **!ELSE IF** directives, this operator is equivalent to the **!IFDEF** or **!ELSE IFDEF** directives. However, unlike these directives, **DEFINED** can be used in complex expressions using binary logical operators.

EXIST (*path*)

Evaluates to true if *path* exists. **EXIST** can be used in complex expressions using binary logical operators. If *path* contains spaces (allowed in some file systems), enclose it in double quotation marks.

Integer constants can use the unary operators for numerical negation (**-**), one's complement (**~**), and logical negation (**!**).

Constant expressions can use any binary operator listed in Table 18.2. To compare two strings, use the equality (**==**) operator and the inequality (**!=**) operator. Enclose strings in double quotation marks.

Table 18.2 Binary Operators for Preprocessing

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

Example

The following example shows how preprocessing directives can be used to control whether the linker inserts debugging information into the .EXE file:

```
!INCLUDE <infrules.txt>
!CMDSWITCHES +D
winner.exe : winner.obj
!IF DEFINED(debug)
!   IF "$ (debug)"=="y"
       LINK /CO winner.obj;
!   ELSE
       LINK winner.obj;
!   ENDEF
!ELSE
!   ERROR Macro named debug is not defined.
!ENDIF
```

In this example, the **!INCLUDE** directive inserts the INFRULES.TXT file into the makefile. The **!CMDSWITCHES** directive sets the /D option, which displays the time stamps of the files as they are checked. The **!IF** directive checks to see if the macro `debug` is defined. If it is defined, the next **!IF** directive checks to see if it is set to `y`. If it is, NMAKE reads the LINK command with the /CO option; otherwise, NMAKE reads the LINK command without /CO. If the `debug` macro is not defined, the **!ERROR** directive prints the specified message and NMAKE stops.

Executing a Program in Preprocessing

NMAKE can run programs before processing the makefile.

You can invoke a program or command from within NMAKE and use its exit code during preprocessing. NMAKE executes the command during preprocessing, and it replaces the specification in the makefile with the command's exit code. A nonzero exit code usually indicates an error. You can use this value in an expression to control preprocessing.

Specify the command, including any arguments, within brackets ([]). You can use macros in the command specification; NMAKE expands the macro before executing the command.

Example

The following part of a makefile tests the space on disk before continuing the NMAKE session:

```
!IF [c:\util\checkdisk] != 0
!   ERROR Not enough disk space; NMAKE terminating.
!ENDIF
```

18.10 Sequence of NMAKE Operations

When you write a complex makefile, it can be helpful to know the sequence in which NMAKE performs operations. This section describes those operations and their order.

NMAKE first looks for a makefile.

When you run NMAKE from the command line, NMAKE's first task is to find the makefile:

1. If the /F option is used, NMAKE searches for the filename specified in the option. If NMAKE cannot find that file, it returns an error.
2. If the /F option is not used, NMAKE looks for a file named MAKEFILE in the current directory. If there are targets on the command line, NMAKE builds them according to the instructions in MAKEFILE. If there are no targets on the command line, NMAKE builds only the first target it finds in MAKEFILE.
3. If NMAKE cannot find MAKEFILE, NMAKE looks for target files on the command line and attempts to build them using inference rules (either defined by the user in TOOLS.INI or predefined by NMAKE). If no target is specified, NMAKE returns an error.

Macro definitions follow a priority.

NMAKE then assigns macro definitions with the following precedence (highest to lowest):

1. Macros defined on the command line
2. Macros defined in a makefile or include file
3. Inherited macros
4. Macros defined in the TOOLS.INI file
5. Predefined macros (such as **CC** and **RFLAGS**)

Macro definitions are assigned first in order of priority and then in the order in which NMAKE encounters them. For example, a macro defined in an include file overrides a macro with the same name from the TOOLS.INI file. Note that a macro within a makefile can be redefined; a macro is valid from the point it is defined until it is redefined or undefined.

Inference rules also follow a priority.

NMAKE also assigns inference rules, using the following precedence (highest to lowest):

1. Inference rules defined in a makefile or include file
2. Inference rules defined in the TOOLS.INI file
3. Predefined inference rules (such as .c.obj)

You can use command-line options to change some of these priorities.

- The /E option allows macros inherited from the environment to override macros defined in the makefile.
- The /R option tells NMAKE to ignore macros and inference rules that are defined in TOOLS.INI or are predefined.

NMAKE preprocesses directives before running the makefile commands.

Next, NMAKE evaluates any preprocessing directives. If an expression for conditional preprocessing contains a program in brackets ([]), the program is invoked during preprocessing and the program's exit code is used in the expression. If an **!INCLUDE** directive is specified for a file, NMAKE preprocesses the included file before continuing to preprocess the rest of the makefile. Preprocessing determines the final makefile that NMAKE reads.

NMAKE updates targets in the makefile.

NMAKE is now ready to update the targets. If you specified targets on the command line, NMAKE updates only those targets. If you did not specify targets on the command line, NMAKE updates only the first target in the makefile. If you specify a pseudotarget, NMAKE always updates the target. If you use the /A option, NMAKE always updates the target, even if the file is not out-of-date.

NMAKE updates a target by comparing its time stamp to the time stamp of each dependent of that target. A target is out-of-date if any dependent has a later time stamp; if the /B option is specified, a target is out-of-date if any dependent has a later or equal time stamp.

If the dependents of the targets are themselves out-of-date or do not exist, NMAKE updates them first. If the target has no explicit dependent, NMAKE looks for an inference rule that matches the target. If a rule exists, NMAKE updates the target using the commands given with the inference rule. If more than one rule applies to the target, NMAKE uses the priority in the **.SUFFIXES** list to determine which inference rule to use.

Errors usually stop the build.

NMAKE normally stops processing the makefile when a command returns a non-zero exit code. In addition, if NMAKE cannot tell whether the target was built successfully, it deletes the target. The /I command-line option, **.IGNORE** directive, **!CMDSWITCHES** directive, and dash (-) command modifier all tell NMAKE to ignore error codes and attempt to continue processing. The /K option tells NMAKE to continue processing unrelated parts of the build if an error occurs. The **.PRECIOUS** directive prevents NMAKE from deleting a partially created target if you interrupt the build with CTRL+C or CTRL+BREAK. You can document errors by using the **!ERROR** directive to print descriptive text. The directive causes NMAKE to print some text, then stop the build.

18.11 A Sample NMAKE Makefile

The following example illustrates many of NMAKE's features. The makefile creates an executable file from C-language source files:

```
# This makefile builds SAMPLE.EXE from SAMPLE.C,  
# ONE.C, and TWO.C, then deletes intermediate files.  
  
CFLAGS = /c /AL /Od $(CODEVIEW) # controls compiler options  
LFLAGS = /CO # controls linker options  
CODEVIEW = /Zi # controls debugging information  
  
OBSJ = sample.obj one.obj two.obj  
  
all : sample.exe  
  
sample.exe : $(OBSJ)  
    link $(LFLAGS) @<<sample.lrf  
$(OBSJ: =+^  
)  
sample.exe  
sample.map;  
<<KEEP  
  
sample.obj : sample.c sample.h common.h  
    CL $(CFLAGS) sample.c  
  
one.obj : one.c one.h common.h  
    CL $(CFLAGS) one.c  
  
two.obj : two.c two.h common.h  
    CL $(CFLAGS) two.c  
  
clean :  
    -del *.obj  
    -del *.map  
    -del *.lrf
```

Assume that this makefile is named `SAMPLE.MAK`. To invoke it, enter

```
NMAKE /F SAMPLE.MAK all clean
```

NMAKE builds `SAMPLE.EXE` and deletes intermediate files.

Here is how the makefile works. The `CFLAGS`, `CODEVIEW`, and `LFLAGS` macros define the default options for the compiler, linker, and inclusion of debugging information. You can redefine these options from the command line to alter or delete them. For example,

```
NMAKE /F SAMPLE.MAK CODEVIEW= CFLAGS= all clean
```

creates an `.EXE` file that does not contain debugging information.

The `OBSJ` macro specifies the object files that make up the executable file `SAMPLE.EXE`, so they can be reused without having to type them again. Their names are separated by exactly one space so that the space can be replaced with

a plus sign (+) and a carriage return in the link response file. (This is illustrated in the second example in “Substitution Within Macros” on page 677.)

The `all` pseudotarget points to the real target, `sample.exe`. If you do not specify any target on the command line, NMAKE ignores the `clean` pseudotarget but still builds `all` because `all` is the first target in the makefile.

The dependency line containing the target `sample.exe` makes the object files specified in `OBJS` the dependents of `sample.exe`. The command section of the block contains only link instructions. No compilation instructions are given since they are given explicitly later in the file. (You can also define an inference rule to specify how an object file is to be created from a C source file.)

The `link` command is unusual because the LINK parameters and options are not passed directly to LINK. Rather, an inline response file is created containing these elements. This eliminates the need to maintain a separate link response file.

The next three dependencies define the relationship of the source code to the object files. The `.H` (header or include) files are also dependents since any changes to them also require recompilation.

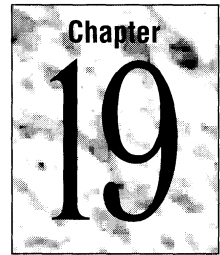
The `clean` pseudotarget deletes unneeded files after a build. The dash (-) command modifier tells NMAKE to ignore errors returned by the deletion commands. If you want to save any of these files, don't specify `clean` on the command line; NMAKE then ignores the `clean` pseudotarget.

18.12 NMAKE Exit Codes

NMAKE returns an exit code to the operating system or the calling program. A value of 0 indicates execution of NMAKE with no errors. Warnings return exit code 0.

Code	Meaning
0	No error
1	Incomplete build (issued only when /K is used)
2	Program error, possibly due to one of the following: <ul style="list-style-type: none">▪ A syntax error in the makefile▪ An error or exit code from a command▪ An interruption by the user
4	System error—out of memory
255	Target is not up-to-date (issued only when /Q is used)

Managing Libraries with LIB



This chapter describes the Microsoft Library Manager (LIB) version 3.20. LIB creates and manages standard libraries, which are used to resolve references to external routines and data during static linking.

19.1 Overview

LIB creates, organizes, and maintains standard libraries. Standard libraries are collections of compiled or assembled object modules that provide a common set of useful routines and data. You use these libraries to provide your program with the routines and data at link time; this is called static linking. After you have linked a program to a library, the program can use a routine or data item exactly as if it were included in the program.

With LIB you can create a library file, add modules to a library, and delete or replace them. You can combine libraries into one library file and copy or move a module to a separate object file. You can also produce a listing of all public symbols in the library modules.

LIB works with the following kinds of files:

- Object files in the Microsoft Relocatable Object-Module Format (OMF), which is based on the Intel 8086 OMF
- Standard libraries in Microsoft library format
- Import libraries created by the Microsoft Import Library Manager (IMPLIB)
- 286 XENIX archives and Intel-style libraries

There is a difference between an object file and an object module.

This chapter distinguishes between an “object file” and an “object module.” An object file is an independent file that can have a full path and extension (usually .OBJ). An object module is an object file that has been incorporated into a library. Object modules in the library have only base names. For example, SORT is an object-module name, while B:\RUN\SORT.OBJ is an object-file name.

19.2 Running LIB

To run LIB, type `LIB` at the operating system prompt and press `ENTER`. You can provide input to LIB in three ways, separately or in combination:

- Specify input on the command line.
- Respond to the prompts that LIB displays.
- Specify a response file that contains the expected input.

The LIB Command Line

You can run LIB and specify all the input it needs from the command line. The LIB command line has the following form:

```
LIB oldlibrary [[options]] [[commands]] [, [[listfile]] [, [[newlibrary]]]] [;]
```

Fields must appear in order but can be left blank (except for *oldlibrary*). A semicolon (;) after any field terminates the command; LIB assumes defaults for any remaining fields. The fields are described in “Specifying LIB Fields,” which begins on page 699.

To terminate the session at any time, press `CTRL+C`.

The following example instructs LIB to combine the object files `FIRST.OBJ` and `SECOND.OBJ` and to name the combined library `THIRD.LIB`:

```
LIB FIRST +SECOND, , THIRD
```

For a more detailed example of running LIB from the command line, see page 707.

LIB Command Prompts

If you do not specify all expected input on the command line and do not end the line with a semicolon, LIB asks you for the missing input by displaying four prompts. LIB waits for you to respond to each prompt and then asks for the next input. The responses you give to the LIB command prompts correspond to the fields on the LIB command line. The following list shows these correspondences:

```
Library name: oldlibrary [[options]]  
Operations: commands  
List file: listfile  
Output library: newlibrary
```

You can select default responses to the remaining prompts at any time by typing a single semicolon (;) followed immediately by a carriage return. The defaults for prompts are the same as the defaults for the corresponding command-line fields.

The following example specifies `THIRD` as the output library-file name at the prompt:

```
Output library: THIRD
```

For a more detailed example of how to use the LIB prompts, see page 707.

The LIB Response File

To run LIB without typing the full command line or responses to prompts, you can use a response file. You must first create a response file, which is a text file containing the command-line information; you can write and edit this file in PWB or use another editor. Then invoke LIB using the following command:

```
LIB @responsefile
```

The *responsefile* is the name of a text file containing some or all of the input expected by LIB. You can specify a full path with the filename. Precede it with an at sign (@).

You can also enter the name of the response file at any position in a command line or after any of LIB's prompts. The input from the response file is treated exactly as if it had been entered in the command line or after prompts. When you run LIB with a response file, LIB displays prompts followed by the input from the response file. If the response file does not contain all expected input and does not end with a semicolon, LIB prompts for the remaining responses.

Each input field in the response file must appear on a separate line or must be separated from other fields on the same line by a comma. A carriage-return and linefeed combination is equivalent to pressing ENTER in response to a prompt or to entering a comma in a command line. Input must appear in the same order as in the command-line fields or at the LIB prompts.

The following response file tells LIB to add the object files `CURSOR.OBJ` and `HEAP.OBJ` as the last two modules in `LIBFOR.LIB`:

```
LIBFOR  
+CURSOR +HEAP;
```

19.3 Specifying LIB Fields

For all three methods of input, LIB expects information to be specified in a definite order and organized into fields. This section describes the input fields in the order required by LIB. The fields are *oldlibrary*, *options*, *commands*, *listfile*, and *newlibrary*.

The Library File

The *oldlibrary* field specifies the name of an existing library or a library to be created. If you omit the extension, LIB assumes an extension of .LIB. You can specify a full path with the filename.

Important The path and filename cannot contain a dash character (–). LIB interprets the dash as the LIB “delete” operator.

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?
```

Press `Y` to create the file or `N` to terminate the library session. If the library name is followed immediately by commands, a comma, or a semicolon, LIB suppresses the message and assumes `Y`.

Performing Consistency Checks

If *oldlibrary* is followed immediately by a semicolon (;), LIB performs a consistency check on the specified library to see if 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. LIB puts the message in the listing file if one is created; otherwise, it writes the message to the standard output.

The following example causes LIB to perform a consistency check of the library file FOR.LIB if the library file exists.

```
LIB FOR;
```

No other action is performed. LIB displays any consistency errors it finds and ends the session. If FOR.LIB does not exist, LIB creates an empty library file with that name.

LIB Options

Options are not case sensitive and can appear only between the *oldlibrary* and *commands* fields on the command line or at the `Library Name:` prompt following

the *oldlibrary* specification. The option name must be preceded by a forward slash (/) as the option specifier. (Do not use a dash, as the option specifier. LIB interprets a dash as the “delete” operator.) Options can be abbreviated to the shortest unique name; the brackets show the optional part of the name. This chapter uses meaningful yet legal forms of the option names, which may be longer than the shortest unique names. LIB has the following options:

/H[[ELP]]

Calls the QuickHelp utility. If LIB cannot find the Help file or QuickHelp, it displays a brief summary of LIB command-line syntax.

/I[[GNORECASE]]

Tells LIB to ignore case when comparing symbols. LIB does this by default. Use the /NOI option to create a library that is marked as case sensitive.

Use /IGN when combining a case-sensitive library with others that are not case sensitive to create a new library that is not case sensitive. (See the /NOI option for more information.)

/NOE[[XTDICTIONARY]]

Prevents LIB from creating an extended dictionary of cross-references between modules. LINK uses the extended dictionary to speed up a library search. (LINK also has an option called /NOE, where /NOE means “do not read an extended dictionary.”)

Creating an extended dictionary requires more memory. If LIB reports the error message `no more virtual memory`, either use /NOE or build the library with fewer modules.

/NOI[[GNORECASE]]

Tells LIB to preserve case when comparing symbols. By default, LIB ignores case. Use /NOI when you have symbols that are the same except for case. (When LINK uses the library, it ignores case unless LINK’s /NOI option is specified.)

If a library is built with /NOI, the library is internally marked to indicate that case sensitivity is in effect. (Libraries for case-sensitive languages such as C are built with /NOI.) If you combine multiple libraries and any one of them is case sensitive, LIB marks the output library as case sensitive. To override this, use the /IGN option.

/NOL[[OGO]]

Suppresses the LIB copyright message.

/P[[AGESIZE]]:*number*

Specifies the page size of a new library or changes the page size of an existing library. The *number* specifies the new page size in bytes. It must be an integer power of 2 between 16 and 32,768. The default page size is 16 bytes for a new library or the current page size for an existing library. Combined libraries take the largest component page size.

The page size of a library sets the alignment of modules stored in the library. Modules start at locations that are a multiple of the page size from the beginning of the file. When creating a library, LIB builds a dictionary, which holds the locations of each name in each module. Each location value represents the number of pages in the file. Because of this addressing method, a library with a large page size can hold more modules than a library with a smaller page size.

The page size also determines the maximum possible size of the .LIB file. This limit is $number * 64K$. For example, `/PAGE:32` limits the .LIB file to 2 megabytes ($32 * 65,536$ bytes). However, for each module in the library, an average of $number/2$ 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.

/?

Displays a brief summary of LIB command-line syntax.

LIB Commands

The *commands* field specifies five operations for performing library-management tasks with LIB and manipulating modules: add, delete, replace, copy, and move. These commands can be used on the command line or in a response file in response to the `Operations: prompt`. To use this field, type a command operator followed immediately by a module name or an object-file name. You can specify more than one operation in this field in any order. If you leave the *commands* field blank, LIB does not make any changes to *oldlibrary*.

If you have many operations to perform during a library session, you can use an ampersand (&) to extend the operations line. Type the ampersand after a module name or filename; do not put the ampersand between an operator and a name. Immediately after the ampersand, press ENTER and then continue to type the rest of the command line. You can use this technique on the command line or in response to a prompt. When the ampersand is entered at a prompt, it tells LIB to repeat the `Operations: prompt`. In a response file, begin a new line of commands after the ampersand. See the examples at the end of this chapter for an illustration of the use of the ampersand.

You can perform one or more library-management functions during a LIB session. For each session, LIB determines whether a new library is being created or an existing library is being examined or modified. It then processes commands in the following order:

1. Deletion and move commands. LIB does not actually delete modules from the existing library 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. If there are no deletion or move commands, LIB creates the new file by copying the original library file. (The *newlibrary* field, described on page 706, controls what happens to the existing library.)

2. Addition commands. Like deletions, additions are not performed on the original library file. Instead, the additional modules are appended to the end of the new library file.

As LIB carries out these commands, it reads the object modules in the library and checks them for validity. It then builds a dictionary, an extended dictionary (unless /NOE is specified), and a listing file (if a *listfile* is specified). The listing file contains a list of all public symbols and the names of the modules in which they are defined.

Important Paths and filenames specified with these commands cannot contain a dash character (-). LIB interprets the dash as the LIB “delete” operator.

The Add Command (+)

Use the add command to create a library file, to add a module, or to combine libraries. The command has the form:

`+name`

where *name* is the name of the object file or library file. If no extension is specified, LIB assumes .OBJ. You can specify a path with the filename.

Creating a New Library Use the add command to create a new library from one or more object files. Specify the name of the new library in the *oldlibrary* field, then specify each object file’s name preceded by a plus sign. In the following example, LIB is instructed to create the library file FIRST.LIB containing the object module called MORE:

```
LIB FIRST +MORE;
```

Adding Library Modules Use the add command to add an object module to a library. Give the name of the object file to be added immediately following the plus sign. LIB adds object modules to the end of a library file.

LIB strips the drive, path, and extension from the object-file name and leaves only the base name. This becomes the name of the object module in the library. For example, if the object file B:\CURSOR.OBJ is added to a library file, the name of the corresponding object module is CURSOR.

In the following example, LIB is instructed to add the module MORE to the already existing library file FIRST.LIB:

```
LIB FIRST +MORE;
```

Combining Libraries To combine the contents of two libraries, supply the name of a library instead of an object file. In addition to standard libraries, LIB lets you combine import libraries (created by IMPLIB), 286 XENIX archives, and Intel-format libraries.

Specify the plus sign followed by the name of the library whose contents you wish to add to the original library. You must include the .LIB extension of the library name. Otherwise, LIB assumes that the file is an object file and looks for the file with an .OBJ extension.

LIB adds the modules of the new library to the end of the original library. 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. If you omit this field, LIB saves the combined library under the name of the original library, that is, the name given in the *oldlibrary* field. The original library is saved with the same base name and the extension .BAK.

The following example combines DRAW.LIB and CHART.LIB into a library with the filename GRAPHICS.LIB: LIB;commands

```
LIB DRAW +CHART.LIB, ,GRAPHICS
```

The Delete Command (-)

Use the delete command to delete an object module from a library. The command has the form:

-name

where *name* is the name of the module to be deleted. A module name does not have a path or extension; it is simply a name, such as CURSOR.

The following example tells LIB to delete the FLOAT module from the MATH.LIB library:

```
LIB MATH -FLOAT;
```

The Replace Command (-+)

Use the replace command to replace a module in the library. The command has the form:

-+name

where *name* is the name of the module to be replaced. A module name has no path and no extension. LIB deletes the given module and 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 directory.

The following three examples of command lines are equivalent. All three instruct LIB to replace the HEAP module in the library LANG.LIB. LIB deletes the HEAP module from the library and then appends the object file HEAP.OBJ as a new module in the library. Delete operations are always carried out before add operations, regardless of the order in which they are specified.

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

The Copy Command (*)

Use the copy command to copy a module from the library file into a newly created object file of the same name. The command has the following form:

**name*

where *name* is the name of the module to be copied. The module remains in the library file. LIB names the object file by using the base name of the module and adding an .OBJ extension. It then puts it in the current directory. You cannot override this filename or location; however, you can later rename the file and copy or move it to any location. LIB writes the full name of the object file (including drive, path to the current directory, base name, and extension) into the header of the object file.

The Move Command (-*)

Use the move command to move an object module from the library file to an object file. The command has the form:

*-*name*

where *name* is the name of the module to be moved. This operation is equivalent to copying the module to an object file using the copy command (*) and then deleting the module from the library using the delete command (-).

The Cross-Reference Listing

A cross-reference listing contains two lists in the following order:

1. An alphabetical list of all public symbols in the library. Each symbol name is followed by the name of the module in which it is defined.

2. A list of the modules in the library with the location and size of each. Under each module name is an alphabetical listing of the public symbols defined in that module.

Create a cross-reference listing by giving a name for the listing file in the *listfile* field of the command line or at the `List file:` prompt. To create it in a directory other than the current one, specify a full path for the listing file. LIB does not supply a default extension if you omit the extension. When you do not specify a filename, the default is the special file named NUL, which tells LIB not to create a listing.

The following example creates a listing called LCROSS.PUB. It does nothing else except perform a consistency check of the library file LANG.LIB.

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

The Output Library

The *newlibrary* field specifies a name for a changed library file. You can specify a full path with the filename. LIB does not supply a default extension if you omit the extension.

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 *commands* field of the command line or at the `Operations:` prompt are performed on that library.

LIB keeps both the unchanged library file and the newly changed version; it copies the library and makes changes to the copy. (This prevents the loss of your original file if you terminate LIB before the session is finished.) It names the two versions as follows:

- If you specify the name of a new library file in the *newlibrary* field, the modified library is stored under that name, and the original library is preserved under its own name.
- If you leave the field blank, LIB replaces the original library file with the changed version of the library and saves the original library file with the extension .BAK. Either way, at the end of a session you have two library files: the changed version and the original version.

Note You need enough space on your disk for both the original library file and the copy.

Examples

All the following examples instruct LIB to:

- Suppress the creation of an extended dictionary of cross-references.
- Delete the module HEAP from the library.
- Move the module STUFF from the library FIRST.LIB to an object file called STUFF.OBJ; the module STUFF is deleted from the library.
- Copy the module MORE from the library to an object file called MORE.OBJ; the module MORE remains in the library.
- Name the revised library SECOND.LIB. The new library contains all the modules in FIRST.LIB except STUFF and HEAP.
- Leave the original library, FIRST.LIB, unchanged.
- Create a cross-listing file called CROSSLST.

Command-Line Example

```
LIB FIRST /NOE -*STUFF *MORE &  
-HEAP, CROSSLST, SECOND
```

LIB Prompt Example

```
Library Name: FIRST /NOE  
Operations: -*STUFF *MORE &  
Operations: -HEAP  
List File: CROSSLST  
Output file: SECOND
```

Response-File Example

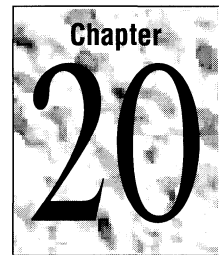
```
FIRST /NOE  
-*STUFF *MORE &  
-HEAP  
CROSSLST  
SECOND
```

19.4 LIB Exit Codes

LIB returns an exit code (also called return code or error code) to the operating system or the calling program. You can use the exit code to control the operation of batch files or makefiles.

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: <ul style="list-style-type: none">▪ There was insufficient memory.▪ An internal error occurred▪ The user interrupted the session.

Creating Help Files with HELPMAKE



This chapter describes how to create and modify Help files using the Microsoft Help File Maintenance Utility (HELMAKE) version 1.08. A “Help file” is a file that can be read by the Microsoft Advisor Help system and QuickHelp. If you’ve used the Programmer’s WorkBench (PWB) or one of the Microsoft Quick languages, you already know the advantages of the Microsoft Advisor. HELPMAKE extends these advantages by allowing you to customize the Microsoft Help files or create your own Help files.

HELMAKE translates Help source files to a Help database accessible within the following environments:

- Microsoft Programmer’s WorkBench (PWB)
- Microsoft QuickHelp utility
- Microsoft CodeView debugger
- Microsoft Editor version 1.02
- Microsoft QuickC compiler versions 2.0 and later
- Microsoft QuickBasic versions 4.5 and later
- Microsoft QuickPascal version 1.0
- Microsoft Word version 5.5
- MS-DOS EDIT version 5.0
- MS-DOS QBasic version 5.0

Warning The PWB editor breaks lines longer than about 250 characters. Some Help sources contain lines longer than this. To edit files that have long lines, you must either use an editor (such as Microsoft Word) that does not restrict line length or extend long lines using the backslash (\) line-continuation character.

20.1 Overview

HELPMAKE creates a Help file by encoding a source file. A Help file contains information that can be read by a Help reader (sometimes referred to in this chapter as an application). Examples of Help readers are the Microsoft Advisor or Microsoft QuickHelp. Help files have an .HLP extension.

Source files for HELPMAKE are text files that contain topic text along with attributes and commands that tell HELPMAKE how to process the file. HELPMAKE encodes text files written in the following formats: QuickHelp, rich text format (RTF), and minimally formatted ASCII.

Encoding compresses the text and translates the commands into information for the Help reader. You can control the amount of compression and other aspects of encoding.

HELPMAKE can also decode an existing Help file. Decoding decompresses the text into ASCII format. Attributes and commands can be preserved or omitted during decoding. You can modify an existing Help file by using HELPMAKE to decode the file and then rebuild it into a different Help file. You can even modify a Microsoft help file by decompressing it and then encoding it with your changes. Regardless of the source format, HELPMAKE always decodes a Help file into the QuickHelp format.

The basic unit of Help is the database. A Help database is an individual file created by HELPMAKE. At the time it is created, it is given an internal name that is the same as the filename on disk. If the file is later renamed, the database retains this internal name as it is known by HELPMAKE and the Help reader.

A Help system consists of one or more physical Help files that are available to a Help reader. A physical Help file is a file on disk with an .HLP extension. It can contain a single database (with either the same or a different filename) or multiple databases. To create a physical Help file that contains several Help databases, use the DOS COPY command. Specify the /b modifier to combine them as binary files. You can merge several databases into one physical Help file, combine two or more physical Help files, or append a Help database to an existing physical Help file. For example, the following command concatenates three individual Help databases into a new physical Help file:

```
COPY help1.hlp /b + help2.hlp /b + help3.hlp /b myhelp.hlp
```

The next example merges the database `yourhelp.hlp` with the existing Help file `utils.hlp`:

```
COPY utils.hlp /b + yourhelp.hlp /b
```

It is recommended that you back up existing Help files before running the COPY command. You may need to concatenate Help files if you reach a limit on physical files imposed by your system or the Help reader.

You can use HELPMMAKE to deconcatenate, or split, a physical Help file that contains multiple databases. If you want to decompress such a Help file, you must first split it and then decompress each database.

When designing a Help system, it is important to know that a single database is more efficient to search than multiple databases or physical Help files.

20.2 Running HELPMMAKE

The following sections describe HELPMMAKE syntax and options for encoding a Help file, decoding or deconcatenating a Help file, and getting Help on HELPMMAKE. Some options apply only to encoding, others apply only to decoding, and a few apply to both.

The following are some general rules for syntax:

- Options are not case sensitive. Precede each option with either a forward slash (/) or a dash (-).
- You can specify a path with a filename. Separate multiple filenames with spaces or tabs. Where multiple files can be specified, you can use wildcard characters (* and ?).

Encoding

To create a Help file, use the following syntax:

```
HELMMAKE /E[[n]] /Ooutfile options sourcefiles
```

The /E option encodes a Help source file and creates a compressed Help database. The *n* is a decimal number that specifies the type of compression. If *n* is omitted, HELPMMAKE compresses the file as much as possible (about 50 percent). The value of *n* is in a range from 0 through 15, which represents the following compression techniques:

Value	Technique
0	No compression
1	Run-length compression
2	Keyword compression
4	Extended keyword compression
8	Huffman compression

You can add these values to combine compression techniques. For example, specify `/E3` to get run-length and keyword compression. Use `/E0` to create the database quickly during the testing stages of database creation when you are not yet concerned with size.

The `/O` option specifies a filename for the database. This option is required when encoding.

Additional options are discussed in the next section and in “Other Options” on page 715.

The `sourcefiles` field specifies one or more text files that contain Help source information.

Options for Encoding

The following options control encoding:

`/Ac`

Specifies *c* as a control character for the Help database. A control character marks a line that contains special information for internal use by the Help reader. Control characters differ for each Help reader. For example, the Microsoft Advisor uses a colon (`:`) to indicate a command, so you must specify `/A:` when building a Help file for use with the Advisor. HELPMAKE assumes `/A:` if the `/T` option is specified.

`/C`

Makes context strings case sensitive.

`/Kfilename`

Optimizes keyword compression by supplying a list of characters to act as word separators. The *filename* is a text file that contains a list of separator characters.

HELMAKE can apply “keyword compression” to words that occur often enough to justify replacing them with shorter character sequences. A “word” is any series of characters that do not appear in the separator list. The default separator list includes all ASCII characters from 0 to 32, ASCII character 127, and the following characters:

```
!"#&'()*+,-./:;<=>?@[\\]^_`{|}~
```

You can improve keyword compression by designing a separator list tailored to a specific Help file. For example, a number sign (`#`) is treated as a separator by default. However, in a Help file about the C language, you might want to have HELPMAKE treat each directive such as `#include` as a keyword instead of as a separator followed by a word. To encode `#include` and other directives as keywords, create a separator list that omits the number sign:

```
! " & ' ( ) * + - , / : ; < = > ? @ [ \ ] ^ _ { | } ~
```

ASCII characters in the range from 0 through 31 are always separators, so you do not need to list them. However, a customized list must include all other separators, including the space (ASCII character 32). If you omit the space, HELPMAKE will not use spaces as word separators.

/L

Locks the Help file so that it cannot be decoded later.

/Sn

Specifies the type of input file, according to the following *n* values:

Option	File Type
<i>/S1</i>	Rich text format (RTF)
<i>/S2</i>	QuickHelp (the default)
<i>/S3</i>	Minimally formatted ASCII

/T

Translates dot commands into internal format. If your source file contains dot commands other than **.context** and **.comment**, you must supply this option. The */T* option is required if you want to use commands in the QuickHelp dot format. Dot commands are described on page 722. HELPMAKE assumes the */A:* option if */T* is specified.

/Wwidth

Sets the fixed width of the resulting Help text in number of characters. The *width* is a decimal number in a range from 11 through 255. If */W* is omitted, the default width is 76. When encoding an RTF source (*/S1*), HELPMAKE wraps the text to *width* characters. When encoding QuickHelp (*/S2*) or minimally formatted ASCII (*/S3*) files, HELPMAKE truncates lines to this width.

Example

The following example invokes HELPMAKE with the */V*, */E*, and */O* options:

```
HELMMAKE /E /V /Omy.hlp my.txt > my.log
```

HELMMAKE reads input from the source file `my.txt` and creates the compressed Help database `my.hlp`. The */E* option, without a compression specification, maximizes compression. The DOS redirection symbol (`>`) sends a log of HELPMAKE diagnostic information to the file `my.log`. You may want to redirect the output to a file when using */V* because the verbose mode can generate a lengthy log.

Decoding

To decode a Help file, use the following syntax:

```
HELMMAKE /D[[c]] [[/Ooutfile]] options sourcefiles
```

The `/D` option decodes a Help file or splits a concatenated file into its component databases. The `/D` option can take a qualifying character *c*, which is either `S` or `U`.

Specify `/D` without a qualifying character to fully decode a database into a text file that is in QuickHelp format, with all links and formatting information intact. If the physical Help file contains concatenated databases, only the first database is decoded.

Specify `/DU` to decompress the database and remove all screen formatting and links. If the physical Help file contains concatenated databases, only the first database is decoded.

Specify `/DS` to split (deconcatenate) a physical Help file that contains one or more databases. HELPMAKE creates a physical Help file for each database in the original Help file. The Help file is not decompressed. HELPMAKE names the deconcatenated files using the names of the databases. The deconcatenated files are placed in the current directory. If a database in the file has a name that matches the name of the original physical Help file, HELPMAKE issues an error. In this case, rename the physical Help file, or run HELPMAKE in another directory and specify a path with the source file. Do not use the `/O` option with `/DS`.

The `/O` option specifies a filename for the decoded file. If `/O` is not specified, HELPMAKE sends the text to standard output. This option is not valid when using `/DS`.

There is one option available to control decoding. The `/T` option translates commands from internal format to dot-command format. This option applies only when using `/D`. It is recommended to always use this option to make the resulting source file more readable.

Additional options are discussed in “Other Options” on page 715.

The *sourcefiles* field specifies one or more physical Help files.

Example

The following example decodes the Help file `my.hlp` into the source file `my.src`:

```
HELPMAKE /D /T /Omy.src my.hlp
```

Getting Help

To get help on HELPMAKE, use the following syntax:

```
HELPMAKE {/H[[ELP]] | /?}
```

The following are the options for Help:

`/?`

Displays a brief summary of the HELPMAKE command-line syntax and exits without encoding or decoding any files. All other information on the command line is ignored.

`/H[[ELP]]`

Calls the QuickHelp utility and displays Help about HELPMAKE. If HELPMAKE cannot find QuickHelp or the Help file, it displays the same information as with the `/?` option. No files are encoded or decoded. All other information on the command line is ignored.

Other Options

The following options apply whether encoding or decoding.

The `/NOLOGO` option

The `/NOLOGO` option suppresses the HELPMAKE copyright message.

The `/V` option

The `/V` option controls the verbosity of diagnostic and informational output. HELPMAKE sends this information to standard output. The syntax for `/V` is:

`/V[[n]]`

Specify `/V` without *n* to get a full output. The decimal number *n* controls the amount of information produced. Numbers in a range from 0 through 3 are valid only for decoding. The values of *n* are:

Option	Output
<code>/V</code>	Maximum diagnostic output
<code>/V0</code>	No diagnostic output and no banner
<code>/V1</code>	HELPMAKE banner only
<code>/V2</code>	Pass names
<code>/V3</code>	Context strings encountered on first pass
<code>/V4</code>	Context strings encountered on each pass
<code>/V5</code>	Any intermediate steps within each pass
<code>/V6</code>	Statistics on Help file and compression

20.3 Source File Formats

You can create Help source files for HELPMAKE in any of three formats. The QuickHelp format is the default format for encoding. When Help databases are decoded, the resulting text files are always in QuickHelp format. The discussion that follows uses QuickHelp format to describe how to create a Help source file. Later sections describe the two other formats: rich text format (RTF) and minimally formatted ASCII.

Rich text format is a Microsoft word-processing format that is supported by several word processors, including Microsoft Word version 5.0 and later and Microsoft Word for Windows. For more information, see “Rich Text Format” on page 725.

Minimally formatted ASCII files define contexts and their topic text. They cannot contain formatting commands or explicit links. For more information, see “Minimally Formatted ASCII” on page 728.

In addition to these three formats, you can link to unformatted ASCII files from within a Help database. Unformatted ASCII files are text files with no formatting commands, context definitions, or special information. You do not process unformatted ASCII files with HELPMAKE. An unformatted ASCII file does not become a database or part of a physical Help file. The file’s name is used as the object of a link. For example, you can create a link to an include file or a program example. Any word that is an implicit link in other Help files is also an implicit link in unformatted ASCII files.

A Help system can use any combination of files with different format types.

20.4 Elements of a Help Source File

The following sections describe how to create the fundamental elements of a Help file.

Defining a Topic

A Help source file is a text file that consists of a sequence of topics. A topic is the fundamental unit of Help information. It is usually a screen of information about a particular subject.

Each topic begins with one or more consecutive **.context** statements or definitions. The topic consists of all subsequent lines up to the next **.context** statement. A context definition associates the topic with a “context string,” which is the word or phrase for which you want to be able to request Help. When Help is requested on

a context string, the Help reader displays the topic. A context definition has the following form:

```
.context string
```

The **.context** command defines a context string for the topic that follows it. A context string can contain one word or several words depending on the Help reader and the delimiters it understands. For example, because Microsoft QuickBasic considers spaces to be delimiters, a context string in a QuickBasic Help file is limited to a single word. Other applications, such as PWB, can handle context strings that span several words. In either case, the application hands the context string to an internal “Help engine” that searches the database for a topic that is marked with the requested context string.

For example, the following line introduces Help for the **#include** directive:

```
.context #include
```

A topic can be associated with more than one context string. For example, the C-language functions **strtod**, **strtol**, **_strtold**, and **strtoul** are described in a single topic, and each is defined in a separate **.context** command for that topic, as follows:

```
.context strtod  
.context _strtold  
.context strtol  
.context strtoul
```

Warning HELPMAKE warns you if it encounters a duplicate context definition within a given Help source file. Each context string must be unique within a database. You cannot associate a single context string with several topics in a single database.

A context string can be global or local. The *string* for a local context is preceded by an at sign (@). For more information, see “Local Contexts” on page 720.

Creating Links to Other Topics

A topic can contain a link to another topic. Links let you navigate a Help database. When a topic is displayed, you can ask for Help on links contained in the topic. These links can be associated with other contexts in the same Help database, contexts in other Help databases, or even ASCII files on disk. You can view the cross-referenced material immediately by activating the link without having to search the Help system’s indexes and tables of contents for the topic.

The keystroke that activates a link depends on the application. Consult the documentation for each product for the various ways to get Help on a link. In Microsoft

language products, use ENTER, SPACEBAR, or F1. If the file that contains the link's destination is not already open, the Help reader finds it and opens it.

The topic text can present the link in various ways, depending on how you want to design your Help system. The link can appear as a "See:" cross-reference, for example, or as a button that contains a title surrounded by special characters. It can even be undistinguished from surrounding text.

A link is either explicit (coded) or implicit (available without coding). It is associated with either a global context (visible throughout the Help system) or a local context (visible only in one database). The following sections discuss these features of links.

Explicit Links

An explicit link is a word or phrase coded with invisible text that provides the context to which the link refers or the action which the Help reader is to take. Use the `\v` formatting attribute to delimit the invisible text. Format the explicit link in the source file using the following syntax:

```
string\vtex\v
```

If *string* consists of more than one word, you must anchor the string with the `\a` formatting attribute as follows:

```
\astring\vtex\v
```

An anchored link must be specified entirely on one line.

The `\v` attributes surround the invisible *text*, which is one of the following commands to the Help reader:

contextstring

Display the topic associated with *contextstring* when the link is activated. The context string must be available either as a local context in the same Help database or as a global context anywhere in the Help system. For a discussion of global and local contexts, see "Local Contexts" on page 720.

helpfile!contextstring

Search *helpfile* for *contextstring* and display the topic associated with it. Only the specified Help database or physical Help file is searched for the context. Since *helpfile* is not in the local database, *contextstring* must be a global context. Use this specification to confine the search to a single database if a context is contained in more than one database and you want only one of the topics to be found.

filename!

Display *filename* as a single topic. The specified file must be a text file no larger than 64K.

!command

Execute the command specified after the exclamation point (!). The command is case sensitive. Commands are application-specific. For example, in the Microsoft Advisor and QuickHelp, the command !B represents the previously accessed topic.

In the following example, the word `Example` is an explicit link:

```
\bSee also:\p Example\vopen.ex\v
```

The `\v` formatting attribute marks the explicit link in the Help text. The `\b` and `\p` are formatting attributes that mark `See also:` as bold text. (Formatting attributes are described on page 721.) The link refers to `open.ex`. On the screen, this line appears as follows:

```
See also: Example
```

If you select any letter in `Example` and request Help, the Help reader displays the topic whose context is `open.ex`.

To create an explicit link that contains more than one word, you must use an anchor, as in the following example:

```
\bSee also:\p \aExample 1\vopen.ex1\v, \aExample 2\vopen.ex2\v
```

The `\a` attribute creates an anchor for the explicit link. The phrase following the `\a` attribute refers to the context specified in the invisible text. The first `\v` attribute marks both the end of the anchored string and the beginning of the invisible text. The second `\v` attribute ends the invisible text. The anchored link must fit on one line.

Implicit Links

An implicit link is a single word for which a global context exists somewhere in the Help system. Any word that appears as a global context is implicitly linked. You do not code the word to create the link. When you ask for Help on a word that exists as an implicit link, the Help reader displays the topic that has a **.context** string that matches the selected word.

For example, suppose that the Help database contains a screen that starts with:

```
.context open
```

If you ask for Help on the word “open” (using the features for requesting Help that are available in your Help reader), the topic that begins with `.context open` is displayed. An explicit link to the topic is not necessary. For example, in PWB you can place the cursor on the word “open” as it appears in your source file or in a

displayed Help topic, then click the right mouse button or press F1. Thus, every occurrence of “open” is a potential implicit link.

Local Contexts

A “local context” is a context string that begins with an at sign (@). Local contexts use less file space and speed access. However, a local context has meaning only within the database in which it appears.

HELPMAKE encodes a local context as an internally generated number rather than a context string. This saves space in the database. Unlike a global context (a context string that is specified without the preceding @), a local context is not stored as a string. Thus, topics headed by local contexts can only be accessed using explicit links and cannot be accessed from another database. Local contexts are not restored as strings when a database is decompressed.

The following source file contains two topics, one marked with a global context and one marked with a local context:

```
.context Global
    This is a topic that is marked with a global context.
    It is accessed using the context string "Global". It
    contains a link to a topic marked with a local context.
    See: \aA Local Topic\@Local\
.context @Local
    This topic can be reached only by the explicit link in
    the previous topic (or by sequentially browsing the file).
```

The text A Local Topic is explicitly linked to @local, which is a local context. If the user asks for Help on the text or scrolls through the Help file, the Help reader displays the topic text that follows the context definition for @local. This topic cannot be accessed any other way (except by sequentially browsing the database).

If you want a topic to be accessible in both local and global contexts, mark the topic text with both global and local **.context** statements:

```
.context Global
.context @Local
    This is a topic that is marked with a global context and
    a local context. It can be accessed using the context
    string "Global" (as an explicit or implicit link) or the
    context string "@Local" (as an explicit link only). (It
    can also be reached by sequentially browsing the file).
```

Both **.context** statements must appear together, immediately before the topic text they are to be associated with.

To create a context that begins with a literal @, precede it with a backslash (\).

Formatting Topic Text

You can use formatting attributes to control the appearance of the text on the screen. Using these attributes, you can make words appear in various colors, inverse video, and so forth, depending on the application and the capabilities of your display. This is useful, for example, to distinguish explicit links in the text.

Each formatting attribute consists of a backslash (\) followed by a character. Table 20.1 lists the formatting attributes.

Table 20.1 Formatting Attributes

Formatting Attribute	Action
\ a	Anchors text for explicit links
\ b, \B	Turns bold on or off
\ i, \I	Turns italics on or off
\ p, \P	Turns off all attributes
\ u, \U	Turns underlining on or off
\ v, \V	Turns invisibility on or off (hides explicit links)
\\	Inserts a single backslash in text

On color monitors, text labeled with the bold, italic, and underline attributes is translated by the application into suitable colors, depending on the user's default color selections. On monochrome monitors, the text's appearance depends on the application.

The \b, \i, \u, and \v options are toggles; they turn their respective attributes on or off. You can use several of these on the same text. Use the \p attribute to turn off all attributes except \v. Use the \v attribute to hide explicit links in the text. Explicit links are discussed on page 718.

Only visible characters count toward the character-width limit specified with the /W command-line option. Lines that begin with an application-specific control character are truncated to 255 characters regardless of the width specification. For more information on truncation and application-specific control characters, see "Options for Encoding" on page 712.

In the following example, \b initiates bold text for Example 1, and \p changes the remaining text to plain text:

```
\bExample 1\p This is a bold head for the first example.
```

Dot Commands

Dot commands identify topics and convey other topic-related information to the Help reader.

The most important dot command is the **.context** command, described in “Defining a Topic” on page 716. Every topic begins with one or more **.context** commands. Each **.context** command defines a context string for the topic. You can define more than one context for a single topic, as long as you do not place any topic text between the context definitions.

Most dot commands have an equivalent colon command, which consists of a colon (:) followed by a character. If you decode a database without using /T, commands in the database are shown as colon commands. You can use both colon commands and dot commands in the same source file. If you use any dot commands other than **.context** or **.comment**, you must supply the /T option when encoding.

Table 20.2 lists the dot commands. Some commands are not supported by all Help readers.

Table 20.2 Dot Commands

Dot Command	Colon Command	Action
.category <i>string</i>	:c	Lists the category in which the current topic appears and its position in the list of topics. The category name is used by the QuickHelp Categories command, which displays the list of topics. Supported only by QuickHelp.
.command	:x	Indicates that the topic cannot be displayed. Use this command to hide command topics and other internal information.
.comment <i>string</i> .. <i>string</i>	none	The <i>string</i> is a comment that appears only in the source file. Comments are not inserted in the database and are not restored during decoding.
.context <i>string</i>	none	The <i>string</i> defines a context.
.end	:e	Ends a paste section. See the .paste command. Supported only by QuickHelp.
.execute	:y	Executes the specified command. For example, .execute Pmark context represents a jump to the specified context at the specified mark. See the .mark command.
.freeze <i>numlines</i>	:z	Locks the first <i>numlines</i> lines at the top of the screen. These lines do not move when the text is scrolled.

Table 20.2 (continued)

Dot Command	Colon Command	Action
.length <i>topiclength</i>	:l	Sets the default window size for the topic in <i>topiclength</i> lines.
.line <i>number</i>	none	Tells HELPMAKE to reset the line number to begin at <i>number</i> for subsequent lines of the input file. Line numbers appear in HELPMAKE error messages. See .source . The .line command is not inserted in the Help database and is not restored during decoding.
.list	:i	Indicates that the current topic contains a list of topics. Help displays a highlighted line; you can choose a topic by moving the highlighted line over the desired topic and pressing ENTER. If the line contains a coded link, Help looks up that link. If it does not contain a link, Help looks within the line for a string terminated by two spaces or a newline character and looks up that string. Otherwise, Help looks up the first word.
.mark <i>name</i> [[<i>column</i>]]	:m	Defines a mark immediately preceding the following line of text. The marked line shows a script command where the display of a topic begins. The <i>name</i> identifies the mark. The <i>column</i> is an integer value specifying a column location within the marked line. Supported only by QuickHelp.
.next <i>context</i>	:>	Tells the Help reader to look up the next topic using <i>context</i> instead of the topic that physically follows it in the file. You can use this command to skip large blocks of .command or .popup topics.
.paste <i>pastename</i>	:p	Begins a paste section. The <i>pastename</i> appears in the QuickHelp Paste menu. Supported only by QuickHelp.
.popup	:g	Tells the Help reader to display the current topic as a pop-up window instead of as a normal, scrollable topic. Supported only by QuickHelp.
.previous <i>context</i>	:<	Tells the Help reader to look up the previous topic using <i>context</i> instead of the topic that physically precedes it in the file. You can use this command to skip large blocks of .command or .popup topics.


```

\iItalic\p      \i
\bBold\p        \b
\uUnderline\p  \u

```

The visual appearance of each attribute or combination of attributes is determined by the application that displays the help.

```
\bSee:\p
```

Coding, Expressions, Grammar, Keywords, Syntax

```
\i\p\aFlow Control\v@flow\v\i\p
```

```
\i\p\Release Notes\v$DOC:README.DOC!\v\i\p
```

```
.context @flow
```

```
.topic Sample Help: Flow Control
```

```
.length 8
```

```
.freeze 3
```

```
\i\p\Back\v!B\v\i\p
```

Here's another sample help screen.

```
This is an explicit link: \i\p\Sample\v@Sample\v\i\p
```

```
This is an implicit link: Sample
```

20.5 Other Help Text Formats

There are two other Help text formats you can use to create a Help database: rich text format (RTF) and minimally formatted ASCII. These formats are described in the next two sections.

Rich Text Format

Rich text format (RTF) is a Microsoft word-processing format supported by several word processors, including Microsoft Word version 5.0 and later and Microsoft Word for Windows. RTF is an intermediate format that allows documents to be transferred between applications without loss of formatting. You can use RTF to simplify the transfer of help files from one format to another. Like QuickHelp files, RTF files can contain formatting attributes and links.

As with the other text formats, each topic in an RTF source file consists of one or more context strings followed by topic text. The Help delimiter (>>) at the beginning of any paragraph marks the beginning of a new Help entry. The text that follows on the same line is defined as a context for the topic. If the next paragraph also begins with the Help delimiter, it also defines a context string for the same

topic. You can define any number of contexts for one topic. The topic text comprises all subsequent paragraphs up to the next paragraph that begins with the Help delimiter.

All QuickHelp dot commands, except **.context** and **.length**, can be used in RTF files. Each command must appear in a separate paragraph.

There are two ways to create an RTF file. The easiest way is to use a RTF word processor. RTF files usually contain additional information that is not visible to the user; HELPMAKE ignores this extra information.

You can also use an ordinary text editor to insert RTF codes manually. Utility programs exist that convert text files in other formats to RTF format. For more information on RTF, see the your Microsoft Word for Windows documentation.

Using a Word Processor

In an RTF-compatible word processor, enter the text and format it as you want it to appear: bold, underlined, hidden, and italic. You can also format paragraphs by selecting body and first-line indenting. Choose a monospace font and set the margin to the /W value you plan to encode the database with. The only item you need to insert into an RTF file manually is the Help delimiter (>>) followed by the context string that starts each entry. If you use dot commands, place each in its own paragraph.

When you have entered and formatted the text, save it in RTF format. In Microsoft Word version 5.5, for example, choose Save As from the File menu, then select RTF under Format.

You cannot see the RTF formatting codes when you load an RTF file into a compatible word processor. The word processor displays the text with the specified attributes. However, you can view these codes by loading an RTF file into a text editor or word processor.

Manually Inserting RTF Formatting Codes

RTF uses braces ({ }) for nesting. Thus, the entire file is enclosed in braces, as is each specially formatted text item.

When you manually insert RTF codes, you must delimit each dot command with the \par code. (An RTF editor or word processor inserts “\par” at the beginning and end of a paragraph.) For example, to use the **.popup** command, write:

```
\par.popup\par
```

HELMMAKE recognizes the subset of RTF codes listed in Table 20.3.

Table 20.3 RTF Formatting Codes

RTF Code	Action
<code>\b</code>	Bold. The Help reader decides how to display this; often it is intensified text.
<code>\fin</code>	Paragraph first-line indent, <i>n</i> twips.*
<code>\i</code>	Italic. The application decides how to display this; often it is reverse video.
<code>\lin</code>	Paragraph indent from left margin, <i>n</i> twips.*
<code>\line</code>	New line (not new paragraph).
<code>\par</code>	End of paragraph.
<code>\pard</code>	Default paragraph formatting.
<code>\plain</code>	Default attributes. On most screens, this is nonblinking normal intensity.
<code>\tab</code>	Tab character.
<code>\ul</code>	Underline. The application decides how to display this attribute; some adapters that do not support underlining display it as blue text.
<code>\v</code>	Hidden text. Hidden text is used for explicit links; it is not displayed.

*A “twip” is 1/20 of a point or 1/1440 of an inch. One space is approximately 180 twips.

Encoding RTF with HELPMMAKE

When HELPMMAKE compresses an RTF file, it formats the text to the width given by the `/W` option and ignores the paragraph formats.

When HELPMMAKE encodes RTF, any text between an RTF code and invisible text becomes an explicit link. This is illustrated in the following example:

```
{\b Formatting table}{\v printf.ex}
```

The string `Formatting table` is displayed in bold and is part of an explicit link to `printf.ex`.

Example

The following example is in RTF format:

```
{\rtf1
\pard\plain >>Sample
\par >@Sample
\par .topic Sample Help Topic
\par .freeze 3
\par \pard \li8000 {\i }{\b Back}{\v !B}{\i }
\par \pard -----
\par
```

```
\par \pard \li360 Help can contain text with three attributes:
\par \pard
\par \pard \li360 {\b Attribute}      {\b QuickHelp Code}
\par \pard
\par \pard \li360 {\i Italic}          \i
\par \pard \li360 {\b Bold}           \b
\par \pard \li360 {\u Underline}     \u
\par \pard
\par \pard \li360\ri720 The visual appearance of each attribute
or combination of attributes is determined
by the application that displays the help.
\par \pard
\par \pard \li360 {\b See:}
\par \pard
\par \pard \li360 Coding, Expressions, Grammar, Keywords, Syntax
\par \pard \li360 {\i }{\b Flow Control}{\v @flow}{\i }
\par {\i }{\b Release Notes}{\v $DOC:README.DOC!}{\i }
\par \pard >@flow
\par .topic Sample Help: Flow Control
\par .freeze 3
\par \pard \li8000 {\i }{\b Back}{\v !B}{\i }
\par \pard -----
\par
\par \pard \li360 Here's another sample help screen.
\par
\par \pard \li360 This is an explicit link: {\i }{\b Sample}{\v
@Sample}{\i }
\par \pard \li360 This is an implicit link: Sample
\par
}
```

Minimally Formatted ASCII

Minimally formatted ASCII files define contexts and their topic text. The Help information is displayed exactly as it appears in the file. A minimally formatted ASCII file cannot contain screen-formatting commands or explicit links. Any formatting codes are treated as ASCII text. Minimally formatted ASCII files have a fixed width.

A minimally formatted ASCII file contains a sequence of topics, each preceded by one or more context definitions. Each context definition must be on a separate line that begins with a help delimiter (>>). The topic consists of all subsequent lines up to the next context definition.

Implicit links work the same way they do in the other formats. A word in the Help text is an implicit link if it exists as a context somewhere in the Help system.

There are two ways to use a minimally formatted ASCII file. You can compress it with HELPMAKE and create a Help database, or a Help reader can access the uncompressed file directly. A Help reader can search a minimally formatted ASCII file faster if it has been compressed.

The following example coded in minimally formatted ASCII shows the same text as the QuickHelp and RTF examples presented elsewhere in this chapter:

```
>>Sample
-----[ Sample Help Topic ]-----
  Help can contain text with three attributes:

  Attribute      QuickHelp Code

  Italic         \i
  Bold           \b
  Underline      \u

  The visual appearance of each attribute
  or combination of attributes is determined
  by the application that displays the help.

  See:

  Coding, Expressions, Grammar, Keywords, Syntax

>>Coding
-----[ Sample Help: Coding ]-----

  Here's another sample help screen.
```

20.6 Context Prefixes

Microsoft Help databases use several context prefixes. A context prefix is a single letter followed by a period. It appears before a context string and has a predefined meaning. You may see these contexts in the resulting text file when you decode a Microsoft help database.

Context prefixes are used internally by Microsoft.

The context prefixes shown in Table 20.4 are used by Microsoft to mark product-specific features. They appear in decompressed databases. However, you do not need to add them to the files you write.

Table 20.4 Microsoft Product Context Prefixes

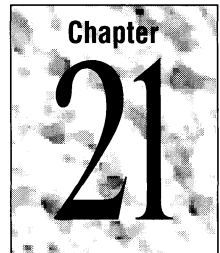
Prefix	Purpose
d.	Dialog box. The context string for the Help on a dialog box is <code>d.</code> followed by the number assigned to that dialog box.
e.	Error number. If a product supports the error numbering used by Microsoft languages, it displays Help for each error using this prefix.
h.	Help item. The context string for miscellaneous Help is <code>h.</code> followed by an assigned string. These strings are described in Table 20.5. For example, most Help readers look for the context string <code>h.contents</code> when Contents is chosen from the Help menu.
m.	Menu item. The strings that can follow <code>h.</code> are defined by the access keys for the product's menu items. For example, the Exit command on the File menu is accessed by ALT+F, X. The context string for Help on the command is <code>m.f.x.</code>
n.	Message number. The context string for the Help on a message box is <code>n.</code> followed by the number assigned to that message box.

You can use the `h.` prefix, shown in Table 20.5, to identify standard Help-file contexts. For instance, **h.default** identifies the default Help screen (the screen that usually appears when you select top-level Help).

Table 20.5 Standard h. Contexts

Context	Description
h.contents	The table of contents for the Help file. You should also define the string "contents" for direct reference to this context.
h.default	The default Help screen, typically displayed when the user presses SHIFT+F1 to get the "top level" in some applications.
h.index	The index for the Help file. You can also define the string "index" for direct reference to this context.
h.pg1	The Help text that is logically first in the file. This is used by some applications in response to a "go to the beginning" request made within the Help window.
h.title	The title of the Help database.

Browser Utilities



This chapter describes two utilities:

- Microsoft Browser Database Maintenance Utility (BSCMAKE) version 2.00
- Microsoft Browse Information Compactor (SBRPACK) version 2.00

These utilities build a browser database for use with the Microsoft Source Browser, a feature of the Microsoft Programmer's Workbench (PWB). As a navigation tool, the browser gives you the means to move around quickly in a large project and find pieces of code in your source and include files. As an interactive program database, the browser can answer questions about where functions are invoked or where variables and types are used. The browser can also generate useful outlines, call trees, and cross-reference tables.

Read this chapter to understand the Browse Option choices in PWB ...

When you tell PWB to create a browser database (.BSC file) for the program you are building, PWB automatically calls BSCMAKE. You do not need to know how to run BSCMAKE to create your database in PWB. However, you may want to read this chapter to understand the PWB options available to modify the database. For information on how to create and use a browser database in PWB, see "Using the Source Browser" in Chapter 5.

... or to learn how to build or maintain a browser database outside of PWB.

If you build your program outside of PWB, you can still create a custom browser database that you can examine with the Browser in PWB. Run the BSCMAKE utility to build the database from .SBR files created during compilation. You might need to run SBRPACK to provide more efficiency during the build. This chapter describes how to use both these utilities to create your browser database. For further information, see "Building Databases for Non-PWB Projects" in Chapter 5.

Note BSCMAKE is the successor to the Microsoft PWBRMAKE Utility. To allow existing makefiles to remain compatible, a file called PWBRMAKE.EXE is provided with BSCMAKE. This version of PWBRMAKE calls BSCMAKE using the arguments and options specified on the PWBRMAKE command line.

21.1 Overview of Database Building

BSCMAKE can build a new database from newly created .SBR files. It can also maintain an existing database using .SBR files for object files that have changed since the last build. The following sections describe how .SBR files are created, what you need to know to build a database, and how you can make the database-building process more efficient.

Preparing to Build a Database

The input files for BSCMAKE are .SBR files that you create when you compile or assemble your source files. When you build or update your browser database, all .SBR files for your project must be available on disk. To create an .SBR file, specify the appropriate command-line option to the compiler or assembler (shown in parentheses below). The following products generate .SBR files:

- Microsoft C Compiler version 6.00 or later (/FR or /Fr)
- Microsoft MASM version 6.00 or later (/FR or /Fr)
- Microsoft FORTRAN version 5.10 or later (/FR or /Fr)
- Microsoft Basic version 7.10 or later (/FBr or /FBx)
- Microsoft COBOL version 4.00 or later (BROWSE)

The options /FR, /FBx, and BROWSE put all possible information into the .SBR file. The options /Fr and /FBr omit local symbols from the .SBR file. (If the .SBR file was created with all possible information, you can still omit local symbols by using BSCMAKE's /El option; see page 736.)

Database building can be more efficient if the .SBR files are first packed by SBRPACK. The Microsoft C Compiler version 7.00 automatically calls SBRPACK when it creates an .SBR file. (If you want to prevent packing, specify the /Zn option.) Before you run BSCMAKE, you may want to run SBRPACK on any .SBR files that were not previously packed. See "SBRPACK" on page 739.

You can create an .SBR file without performing a full compile. For example, you can specify CL's /Zs option to perform a syntax check and still generate an .SBR file if you specify /FR or /Fr.

How BSCMAKE Builds a Database

BSCMAKE builds or rebuilds a database in the most efficient way it can. To avoid some potential problems, it is important to understand the database-building process.

BSCMAKE changes .SBR files to zero-length files.

When BSCMAKE builds a database, it truncates the .SBR files to zero length. During a subsequent build of the same database, a zero-length (or empty) .SBR file tells BSCMAKE that the .SBR file has no new contribution to make. It lets BSCMAKE know that an update of that part of the database is not required and an incremental build will be sufficient. During every build (unless the /n option is specified), BSCMAKE first attempts to update the database incrementally by using only those .SBR files that have changed.

BSCMAKE attempts an incremental build before it runs a full build.

BSCMAKE looks for a .BSC file that has the name specified with the /o option (described on page 736); if /o is not specified, BSCMAKE looks for a file that has the base name of the first .SBR file and a .BSC extension. If the database exists, BSCMAKE performs an incremental build of the database using only the contributing .SBR files. If the database does not exist, BSCMAKE performs a full build using all .SBR files.

Requirements for a Full Build

For a full build to succeed, all specified .SBR files must exist and must not be truncated. If any .SBR file is truncated, you must first rebuild it (by recompiling) before running BSCMAKE.

Requirements for an Incremental Build

For an incremental build to succeed, the .BSC file must exist. All contributing .SBR files, even empty files, must exist and must be specified on the BSCMAKE command line. If you omit an .SBR file from the command line, BSCMAKE removes its contribution from the database.

Methods for Increasing Efficiency

The database-building process can require large amounts of time, memory, and disk space. However, there are several ways to reduce these requirements.

Managing Memory Under DOS

Building a database uses a lot of memory. Large projects benefit the most from use of the Source Browser, but under DOS their large size can cause BSCMAKE to run out of memory. There are several ways to run BSCMAKE under DOS that make use of virtual memory and extended memory. The commands to run these forms of BSCMAKE are described in “System Requirements for BSCMAKE” on page 734.

Making a Smaller Database

Smaller databases take less time to build, use up less space on disk, have a lower risk of causing BSCMAKE to run out of memory, and run faster in the browser. The following list gives some methods of making a smaller database:

- Use BSCMAKE options to exclude information from the database. These options are described on page 736.
- Omit local symbols in one or more .SBR files when compiling or assembling.
- If an object file does not contain information that you need for your current stage of debugging, omit its .SBR file when rebuilding the database.

Saving Build Time and Disk Space

Unreferenced definitions cause .SBR files to take up more disk space and cause BSCMAKE to run less efficiently. The SBRPACK utility removes unreferenced definitions from .SBR files. For more information, see “SBRPACK” on page 739.

21.2 BSCMAKE

The Microsoft Browser Database Maintenance Utility (BSCMAKE) converts .SBR files created by a compiler or assembler into database files that can be read by the PWB Source Browser. The filename of the resulting browser database has the extension .BSC. For more information on the browser, see “Using the Source Browser” in Chapter 5.

System Requirements for BSCMAKE

BSCMAKE exists as two executable files. The form of BSCMAKE that you run is determined by your computer’s memory. The following executable files are discussed in this section:

- BSCMAKE.EXE for DOS; can use only extended memory
- BSCMAKEEV.EXE for DOS; can use virtual and extended memory

BSCMAKE can use either virtual memory or extended memory (or both) to avoid running out of memory. BSCMAKE.EXE uses extended memory if available. If extended memory is unavailable, BSCMAKE runs under DOS in real mode. The command to invoke this version of BSCMAKE.EXE is:

```
BSCMAKE
```

followed by the rest of the command line. For best results, the sum of available conventional and extended memory should be at least around half of the size of the finished database on disk.

If your computer does not have extended memory or if it is insufficient for your database, you can use virtual memory. BSCMAKEV.EXE uses extended memory if it is available. If extended memory is unavailable or insufficient, BSCMAKEV uses virtual memory, copying information to your disk as needed during the database build. Swapping to disk is slower but can overcome a shortage of memory. The command to invoke this form of BSCMAKE is:

```
BSCMAKEV
```

followed by the rest of the command line.

To prevent BSCMAKE or BSCMAKEV from using extended memory, specify the `/r` option as the first option on the command line.

The BSCMAKE Command Line

To run BSCMAKE, use the following command line:

```
BSCMAKE [[options]] sbrfiles
```

This syntax applies to all forms of BSCMAKE. Specify either BSCMAKE or BSCMAKEV in the first position on the command line.

Options can appear only in the *options* field on the command line. If the `/r` option is used, it must be first.

The *sbrfiles* field specifies one or more .SBR files created by a compiler or assembler. If you specify more than one file, separate the names with spaces or tabs. You must specify the extension; there is no default. You can specify a path with the filename, and you can use operating-system wildcards (`*` and `?`).

During an incremental build, you can specify new .SBR files that were not part of the original build. If you want all contributions to remain in the database, you must specify all .SBR files (including truncated files) that were originally used to create the database. If you omit an .SBR file, that file's contribution to the database is removed.

Do not specify a truncated .SBR file for a full build. A full build requires contributions from all specified .SBR files. Before you perform a full build, recompile and create a new .SBR file for each empty file.

Example

The following command runs BSCMAKE to build a file called MAIN.BSC from three .SBR files:

```
BSCMAKE main.sbr file1.sbr file2.sbr
```

BSCMAKE Options

This section describes the options available for controlling BSCMAKE. Several options control the content of the database by telling BSCMAKE to exclude or include certain information. The exclusion options can allow BSCMAKE to run faster and may result in a smaller .BSC file. Option names are case sensitive (except for /HELP and /NOLOGO).

/Ei filename

/Ei (filename...)

Excludes the contents of the specified include files from the database. To specify multiple files, separate the names with spaces and enclose the list in parentheses. Use /Ei along with the /Es option to exclude files not excluded by /Es.

/El

Excludes local symbols. The default is to include local symbols in the database. For more information about local symbols, see “Preparing to Build a Database” on page 732.

/Em

Excludes symbols in the body of macros. Use /Em to include only the names of macros in the database. The default is to include both the macro names and the result of the macro expansions.

/Er symbol

/Er (symbol...)

Excludes the specified symbols from the database. To specify multiple symbol names, separate the names with spaces and enclose the list in parentheses.

/Es

Excludes from the database every include file specified with an absolute path or found in an absolute path specified in the INCLUDE environment variable. (Usually, these are the system include files, which contain a lot of information that you may not need in your database.) This option does not exclude files specified without a path or with relative paths or found in a relative path in INCLUDE. You can use the /Ei option along with /Es to exclude files that /Es does not exclude. If you want to exclude only some of the files that /Es excludes, use /Ei instead of /Es and list the files you want to exclude.

/HELP

Calls the QuickHelp utility. If BSCMAKE cannot find the Help file or QuickHelp, it displays a brief summary of BSCMAKE command-line syntax.

/Iu

Includes unreferenced symbols. By default, BSCMAKE does not record any symbols that are defined but not referenced. If an .SBR file has been processed by SBRPACK, this option has no effect for that input file because SBRPACK has already removed the unreferenced symbols.

/n

Forces a nonincremental build. Use */n* to force a full build of the database whether or not a .BSC file exists and to prevent .SBR files from being truncated. See “Requirements for a Full Build” on page 733.

/NOLOGO

Suppresses the BSCMAKE copyright message.

/o filename

Specifies a name for the database file. By default, BSCMAKE assumes that the database file has the base name of the first .SBR file and a .BSC extension.

/r

Prevents BSCMAKE from using extended memory under DOS. The */r* option must appear first in the options field on the command line and cannot appear in a response file. BSCMAKE.EXE and BSCMAKEV.EXE are extender-ready and use extended memory if it exists. This option forces BSCMAKE to use only conventional memory and forces BSCMAKEV to use conventional memory and virtual memory. For more information, see “System Requirements for BSCMAKE” on page 734.

/S filename

/S (filename...)

Tells BSCMAKE to process the specified include file the first time it is encountered and to exclude it otherwise. Use this option to save processing time when a file (such as a header, or .H, file for a .C source file) is included in several source files but is unchanged by preprocessing directives each time. You may also want to use this option if a file is changed in ways that are unimportant for the database you are creating. To specify multiple files, separate the names with spaces and enclose the list in parentheses. If you want to exclude the file every time it is included, use the */Ei* or */Es* option.

/v

Provides verbose output, which includes the name of each .SBR file being processed and information about the complete BSCMAKE run.

/?

Displays a brief summary of BSCMAKE command-line syntax.

Example

The following command line tells BSCMAKE to use virtual memory and conventional memory (but not extended memory) to do a full build of MAIN.BSC

from three .SBR files. It also tells BSCMAKE to exclude duplicate instances of TOOLBOX.H:

```
BSCMAKEV /r /n /S toolbox.h /o main.bsc file1.sbr file2.sbr file3.sbr
```

Using a Response File

You can provide part or all of the command-line input in a response file, which is a text file that contains options and/or filenames.

Specify the response file using the following syntax:

```
BSCMAKE @responsefile
```

This syntax applies to all forms of BSCMAKE; you can specify BSCMAKE or BSCMAKEV in the first position on the command line. Only one response file is allowed. You can specify a path with *responsefile*. Precede the filename with an at sign (@). BSCMAKE does not assume an extension. You can specify additional *sbrfiles* on the command line after *responsefile*. If you use */r*, you must specify it on the command line before the response file.

In the response file, specify the input to BSCMAKE in the same order as you would on the command line. Separate the command-line arguments with one or more spaces, tabs, or newline characters.

Example

The following command calls BSCMAKE using a response file:

```
BSCMAKE @prog1.txt
```

Example

The following is a sample response file:

```
/n /v /o main.bsc /E1
/S (
toolbox.h
verdate.h c:\src\inc\screen.h
)
/Er ( HWND HpOfSbIb
LONG LPSTR
NEAR NULL
PASCAL
VOID
WORD
)
file1.sbr file2.sbr file3.sbr file4.sbr
```

BSCMAKE Exit Codes

BSCMAKE returns an exit code (also called return code or error code) to the operating system or the calling program. You can use the exit code to control the operation of batch files or makefiles.

Code	Meaning
0	No error
1	Command-line error
4	Fatal error during database build

21.3 SBRPACK

The Microsoft Browse Information Compactor (SBRPACK) removes unreferenced symbols from .SBR files before they are processed by BSCMAKE. This can result in smaller .SBR files, which allow BSCMAKE to run faster. Smaller .SBR files also save space on disk.

Packing .SBR files is optional. The Microsoft C Compiler version 7.0 (CL) automatically calls SBRPACK when you specify either /FR or /Fr to create an .SBR file. If you specify /FRn or /Frn, CL does not call SBRPACK to pack the .SBR file. Other compilers and assemblers do not pack .SBR files. You may want to use SBRPACK to pack an .SBR file that was created without packing.

SBRPACK.EXE runs under real-mode DOS. It does not use virtual memory, expanded memory, or extended memory.

Overview of SBRPACK

When symbols such as functions or data are defined but not referenced, you can use SBRPACK to remove them from the .SBR files before the files are processed by BSCMAKE. A common source of unreferenced symbols is an include, or header, file. When a source file includes a header file, it often brings in a large number of unreferenced definitions. Therefore, the .SBR file that results from compiling this source file can contain a large amount of unneeded information. The time or disk space saved by SBRPACK is directly related to the number of unreferenced symbols in the .SBR files.

If SBRPACK is not used, BSCMAKE will remove the same information (unless you specify BSCMAKE's /Iu option to preserve this information). However, BSCMAKE can run more efficiently if the .SBR files are first processed by SBRPACK. The time it takes to run both utilities can be less than if BSCMAKE is used alone, especially under real-mode DOS or under extended DOS using virtual memory.

You can run SBRPACK every time you create an .SBR file, or you can run it just once before running BSCMAKE. If you need to save room on your disk, run SBRPACK after every compilation. The .SBR files will then be stored in a more compact form. If you need to accelerate your program-build process, run SBRPACK only as needed, just before running BSCMAKE. The example in the following section shows how to run SBRPACK to perform each kind of efficiency.

The SBRPACK Command Line

To run SBRPACK, use the following command line:

```
SBRPACK [[option]] sbrfiles
```

Options names are not case sensitive. Only the /NOLOGO option applies to a packing session; the other options provide help and then halt SBRPACK.

The *sbrfiles* field specifies one or more .SBR files created by a compiler or assembler. If you specify more than one file, separate the names with spaces or tabs. You must specify the extension; there is no default. You can specify a path with the filename, and you can use operating-system wildcards (* and ?).

You do not specify a name for the resulting files; SBRPACK saves the changed files under their original name. If you want to preserve the original files, copy them to another name before running SBRPACK.

SBRPACK has the following options:

/HELP

Calls the QuickHelp utility. If SBRPACK cannot find the Help file or QuickHelp, it displays a brief summary of SBRPACK command-line syntax.

/NOLOGO

Suppresses the SBRPACK copyright message.

/?

Displays a brief summary of SBRPACK command-line syntax.

Example

The following commands compile a file using the Microsoft C Optimizing Compiler (CL), assemble a file using the Microsoft Macro Assembler (ML), and build a browser database using both SBRPACK and BSCMAKE:

```
CL /FR /c prog1.c
ML /FR /c prog2.asm
SBRPACK prog2.sbr
.
.
.
BSCMAKE *.sbr
```

These commands run SBRPACK every time an .SBR file is created. A separate SBRPACK command isn't needed for PROG1.SBR because CL calls SBRPACK automatically. Later in the program-building session, BSCMAKE builds a database and names it PROG1.BSC. This combination of commands saves space on disk during the program-building session.

The same commands can be configured to create the same database but save running time. In the following example, SBRPACK is called only when BSCMAKE is about to run. If these commands are in a makefile, time is saved if the program-building sequence stops before a database is built.

```
CL /FR /Zn /c prog1.c
ML /FR /c prog2.asm
.
.
.
SBRPACK *.sbr
BSCMAKE *.sbr
```

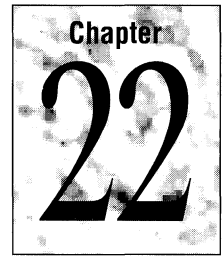
SBRPACK Exit Codes

SBRPACK returns an exit code (also called return code or error code) to the operating system or the calling program. You can use the exit code to control the operation of batch files or makefiles.

Code	Meaning
0	No error
1-4	Fatal SBRPACK error

Each fatal error generates a specific exit code. See “SBRPACK Error Messages” in the *Comprehensive Index and Errors Reference* for individual exit codes associated with each error.

Using Other Utilities



This chapter explains how to use the following utilities:

- CVPACK (Microsoft Debugging Information Compactor) version 4.00
Prepares executable files for use with the CodeView debugger by reducing the size of debugging information within the files.
- IMPLIB (Microsoft Import Library Manager) version 1.30
Creates an import library for use in resolving external references from a Windows program to a dynamic-link library (DLL).
- RM (Microsoft File Removal Utility) version 2.00
UNDEL (Microsoft File Undelete Utility) version 2.00
EXP (Microsoft File Expunge Utility) version 2.00
Manage, delete, and recover backup files.

22.1 CVPACK

This section describes the Microsoft Debugging Information Compactor (CVPACK) version 4.00. This version is a major revision of the CVPACK utility. CVPACK 4.00 prepares an executable file for use with the Microsoft CodeView debugger version 4.00. If your executable file contains debugging information for an earlier version of CodeView, you can use CVPACK to convert it for use with CodeView 4.00.

Run CVPACK only if you want to use CodeView on a file linked with another linker.

CVPACK is automatically called by LINK version 5.30 when you specify LINK's /CO option. It is also called by ILINK if the file was originally linked with /CO. You do not need to run CVPACK as a separate step. However, if you want to use CodeView 4.00 to debug a file that was built by another linker (either an earlier Microsoft linker or a third-party linker), you must run CVPACK to convert the executable file to the later CodeView format. Be sure that the executable file has not been packed by an earlier version of CVPACK; if it has, you must relink the file.

Overview of CVPACK

An executable file to be run under CodeView 4.00 must first be packed by CVPACK 4.00. The debugging information in the file must be in the form given in the Microsoft Symbolic Debugging Information specification. This is the format supported by current Microsoft compilers and linkers.

Earlier formats of debugging information and CVPACK-packing are not compatible with CodeView 4.00. If an executable file contains debugging information in an earlier format but has not been packed, packing with CVPACK 4.00 is all that is needed for the file to run in CodeView 4.00. However, if the executable file has been packed with an earlier version of CVPACK, you must relink the file.

Executable files packed using CVPACK 4.00 are not compatible with earlier versions of CodeView. The debugging information produced by the Microsoft C Compiler version 7.00 and packed by CVPACK 4.00 is for use with CodeView 4.00 and is not compatible with earlier versions of CodeView.

CVPACK compresses debugging information by removing duplicate type definitions. To be removed by CVPACK, the definitions must be absolutely identical. For example, if a structure defined in two modules contains a pointer to another structure, but the second structure is defined in only one module, the pointer size is unknown in the other module. In this case, CVPACK cannot pack the duplicate structure definitions in the same way, which causes less efficient compression.

CVPACK can pack an executable that is in .COM format. The linker puts debugging information for a .COM file into a file with the same base name as the executable file and with a .DBG extension. When you specify a .COM file to be packed, CVPACK looks for a .DBG file with the same base name and in the same location as the .COM file.

The CVPACK Command Line

To run CVPACK, use the following command line:

```
CVPACK [[option]] exefile
```

The *exefile* specifies a single executable file to be packed. You can specify a path with the filename. If you do not specify an extension, CVPACK assumes the default extension .EXE.

CVPACK Options

CVPACK has the following options; the option names are not case sensitive:

/H[[ELP]]

Calls the QuickHelp utility. If CVPACK cannot find the help file or QuickHelp, it displays a brief summary of CVPACK command-line syntax.

/M[[INIMUM]]

Preserves only public symbols and line numbers. All other debugging information is removed from the executable file.

/N[[OLOGO]]

The /NOLOGO option suppresses the copyright message displayed when CVPACK starts.

/?

Displays a brief summary of CVPACK command-line syntax.

Note The /P option is not a valid option for the current version of CVPACK. Using this option causes an error.

Example

The following command packs the file PROJECT.EXE, located in the directory \TEST on the current drive:

```
CVPACK \TEST\PROJECT.EXE
```

CVPACK Exit Codes

CVPACK returns an exit code (also called return code or error code) to the operating system or the calling program. You can use the exit code to control the operation of batch files or makefiles.

Code	Meaning
0	No error
1	Program error caused by commands or files given as input to CVPACK

22.2 IMPLIB

This section describes the Microsoft Import Library Manager (IMPLIB) version 1.30. This utility creates an import library from one or more module-definition (.DEF) files and dynamic-link libraries (DLLs) for use in resolving external references from a Windows program to a DLL. IMPLIB version 1.30 is designed to use .DEF files and DLLs that work with the Microsoft Segmented Executable Linker version 5.30. IMPLIB.EXE runs under real-mode DOS.

About Import Libraries

An “import library” is a static library (usually with a .LIB extension) that can be read by the LINK utility. You specify the import library to LINK in the same ways you specify standard libraries created by the LIB utility. You can use LIB to combine an import library with other static libraries, either standard or import. For more information on LINK, see Chapter 14. For more information on LIB, see Chapter 19.

Import libraries are recommended for resolving references from applications to DLLs. Without an import library, an external reference to a dynamic-link routine must be either declared in an **IMPORTS** statement in the application’s .DEF file or explicitly coded in your program.

This section assumes you are familiar with import libraries, dynamic linking, and module-definition files. For information on module-definition files, see Chapter 16. For information on dynamic linking and import libraries, see the *C for Windows* manual.

IMPLIB uses only the following statements from a module-definition file and ignores other text in the .DEF file:

- **LIBRARY**
- **EXPORTS**
- **INCLUDE**

The IMPLIB Command Line

To run IMPLIB, use the following command line:

```
IMPLIB [[options]] implibname { dllfile... | deffile... }
```

The *options* field specifies IMPLIB options, which are explained in the next section.

The *implibname* field specifies the name for the new import library. You can specify a path with the filename.

The *dllfile* field specifies the name of a DLL. You can use the *deffile* field to specify a module-definition file for the DLL rather than the DLL itself. You can enter multiple *dllfile* and *deffile* specifications. When you specify a DLL, IMPLIB puts all exports from the DLL into the import library. To include only a subset of the DLL’s exported items in the import library, specify a module-definition file that contains only those exports.

IMPLIB does not assume default extensions for any field. You must specify the full names of input and output files and include the file extensions.

Example

```
IMPLIB mylib.lib mylib.dll
```

This command creates the import library named MYLIB.LIB from the dynamic-link library MYLIB.DLL.

Options

Options names are not case sensitive and can be abbreviated to the shortest unique name. IMPLIB has the following options:

`/H[[ELP]]`

Calls the QuickHelp utility. If IMPLIB cannot find the help file or QuickHelp, it displays a brief summary of IMPLIB command-line syntax.

`/NOI[[GNORECASE]]`

Preserves case sensitivity in exported and imported names.

`/NOL[[OGO]]`

Suppresses the IMPLIB copyright message.

`/?`

Displays a brief summary of IMPLIB command-line syntax.

22.3 RM, UNDEL, and EXP

This sections describes the following utilities:

- Microsoft File Removal Utility (RM) version 2.00
- Microsoft File Undelete Utility (UNDEL) version 2.00
- Microsoft File Expunge Utility (EXP) version 2.00

You can use these utilities to create hidden backup files, recover the files when needed, and delete them when no longer needed. You can also use them to manage the backup files created by the Microsoft Programmer's WorkBench (PWB).

RM, UNDEL, and EXP run under real-mode DOS.

Be sure to use matching versions of the RM, EXP, and UNDEL utilities. You can check version numbers by running each utility with the `/?` option.

Overview of the Backup Utilities

The RM, UNDEL, and EXP utilities help you create backup files and manage those files. RM ("remove") moves a file into a hidden subdirectory named

DELETED. UNDEL (“undelete”) makes the file visible again by moving it into DELETED’s parent directory. EXP (“expunge”) deletes the DELETED directory and all files contained within; after being expunged, these files cannot be restored by UNDEL.

You can use RM, UNDEL, and EXP to manage the backup files created by PWB. PWB stores backup files in a DELETED directory when its **Backup** switch is set to **Undel**.

The RM Utility

The RM utility moves one or more files to a hidden directory named DELETED. DELETED is a subdirectory of the directory that contains the file being deleted. Thus RM may create many DELETED directories on your drives or floppy disks. RM creates a DELETED subdirectory of a given directory if one does not already exist. Run RM using the following command line:

```
RM [[options]] [[files]]
```

The *files* field specifies the files to be deleted. You can name more than one file, and you can use operating-system wildcards (* and ?). You can specify a path with the filename. RM prompts for permission before removing a read-only file unless /F is specified.

RM has the following options; the option names are not case sensitive:

/F

Deletes read-only files without prompting for permission.

/HELP

Calls the QuickHelp utility. If RM cannot find the help file or QuickHelp, it displays a brief summary of RM command-line syntax.

/I

Inquires for permission before deleting any file.

/K

Keeps read-only files without deleting or prompting.

/R *directory*

Recurses into subdirectories of *directory* and moves all files into corresponding DELETED directories.

/?

Displays a brief summary of RM command-line syntax.

Example

```
RM /R \PROJECT
```

This command line tells RM to delete all files in the directory tree whose root is the directory named PROJECT. The PROJECT directory lies in the root directory on the current drive. RM moves all files in this tree to hidden directories named DELETED, each of which is created as a subdirectory of a directory that contains the file to be deleted.

The UNDEL Utility

The utility restores one or more deleted files by moving them from a hidden DELETED subdirectory to the parent directory. Run UNDEL using the following command line:

```
UNDEL [{option | files}]
```

The *files* field specifies the files to be restored. If you specify more than one file, separate the names with spaces. You cannot use operating-system wildcards (* and ?). You can specify a path with the filename. If more than one file in DELETED has the specified name, UNDEL lists the versions and prompts for which file to restore. If a file with the same name already exists in the parent directory, UNDEL moves it to the DELETED directory before restoring the specified file.

To list all files in the current directory's DELETED subdirectory, specify the UNDEL command alone. However, you cannot list files in a remote directory; UNDEL does not accept a path without a filename.

UNDEL has the following options; the option names are not case sensitive:

/HELP

Calls the QuickHelp utility. If UNDEL cannot find the help file or QuickHelp, it displays a brief summary of UNDEL command-line syntax.

/?

Displays a brief summary of UNDEL command-line syntax.

Example

```
UNDEL \PROJECT\WORK\REPORT.TXT
```

This command line tells UNDEL to restore the file called REPORT.TXT in the directory \PROJECT\WORK on the current drive. If a file called REPORT.TXT already exists in that directory, UNDEL changes the file to a backup file in \PROJECT\WORK\DELETED before restoring REPORT.TXT. If more than one file called REPORT.TXT exists in \PROJECT\WORK\DELETED, UNDEL prompts for which version to restore.

The EXP Utility

The EXP utility removes a hidden DELETED directory and all files contained within. To run EXP, use the following command line:

```
EXP [[options]] [[directories]]
```

The *directories* field specifies one or more directories containing DELETED directories to be expunged. If no directory is specified, EXP deletes the current directory's DELETED subdirectory.

EXP has the following options; the option names are not case sensitive:

/HELP

Calls the QuickHelp utility. If EXP cannot find the help file or QuickHelp, it displays a brief summary of EXP command-line syntax.

/Q

Suppresses display of the names of deleted files.

/R

Recurse into subdirectories of the current or specified directory and expunges all DELETED directories and files.

/?

Displays a brief summary of EXP command-line syntax.

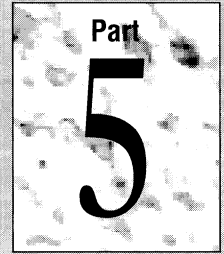
Example

```
EXP /R \PROJECT\WORK
```

This command line tells EXP to:

- Delete the hidden directory \PROJECT\WORK\DELETED along with any files in the directory.
- Recurse through the tree whose root is \PROJECT\WORK and delete any DELETED directories and associated files.

Using Help



Chapter 23	Using Help	755
-------------------	------------------	-----

Using Help

The Microsoft C/C++ provides an online reference to the C language, the development tools, and the PWB and CodeView environments.

Chapter 23 presents the Microsoft Advisor and the QuickHelp program. It describes the system of Help files and explains how to browse it using navigation aids in PWB, CodeView, and QuickHelp.

Microsoft C/C++ offers two systems for accessing Help:

- The Microsoft Advisor, found within the Programmer's WorkBench (PWB) and CodeView
- QuickHelp, the standalone Help program

Both systems provide the same information on important topics and utilities provided with the development system, which include the language, run-time libraries, PWB, and CodeView.

The first part of this chapter, "Structure of the Microsoft Advisor," outlines the structure and contents of Help. The second section, "Navigating Through the Microsoft Advisor," takes you on a quick tour of the system. The third section shows how to use some of PWB's advanced Help features. The next section, "Using QuickHelp," explains how to use the QuickHelp program and how it differs from the Microsoft Advisor. The final section discusses how to manage your Help files.

23.1 Structure of the Microsoft Advisor

The Microsoft Advisor can be compared to a librarian managing a collection of books. Each book, or Help file, has its own table of contents, index, and pages of information. The Advisor organizes the Help files with a global contents and index. All of the files are listed, and their specific tables of contents and indexes can be accessed through the global references. The global contents screen is shown in Figure 23.1.

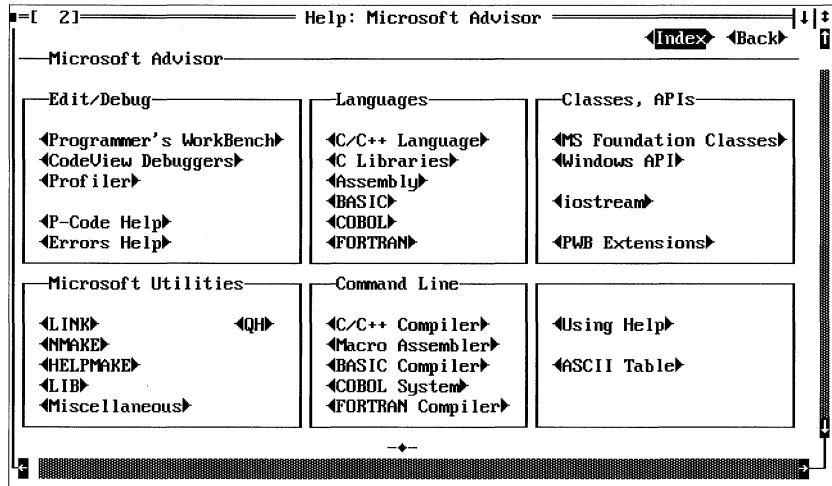


Figure 23.1 Microsoft Advisor Global Contents Screen

You can access a variety of information from the Help system. Information is available on the languages, run-time libraries, errors, and the Help system itself.

23.2 Navigating Through the Microsoft Advisor

You request information about a topic in a window by moving the cursor to it and pressing F1 or by clicking it with the right mouse button. The Help system then searches through the Help files for information about the topic. If it finds the topic, the Help system displays information in the Help window. If Help cannot be found for a particular word or symbol, a message informs you that no information is associated with the topic.

Sometimes, a topic with the same name occurs in several Help files. When you request Help in PWB for one of these names, PWB displays a dialog box in which you can select the context of the topic. The Next command on the Help menu takes you to the next occurrence. When you are using QuickHelp, the first topic is displayed. You can then press E to go to the next occurrence.

Note CodeView does not use the right mouse button for Help in the Source window. Clicking the right button on a line in the Source window executes the program to that line. However, the right mouse button activates Help in the other CodeView windows.

Using the Help Menu

The simplest method for accessing Help is by using the commands found in the PWB and CodeView Help menus. These commands present information in the Help window.

Command	Description
Index	Displays the global index of categories (see Figure 23.2).
Contents	Displays the global Help contents screen.
Topic: <i>topic</i>	Provides information about the topic at the cursor. If information about the topic is available, the topic's name is appended to the Topic command. Otherwise, this command is dimmed.
Help on Help	Displays information about using Help itself.

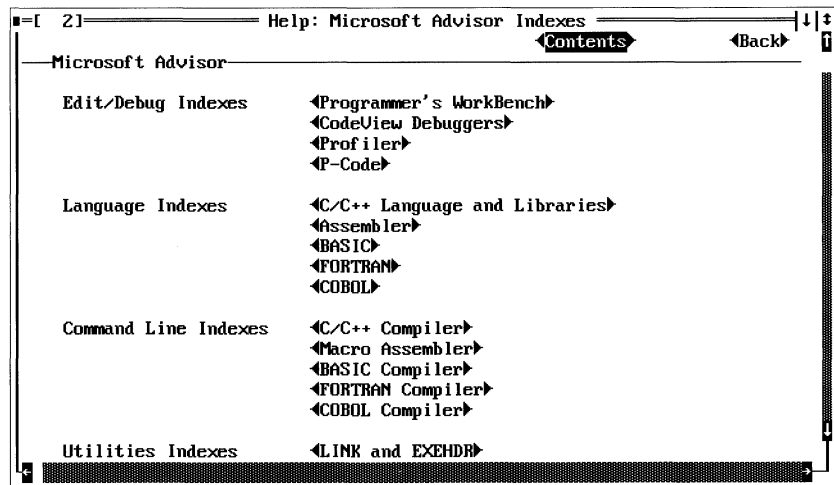


Figure 23.2 Microsoft Advisor Global Index Screen

PWB and QuickHelp provide additional commands to access Help. These commands are described in the program-specific sections at the end of this chapter.

Using the Mouse and the F1 Key

You can use the mouse and the F1 key to get information about any menu command or dialog box, as well as information on keywords, operators, and run-time library functions.

Help on Menu Commands

► **To view information about a menu item:**

1. Open the menu.
2. Drag the mouse to the command and click the right mouse button.
-or-
Use the ARROW keys to select the command and press F1.

The information on the selected command is displayed in a Help dialog box. Figure 23.3 shows the Help information for the Cut command on PWB's Edit menu.

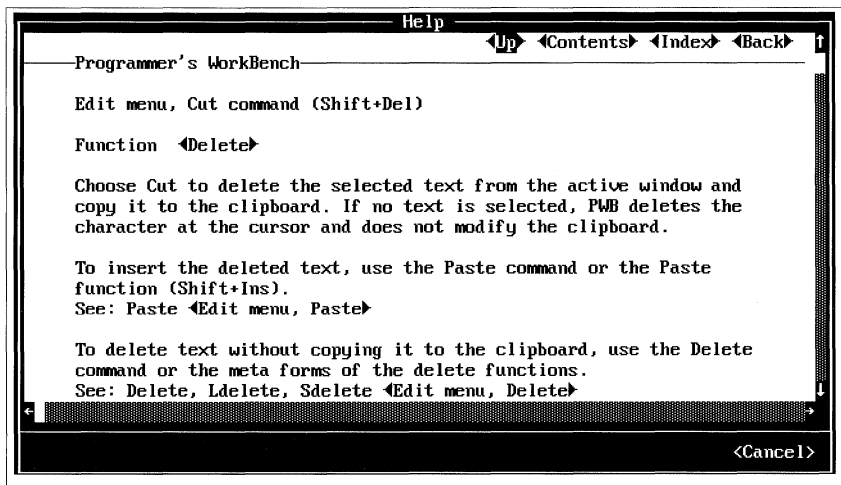


Figure 23.3 Help on the PWB Cut Command

Help on Dialog Boxes

► **To view information about a dialog box:**

1. Open the dialog box.
2. Click the Help button.
-or-
Press F1.

The information on the dialog box is displayed in a Help dialog box.

Using Hyperlinks

Hyperlinks are cross-references that connect related information.

Hyperlinks enclosed by the ◀ and ▶ characters are called “buttons.” Navigate through the Help system by using these buttons.

You can press TAB to move to the next hyperlink button within the Help window. Pressing SHIFT+TAB moves to the previous button. In PWB and CodeView, typing any letter moves the cursor to the next button that begins with that letter; holding down SHIFT and typing a letter moves the cursor backward.

The Microsoft Advisor also recognizes language keywords, library functions, constants, and similar identifiers as hyperlinks, but these are not marked. Unmarked hyperlinks are recognized by the Microsoft Advisor wherever they appear in the Help text or *in your source code*. However, an unmarked hyperlink is not delimited with the ◀ and ▶ characters, and you can't move to it with the TAB key.

An unmarked hyperlink can be executed only by pointing to it with the mouse and clicking the right mouse button or by placing the cursor on it and pressing F1. In QuickHelp, press the S key and then type the text of the hyperlink in the dialog box. In CodeView, use the Help (H) Command-window command.

▶ **To activate a hyperlink with the mouse:**

1. Move the mouse pointer to the hyperlink.
2. Click the right mouse button.
-or-
Click the left mouse button twice (double-click). Double-clicking works only in the Help window.

▶ **To activate a hyperlink with the keyboard:**

1. Press TAB, SHIFT+TAB, or the ARROW keys to move the cursor to the hyperlink. When you move the cursor to a hyperlink button, the entire button is selected.
2. Press F1, ENTER, or SPACEBAR.

Any of these actions displays information about the topic at the cursor. If the topic isn't a hyperlink, a message informs you that no information on the topic could be found.

Note CodeView uses the right mouse button differently in the Source window. Clicking the right button in the Source window executes the program to the line where the mouse was clicked. However, once the Help window is displayed, the right mouse button can be used to activate hyperlinks.

Using Help Windows and Dialog Boxes

The Microsoft Advisor displays information in windows or dialog boxes. Help windows and dialog boxes function in the same way as other windows and dialog boxes found in PWB and CodeView. For a complete description of windows and dialog boxes, see Chapter 4, “User Interface Details.”

Using the Help Window

The Help window displays various contents, indexes, and information about selected topics. Some screens of information are larger than the Help window; information beyond the window borders can be viewed by using the scroll bars or the cursor-movement keys. The **-◆-** symbol indicates the end of information in the Help window.

Navigating with Hyperlinks At the top of most Help windows is a row of hyper-link buttons that are useful for moving through the Help system:

Button	Description
◀Up▶	Moves upward in the hierarchy of Help screens. Since information is ordered in a logical way, moving from the general to the specific, this command is useful for moving up the information tree.
◀Contents▶	Displays the global contents screen. This command is useful because it returns you to a known point in the Help hierarchy. For some Help databases, the Contents button goes to that database's contents.
◀Index▶	Displays the global index list. Selecting an item from the list displays the index for that category. When you are viewing an index for a particular category, the letters on the bar across the top of the screen are hyperlinks. For some Help databases, the Index button goes to that database's index.
◀Back▶	Moves you to the last Help you saw.

The Contents and Index commands on the Help menu always display the global Contents and Index screens.

Screens on a particular topic are frequently grouped together in a Help file. You can press **CTRL+F1** to display information about the next physical topic in the Help file.

Viewing the Previous Help The Microsoft Advisor remembers the last 20 Help screens you've accessed. To return to a previous screen, use the **◀Back▶** button or press **ALT+F1** as many times as necessary to return to the screen you want to see. The Help screen that appears is active; you can ask for Help on any of its hyperlinks or topics.

You can always return to the global Contents screen by choosing Contents from the Help menu or by pressing SHIFT+F1.

Copying and Pasting Help Any text that appears in the PWB Help window can easily be copied to another window. For example, to test an example program from the Help window, you only have to copy it to a new file and compile it. You select and copy text in the Help window just as you do for any other window in PWB.

If you are using QuickHelp, you cannot cut and paste directly into your text editor. However, you can use the commands in the QuickHelp Paste menu to extract pre-determined portions of the Help screen to a file. To change the name of the paste file, choose Rename Paste File from the File menu.

Closing the Help Window Once you're through working with the Help system, you can close the active Help window.

► **To close the Help window:**

- Click the Close button in the upper left corner of the window.
-or-
Press ESC.

Using Help Dialog Boxes

Help dialog boxes provide information about menu commands and dialog boxes. A Help dialog box appears over the windows on the desktop. Unlike the Help window, a Help dialog box must be closed before you can continue. The Cancel button closes the Help dialog box.

► **To view information about a dialog box:**

- Click the Help button in the dialog box.
-or-
Press F1.

► **To close a Help dialog box:**

- Click the Cancel button.
-or-
Press ESC.

Accessing Different Types of Information

This section presents some strategies for accessing the different types of information available within Help system.

Keyword Information

The Help system contains information about all keywords, operators, symbolic constants, and library functions in the development system. If you know the exact name of a keyword, you can type it in a window and click it with the right mouse button or press F1. For operators that do not have an alphabetic name, you must select the operator before activating Help. You can also use the index for the appropriate category of Help.

► To get Help using the index:

1. From the Help menu, choose Index.
-or-
Choose the Index button on any Help screen.
2. Choose the appropriate category of Help from the list of indexes.
Each index has a row of letters across the top.
3. Choose the keyword's first letter from the row of letters. If you want Help for a nonalphabetic operator, choose the asterisk (*).
4. Scroll down the list of entries and choose the topic's hyperlink.

In PWB, you can get Help on a keyword or operator by using the **Arg** function, typing the keyword in the Text Argument dialog box, then pressing F1. Assuming that **Arg** is assigned to ALT+A (the default assignment), the following procedure displays Help for the **printf** function.

► To get Help using the Arg function in PWB:

1. Press ALT+A
PWB displays the message Arg[1] on the status bar.
2. Type printf.
When you type the first letter of the keyword, PWB displays the Text Argument dialog box. Continue typing the keyword.
3. Press F1.
PWB displays the Help for the **printf** function.

► **To get Help on a topic in QuickHelp:**

1. Choose Search from the View menu or press the s key.
QuickHelp displays a dialog box where you can type the topic name.
2. Type the keyword.
3. Click OK or press ENTER.

Figure 23.4 shows a PWB window with the information for **printf**.

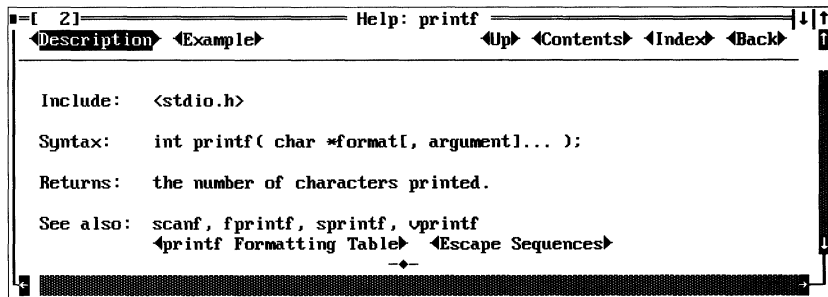


Figure 23.4 Help for printf in a PWB Window

When information about a programming-language keyword or function is shown in the Help window, two additional hyperlink buttons are displayed.

◀Description▶

Provides a detailed explanation of the function. When the description information is displayed, the button changes to ▶Summary▶. Click this button to return to the summarized information about the function.

◀Example▶

Displays source code that provides an example of how the function is used.

Topical Information

The Help system is useful when you want an overview of the available reference topics or when you only have a general idea of what information you need. Start with the global contents screen, and then select any hyperlinks that relate to the topic. You can traverse the hyperlinks until you locate the necessary information.

Menu and Dialog-Box Help

You can get information about any menu command or dialog box by pressing F1 when the menu command is highlighted or the dialog box is displayed. This is helpful when you are first learning to use the development system and you are not completely familiar with all of the features.

Error Help

The Microsoft Advisor provides information about compiler and linker error messages. Whenever a message is displayed on the bottom line of the window in PWB, press F1 to see Help on that error.

You can also get Help for any error in the Build Results window.

► **To find the meaning of an error message using the mouse:**

1. Position the mouse pointer on the error number in the Build Results window.
2. Click the right mouse button.

► **To find the meaning of an error message using the keyboard:**

1. Move the cursor to the Build Results window.
2. Position the cursor on the error number.
3. Press F1.

Help on error messages is also available directly by executing the **Arg** function, typing the error number and its alphabetic prefix, and then pressing F1. Make sure that you type the number exactly—case is significant.

Using Different Help Screens

In addition to the global screens and the topic screens that have already been described in this chapter, the Microsoft Advisor contains some other types of screens that you use in special ways.

Using Index Screens

An index screen has a bar of letters at the top of the screen, below the row of hyperlink buttons. Each letter on the bar is a hyperlink to that letter's list of index entries. The asterisk (*) at the end of the bar is also a hyperlink. This screen lists the nonalphabetic entries. Click the right mouse button on the letter to see that part of the index.

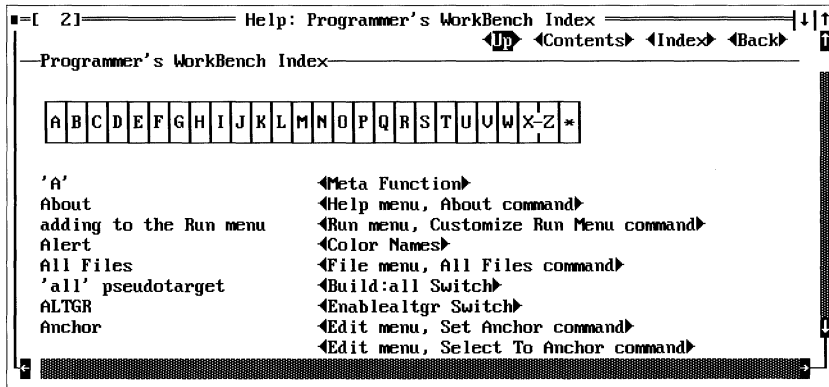


Figure 23.5 PWB Index

Figure 23.5 shows the PWB index screen for the A category. Below the row of alphabet hyperlinks is a list of index entries. Each entry is a hyperlink to the indicated topic.

Using Topic Lists

Some topics are not a screen of text with fixed hyperlink buttons at the top. Instead, they are a list of topics in which each line is a hyperlink. The entire line is highlighted at a time. You can point to the line and click the right mouse button to activate the hyperlink. You can also use the UP ARROW and DOWN ARROW keys to select a topic, and then press F1 or ENTER to go to that topic.

23.3 Using Help in PWB

PWB provides additional Help features to help you find the information you need.

Opening a Help File

You can open Help files temporarily in PWB by using the **SetHelp** function. If you keep rarely used Help files in a directory that is not listed in the HELPFILES environment variable, you can still open the files when you need them.

► To open another Help file in PWB:

1. Execute the **Arg** function (press ALT+A).
2. Type the name of the Help file to open. PWB Displays the Text Argument dialog box when you type the first letter of the filename.

3. Execute the **SetHelp** function (press SHIFT+CTRL+S).

To close a Help file, execute **Arg Meta file SetHelp**. That is, press ALT+A, F9, type the filename, then press SHIFT+CTRL+S.

Global Search

The Global Search command on the Help menu in PWB lets you search all open Help files for a string of text or a regular expression. All text in the Help files is searched, not just the topic names. A global search results in a list of topics, each of which contains text that matches the search string. QuickHelp can also perform global Help searches but does not offer regular-expression matching.

Searching all the Help can take a long time. Therefore, it is recommended that you use the Global Search command only after you have tried other methods of finding the information you need.

Running a Global Search

When you choose the Global Search command, PWB displays the Global Search dialog box where you can specify options for the search. Enter the string or pattern you want to locate in the Find Text box. If you want the search to be case sensitive, turn on the Match Case option. To match a regular expression rather than literal text, turn on the Regular Expression option.

Regular expressions allow you to specify general patterns of text or several alternative strings to match. The current regular-expression syntax is displayed in parentheses after the Regular Expression option. For more information about searching with regular expressions, see Chapter 5, “Advanced PWB Techniques,” and Appendix A.

When you choose OK, PWB starts searching for the specified string or regular expression. The search begins with the Help file that was opened most recently. Because the search can take a long time, it is recommended that you choose a likely category of Help from the global Contents screen before starting a global search.

When you start a global search, PWB displays a dialog box that shows the progress of the search. Choose the Stop Search button at any time to stop the search and view the partial results. When the search ends, PWB displays a list of matching topics.

Using Search Results

When the search is finished, or when you halt the search by choosing Stop Search, PWB displays a list of the topics that contain text that matches the specified string.

Each topic is represented by its title if it has one, followed by the name of the database that contains the topic, and sometimes followed by the topic name.

► **To select a topic from the list:**

- Click the right mouse button on the line.
-or-
Press the UP ARROW and DOWN ARROW keys until the topic is highlighted, and then press ENTER or F1.

PWB displays the selected topic. If that topic does not supply the information that you need, go back to the list and select another topic.

► **To go back to the list:**

- Choose Search Results from the Help menu.
-or-
Press ALT+F1 until the list is displayed.

Restricting the Search

By default, PWB performs a global search in all open databases. There are several ways to control which databases are searched:

- Before the search, display Help from the database that is most likely to contain the information you want. When you run the search, choose Stop Search when the dialog box indicates that the first database has been searched.
- Close some databases by using the **Meta** form of the **SetHelp** function.
- Set the HELPFILES environment variable to the file or files to be searched by using the Environment Variables command on the Options menu. The list of files cannot exceed the MS-DOS limit of 128 characters.

Note that the changes you make to HELPFILES may be restored the next time you start PWB or use the project, depending on the settings of the **Envcursave** and **Envprojsave** switches.

- Choose the Editor Settings command from the Options menu. Then select PWBHELP as the Switch Owner and Text as the switch type. Assign a value to the **Helpfiles** switch to open other Help files in addition to the ones listed in the HELPFILES environment variable.

To see a list of all open Help files and databases, execute the **Arg ? SetHelp** command. The default keystrokes for this are ALT+A, ?, SHIFT+CTRL+S. The resulting list of physical Help files and Help databases is displayed in the Help window.

23.4 Using QuickHelp

QuickHelp is a separate application that provides access to any Help file. It uses the same Help files as the Microsoft Advisor and presents information about topics in the same way. QuickHelp is designed for the developer who prefers using command-line utilities or another editor and doesn't have access to the Microsoft Advisor through PWB.

Major utilities that come with Microsoft C/C++ invoke QuickHelp and display related information when you use the /Help option. You can also use QuickHelp from the command line, as explained in the following sections.

Using the /Help Option

You can get immediate information on major components of Microsoft C/C++ by using the /Help option. The following procedures use the LIB utility as an example. However, you can use these methods for all command-line utilities in the development system.

► **To learn about the LIB utility:**

- At the operating-system command line, type:

```
LIB /Help
```

LIB starts QuickHelp which displays information about LIB.

Using the QH Command

You can also run QuickHelp from the MS-DOS command line or by double-clicking the QuickHelp - Microsoft Advisor icon in the Windows Program Manager.

► **To get Help on the LIB utility:**

- At the operating-system command line, type:

```
QH LIB.EXE
```

You can type the name of any Microsoft utility instead of LIB.

► **To start QuickHelp to view the Advisor Contents screen:**

- At the operating-system command line, type:

```
QH Advisor
```

In addition to information about programs, QuickHelp can also display information about compiler and run-time errors. Type `QH` and the error number with its alphabetic prefix on the command line.

Opening and Closing Help Files

When you run QuickHelp, it looks for the environment variable `HELPPFILES` and opens all listed `.HLP` files. If the `HELPPFILES` variable isn't defined, QuickHelp opens all `.HLP` files in directories specified by the `PATH` environment variable.

Warning Windows Help files are not compatible with QuickHelp. Make sure that Windows Help files are not listed in the `HELPPFILES` environment variable.

Choose the List Database command on the File menu to view a list of all the open Help files.

► **To open additional Help files:**

- Choose the Open Database command from the File menu.
- Type the name of the Help file to be opened in the dialog box that appears. You can specify all Help files in a directory by typing `*.HLP`.
- Press `ENTER` or click the OK button with the left mouse button.

► **To close an open Help file:**

1. Choose the Close Database command from the File menu.
The File menu changes to a list of open Help files.
2. Choose the Help file to close.

Displaying a Topic

You can view information about a topic by using the Search command on the View menu. When topic information is displayed, it is shown in the same format as information presented by the Microsoft Advisor.

► **To display a topic from any of the open Help files:**

1. Choose the Search command from the View menu.
2. Type the topic you want information about in the dialog box.
3. Click the OK button or press `ENTER`.

QuickHelp searches for the topic in the open Help files. If the topic cannot be found, a dialog box informs you that the search failed. If the search is successful, information about the topic is displayed in the QuickHelp window.

Navigating Through Topics

A series of commands on the View menu allow you to display selected topics. These commands include the following:

Command	Description
View History	Displays a list of all the topics that have recently been displayed. See “Using Topic Lists” on page 765 for information on using the list.
View Last	Displays the last topic you looked at.
View Next	Displays the next topic in the Help file.
View Back	Moves backward one topic in the Help file.

Using the QuickHelp Window

The QuickHelp window shown in Figure 23.6 is similar to the Microsoft Advisor Help window. Information that doesn't fully fit in a window can be scrolled, and hyperlinks are used to display additional information. The main difference is that information presented in QuickHelp cannot be copied selectively.

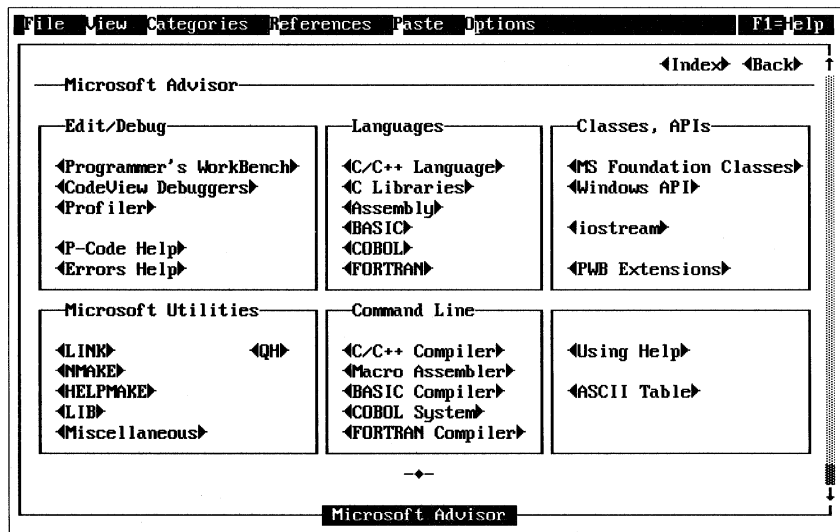


Figure 23.6 The QuickHelp Window

Copying and Pasting in QuickHelp

To transfer information from QuickHelp to another program, specify a file with the Rename Paste File command in the File menu. Once the file is specified, choose the Current Window or the Current Topic command in the Paste menu to transfer the text to that file. Be sure to specify a new file when you paste because QuickHelp overwrites the existing file by default. To append to an existing file, choose the Paste Mode command from the Options menu. The default filename is PASTE.QH in the directory specified by the TMP environment variable.

More About QuickHelp

In addition to the features mentioned previously, QuickHelp has a variety of other options such as changing the appearance of the Help window, searching for text within topics, and controlling the function of the right mouse button.

► **To learn more about QuickHelp's features:**

1. Make sure the QH.HLP file is open.
2. To view QuickHelp's Help, press F1.

-or-

To get information about a menu command, click it with the right mouse button, or highlight the command and press F1.

23.5 Managing Help Files

When you run the SETUP program for Microsoft C/C++, you are given a choice of whether to install the Help files. If you choose to install Help, SETUP copies the Help files to the directory that you specify. By default, this is the C700\Help directory.

Several other Microsoft products contain a Microsoft Advisor Help system. If you have more than one of these products, you can use all the files as one system by copying all .HLP files to a common directory. However, make sure that Windows Help files are separate from the Advisor Help files.

Some Help files, such as UTILS.HLP, exist in other Microsoft language products. When an existing Help file has the same filename as a Microsoft C/C++ Help file, use the most recent file. Note that the files RC.HLP and UTILERR.HLP are obsolete and should be deleted or moved to another directory.

The HELPFILES environment variable tells the Advisor where to find Help files. You usually set this variable in AUTOEXEC.BAT. If you move the Help files, make sure to change the SET command in AUTOEXEC.BAT to point HELPFILES to the new location.

Managing Many Help Files

If you have a large number of Help files, you may reach a limit on the number of physical Help files or Help databases that can be open at one time. QuickHelp, PWB, and CodeView display a message when you have too many Help files. If this is the case, you must do one or more of the following:

- Delete all obsolete Help files.
- Move rarely used Help files to another directory. You can then open these files as you need them.
- Concatenate some Help files.

It is recommended that you always keep ADVISOR.HLP. Moreover, for Help on error messages, you must use the Help file for the tool that issues the error. It is recommended that you save backup copies of all Help files before concatenating, splitting, or deleting any files.

To open and close Help files in PWB, use the **SetHelp** function. To open and close Help files in QuickHelp, choose the Open Database and Close Database commands from the File menu.

You can get a listing of the open Help files in PWB and QuickHelp. These lists show the open Help files, the Help databases contained in the files, and the title for each database if it has one. To get a list of open Help files in PWB, execute the function sequence **Arg ? SetHelp**. With the default keystrokes, press ALT+A, type a question mark (?), then press SHIFT+CTRL+S. To get a list of open Help files in QuickHelp, choose the List Databases command from the File menu. Once you have created the list of Help files, you can print it for later reference.

Concatenating Help Files

To concatenate two or more physical Help files, use the MS-DOS COPY command. The syntax for using the COPY command to combine Help files is:

COPY *file /b* [[+ *file /b*]]... *newfile*

Use a plus sign (+) between the filenames of the original Help files. Specify the /b option to copy the files as binary files. If you don't specify a new filename, the resulting file takes the name of the first file and the original file is overwritten.

You can use this command to combine two Microsoft Advisor Help files. For example, to create a physical Help file named ADVISOR.HLP that contains ADVISOR.HLP and QH.HLP, use the following command:

```
COPY ADVISOR.HLP /b + QH.HLP /b
```

You can also combine your own Help file (created using HELPMAKE) with Microsoft Help files.

Splitting Help Files

To split a physical Help file into its component databases, use the HELPMAKE utility. The syntax for using HELPMAKE to split a Help file is:

HELMAKE /DS *file*

Specify the /DS option when splitting a Help file. For more information on the /DS option, as well as other uses of HELPMAKE, see Chapter 20. HELPMAKE creates individual physical files with the name of the original Help database. The resulting files are created in the current directory.

For example, the following command extracts the component Help databases from the UTILS.HLP file:

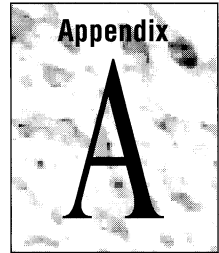
```
HELMAKE /DS UTILS.HLP
```

The UTILS.HLP file itself is not changed. You can delete the unneeded component files and then concatenate the remaining files to create a new version of UTILS.HLP.

Appendixes

Appendix A	Regular Expressions	777
B	Decorated Names.....	789
C	United States ASCII Character Chart (Code Page 437)	793
D	Multilingual ASCII Character Chart (Code Page 850)	797
E	Key Codes.....	799

Regular Expressions



A regular expression (sometimes called a “pattern”) is a find string that uses special characters to match patterns of text. You can use regular expressions to find patterns such as five-digit numbers or strings in quotation marks. Selected portions of found text can be used in a replacement.

In PWB you can specify regular expressions in two ways: UNIX syntax and non-UNIX syntax. UNIX regular expressions have a syntax similar to regular expressions in the UNIX and XENIX operating systems. CodeView uses a subset of the UNIX regular-expression syntax. Non-UNIX regular-expression syntax has the features of UNIX regular expressions but includes additional features and uses a more compact syntax.

The **Unixre** switch determines whether PWB uses UNIX or non-UNIX regular expressions in searches. PWB switches that accept regular expressions, such as **Build** and **Word**, always use UNIX syntax.

A.1 Regular-Expression Summaries

The following table summarizes PWB's UNIX regular-expression syntax.

Table A.1 UNIX Regular-Expression Summary

Syntax	Description
<code>\c</code>	Escape: literal character <i>c</i>
<code>.</code>	Wildcard: any character
<code>^</code>	Beginning of line
<code>\$</code>	End of line
<code>[class]</code>	Character class: any one character in set
<code>[^class]</code>	Inverse class: any one character not in set
<code>x*</code>	Repeat: zero or more occurrences of <i>x</i>
<code>x+</code>	Repeat: one or more occurrences of <i>x</i>
<code>\{x\}</code>	Grouping: group subexpression for repetition
<code>\{x!y!z\}</code>	Alternation: match one from the set
<code>\~x</code>	"Not": fail if <i>x</i> at this point
<code>\(x)</code>	Tagged expression
<code>\n</code>	Reference to tagged expression number <i>n</i>
<code>\:e</code>	Predefined expression

The following table summarizes the UNIX predefined expressions.

Table A.2 UNIX Predefined Expressions

Syntax	Description
<code>\:a</code>	Alphanumeric character
<code>\:b</code>	White space
<code>\:c</code>	Alphabetic character
<code>\:d</code>	Digit
<code>\:f</code>	Part of a filename
<code>\:h</code>	Hexadecimal number
<code>\:i</code>	Microsoft C/C++ identifier
<code>\:n</code>	Unsigned number
<code>\:p</code>	Path
<code>\:q</code>	Quoted string
<code>\:w</code>	English word
<code>\:z</code>	Unsigned integer

CodeView uses a subset of the UNIX regular-expression syntax. You can use regular expressions as arguments to the Search (/) command and Examine Symbols (X) command. The following table summarizes CodeView regular expressions.

Table A.3 CodeView Regular Expressions

Character	Syntax	Meaning
Backslash	\c	Matches a literal character <i>c</i> . (Escape)
Period	.	Matches any single character. (Wildcard)
Caret	^	Matches the beginning of a line. The caret must appear at the beginning of the pattern.
Dollar sign	\$	Matches the end of a line. The dollar sign must appear at the end of the pattern.
Asterisk	c*	Matches zero or more occurrences of <i>c</i> .
Brackets	[...]	Matches any one character in the set of the characters within the brackets.

Within the brackets, you can specify a negated set and ranges of characters by using the following notation:

Character	Syntax	Meaning
Dash	-	Specifies a range of characters in the ASCII order between the characters on either side, inclusive. For example, [a-z] matches the lowercase alphabet.
Caret	^	Matches any one character not within the brackets. The caret must be the first character within the brackets. For example, [^0-9] matches any character except a digit.

The following table summarizes the non-UNIX regular-expression syntax.

Table A.4 Non-UNIX Regular-Expression Summary

Syntax	Description
<code>\c</code>	Escape: literal character <i>c</i>
<code>?</code>	Wildcard: any character
<code>^</code>	Beginning of line
<code>\$</code>	End of line
<code>[class]</code>	Character class: any one character in set
<code>[~class]</code>	Inverse class: any one character not in set
<code>x*</code>	Repeat: zero or more occurrences of <i>x</i>
<code>x+</code>	Repeat: one or more occurrences of <i>x</i>
<code>x@</code>	Repeat: maximal zero or more occurrences of <i>x</i>
<code>x#</code>	Repeat: maximal one or more occurrences of <i>x</i>
<code>(x)</code>	Grouping: group subexpression for repetition
<code>(x!y!z)</code>	Alternation: match one from the set
<code>~x</code>	“Not”: fail if <i>x</i> at this point
<code>x^n</code>	“Power”: match <i>n</i> copies of <i>x</i>
<code>{x}</code>	Tagged expression
<code>\$n</code>	Reference to tagged expression number <i>n</i>
<code>:e</code>	Predefined expression

The following table summarizes the non-UNIX predefined expressions.

Table A.5 Non-UNIX Predefined Expressions

Syntax	Description
<code>:a</code>	Alphanumeric character
<code>:b</code>	White space
<code>:c</code>	Alphabetic character
<code>:d</code>	Digit
<code>:f</code>	Part of a filename
<code>:h</code>	Hexadecimal number
<code>:i</code>	Microsoft C/C++ identifier
<code>:n</code>	Unsigned number
<code>:p</code>	Path
<code>:q</code>	Quoted string
<code>:w</code>	English word
<code>:z</code>	Unsigned integer

A.2 UNIX Regular-Expression Syntax

PWB uses the following UNIX-style regular-expression syntax:

Table A.6 UNIX Regular-Expression Syntax

Syntax	Description
<code>\c</code>	Escape: matches a literal occurrence of the character <i>c</i> and ignores any special meaning of <i>c</i> in a regular expression. For example, the expression <code>\?</code> matches a question mark (?), <code>\^</code> matches a caret (^), and <code>\\</code> matches a backslash (\).
<code>.</code>	Wildcard: matches any single character. For example, the expression <code>a.a</code> matches <code>aaa</code> and <code>aθa</code> .
<code>^</code>	Beginning of line. For example, the expression <code>^The</code> matches the word <code>The</code> only when it occurs at the beginning of a line.
<code>\$</code>	End of line. For example, the expression <code>end\$</code> matches the word <code>end</code> only when it occurs at the end of a line.
<code>[class]</code>	Character class: matches any one character in the class. Use a dash (–) to specify a range of characters. Within a class, all characters except <code>^-\\</code> are treated literally. For example, <code>[a-zA-Z0-9]</code> matches any character or digit, and <code>[abc]</code> matches <code>a</code> , <code>b</code> , or <code>c</code> .
<code>[^class]</code>	Inverse of character class: matches any character not specified in the class. For example, <code>[^0-9]</code> matches any character that is not a digit.
<code>x*</code>	Repeat operator: matches zero or more occurrences of <i>x</i> , where <i>x</i> is a single character, a character class, or a grouped expression. For example, the regular expression <code>ba*b</code> matches <code>baaab</code> , <code>bab</code> , and <code>bb</code> . This operator always matches as many characters as possible.
<code>x+</code>	Repeat operator (shorthand for <code>xx*</code>): matches one or more occurrences of <i>x</i> . For example, the regular expression <code>ba+b</code> matches <code>baab</code> and <code>bab</code> but not <code>bb</code> .
<code>\(x)</code>	Tagged expression: marked text, which you can refer to as <code>\n</code> elsewhere in the find or replacement string. Within a find string, PWB finds text that contains the previously tagged text. Within a replacement string, PWB reuses the matched text in the replacement.
<code>\n</code>	References the characters matched by a tagged expression, where <i>n</i> is a one-digit number and indicates which expression. The first tagged expression is <code>\1</code> , the second is <code>\2</code> , and so on. The entire expression is represented as <code>\0</code> .
<code>\{x\}</code>	Grouping. Groups a regular expression so that you can use a repeat operator on the subexpression. For example, the regular expression <code>\{Test\}+</code> matches <code>Test</code> and <code>TestTest</code> .
<code>\{x\ y\ z\}</code>	Alternation: matches one from a set of alternate patterns. The alternates are tried in left-to-right order. The next alternate is tried only when the rest of the pattern fails. For example, <code>\{ +\!\$\}</code> matches a sequence of blanks or the end of a line.

Table A.6 (continued)

Syntax	Description
<code>\~x</code>	“NOT” function: matches nothing but checks to see whether the text matches <i>x</i> at this point and fails if it does. For example, <code>^\~\{ \!\$\}.*</code> matches all lines that do not begin with white space or end of line.
<code>\:e</code>	Predefined regular expression, where <i>e</i> is a letter specifying the regular expression.

Examples

In PWB, to find the next occurrence of a number (a string of digits) that begins with the digit 1 or 2:

1. Execute **Arg Arg** (ALT+A ALT+A)
2. Type `[12][0-9]*`
3. Execute **Psearch** (F3)

The special characters in regular expression syntax are most powerful when they are used together. For example, the following combination of the wildcard (.) and repeat (*) characters

`.*`

matches any string of characters. This expression is useful when it is part of a larger expression, such as

`B.*ing`

which matches any string beginning with `B` and ending with `ing`.

A.3 Tagged Regular Expressions

Tagged expressions are regular expressions enclosed by the delimiters `\` and `\` (UNIX) or `{` and `}` (non-UNIX). Use tagged expressions to match repeated elements and to mark substrings for use in a replacement. Note that a tagged expression is not the same as a grouped expression.

When you specify a regular expression with tagged subexpressions, PWB finds text that matches the regular expression and marks each substring matching a tagged subexpression.

Example

The UNIX regular expression

```
\(<\)\([^\>]+\)\(>\)
```

matches the string

```
<bracketed>
```

and tags the < , bracketed, and > substrings.

To refer to tagged text in a find or replacement pattern, use $\backslash n$ (UNIX) or $\$n$ (non-UNIX), where n is the number of a tagged subexpression from 1 to 9. In a find pattern, this reference matches another occurrence of the previously matched text, not another occurrence of the regular expression. In a replacement, PWB uses the matched text.

The entire match is implicitly tagged for use in replacement text. Use $\backslash 0$ (UNIX) or $\$0$ (non-UNIX) to refer to the entire match. For example, the UNIX find pattern

```
^\([^\ ]+\) +\([^\ ]+\).*
```

with the replace pattern

```
\2 \1 (\0)
```

matches lines without leading spaces and at least two words. It replaces them with lines that consist of the transposed words followed by the original line in parentheses.

Example

The tagged expressions:

UNIX

Non-UNIX

```
\([A-Za-z]+\)=\1      {[A-Za-z]+}==\1
```

match one or more letters followed by two equal signs (==) and a repetition of the letters. They match the first two strings below, but not the third:

```
ABCxyz==ABCxyz
```

```
i==i
```

```
ABCxyz==KBCxjj
```

The following example finds one or more hexadecimal digits followed by the letter H. Each matching string is replaced by a string that consists of the original digits (which were tagged so they could be reused) and the prefix 16#.

1. Find strings of the form *hexdigits*H with the UNIX and non-UNIX patterns:

UNIX	Non-UNIX
<code>\([0-9a-fA-F]+\)H</code>	<code>{[0-9a-fA-F]+}H</code>

These patterns can also be expressed by using the predefined pattern for hexadecimal digits:

UNIX	Non-UNIX
<code>\(:h*\)H</code>	<code>{:h}H</code>

2. Replace with the patterns:

UNIX	Non-UNIX
<code>16#\1</code>	<code>16#\\$1</code>

Tagged Expressions in Build:Message

PWB uses tagged UNIX regular expressions to find the location of errors and warnings displayed in the Build Results window. The tagged portions of the message indicate the file and the location or token in error.

To define new messages for PWB to recognize, add a new **Build:message** switch definition to the [PWB] section of TOOLS.INI. The syntax for this switch is:

```
Build:message "pattern" [[file [[line [[column]] ]] | token]]
```

The pattern is a macro string that specifies a tagged UNIX regular expression. The file, line, col, and token keywords indicate the meaning of each tagged subexpression.

For example, if the messages you want to match look like:

```
Error: Missing ';' on line 123 in SAMPLE.XYZ
```

Place the following setting in TOOLS.INI:

```
Build:message "^Error: .* on line \\(\\:z\\) in \\(\\:p\\)" \
    file line
```

Note that each backslash in the regular expression is doubled within the macro string. This pattern uses the predefined expressions for integer (**\:z**) and path (**\:p**).

Table A.7 (continued)

:e	Description Definition (non-UNIX)
:i	Microsoft C/C++ identifier "([a-zA-Z_ \$][a-zA-Z0-9_ \$]@)"
:n	Unsigned number "([0-9]#[0-9]@[0-9]@[0-9]#![0-9]#)"
:p	Path "((([A-Za-z\\:!])(\\\\!\\/!))(:f(.:f!)(\\\\!\\/!))@:f(.:f!\\.!))"
:q	Quoted string "(\\"[~\"]@\"!' [~']@')"
:w	English word "([a-zA-Z]#)"
:z	Unsigned integer "([0-9]#)"

A.6 Non-UNIX Regular-Expression Syntax

PWB uses the following non-UNIX regular-expression syntax:

Table A.8 Non-UNIX Regular Expression Syntax

Syntax	Description
<code>\c</code>	Escape: matches a literal occurrence of the character <i>c</i> and ignores any special meaning of <i>c</i> in a regular expression. For example, the expression <code>\?</code> matches a question mark <code>?</code> , <code>\^</code> matches a caret <code>^</code> , and <code>\\</code> matches a backslash <code>\</code> .
<code>?</code>	Wildcard: matches any single character. For example, the expression <code>a?a</code> matches <code>aaa</code> and <code>a1a</code> but not <code>aBBa</code> .
<code>^</code>	Beginning of line. For example, the expression <code>^The</code> matches the word <code>The</code> only when it occurs at the beginning of a line.
<code>\$</code>	End of line. For example, the expression <code>end\$</code> matches the word <code>end</code> only when it occurs at the end of a line.
<code>[class]</code>	Character class: matches any one character in the class. Use a dash (<code>-</code>) to specify a range of characters. Within a class, all characters except <code>~- \]</code> are treated literally. For example, <code>[a-zA-Z*]</code> matches any alphabetic character or asterisk, and <code>[abc]</code> matches a single <code>a</code> , <code>b</code> , or <code>c</code> .
<code>[~class]</code>	Inverse of character class: matches any single character not in the class. For example, <code>[~0-9]</code> matches any character that is not a digit.
<code>x*</code>	Minimal matching: matches zero or more occurrences of <i>x</i> , where <i>x</i> is a single character or a grouped expression. For example, the expression <code>ba*b</code> matches <code>baaab</code> , <code>bab</code> , and <code>bb</code> .

Table A.8 (continued)

Syntax	Description
$x+$	Minimal matching plus (shorthand for xx^*): matches one or more occurrences of x . For example, the expression $ba+b$ matches $baab$ and bab but not bb .
$x@$	Maximal matching: identical to x^* , except that it matches as many occurrences as possible.
$x\#$	Maximal matching plus: identical to $x+$, except that it matches as many occurrences as possible.
$(x!y!z)$	Alternation: matches one from a set of alternate patterns. The alternates are tried in left-to-right order. The next alternate is tried only when the rest of the pattern fails. For example, the expression $(ww!xx!xxyy)zz$ matches $xxzz$ on the second alternative and $xxyyzz$ on the third.
(x)	Grouping. Groups an expression so that you can use a repeat operator with the expression. For example, the expression $(Test)^+$ matches $Test$ and $TestTest$.
$\sim x$	“NOT” function: matches nothing but checks to see if the text matches x at this point and fails if it does. For example, $\sim(if!while)?*\$$ matches all lines that do not begin with <code>if</code> or <code>while</code> .
x^n	Power function: matches n copies of x . For example, w^4 matches <code>www</code> , and $(a?)^3$ matches <code>a#aba5</code> .
$\{x\}$	Tagged expression: marked text, which you can refer to as $\$n$ elsewhere in the find or replacement string. Within a find string, PWB finds text that contains the previously tagged text. Within a replacement string, PWB reuses the matched text in the replacement.
$\$n$	Reference to text matched by a tagged expression. The specific substring is indicated by n . The first tagged substring is indicated as $\$1$, the second as $\$2$, and so on. A $\$0$ represents the entire match.
$:e$	Predefined regular expression, where e is a letter that specifies the regular expression.

Examples

In PWB, to find the next occurrence of a number (a string of digits) that begins with the digit 1 or 2:

1. Execute **Arg Arg** (ALT+A ALT+A).
2. Type `[12][0-9]*`
3. Execute **Psearch** (F3).

Regular expressions are most powerful when they are used together. For example, the combination of the wildcard (?) and repeat (*) operators

?*

matches any string of characters. This expression is useful when it is part of a larger expression, such as

```
B?*ing
```

which matches any string beginning with `B` and ending with `ing`.

Non-UNIX Matching Method

The type of non-UNIX matching method is significant only when you use a find-and-replace command. “Matching method” refers to the technique used to match repeated expressions. For example, does the expression `a*` match as few or as many characters as it can? The answer depends on the matching method.

PWB supports two matching methods in non-UNIX regular expressions:

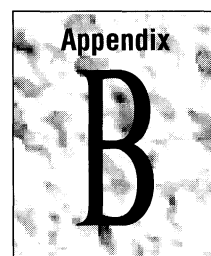
- “Minimal matching” matches as few characters as possible to find a match. For example, `a+` matches only the first character in `aaaaa`. However, `ba+b` matches the entire string `baaaab` because it is necessary to match every `a` to match both occurrences of `b`.
- “Maximal matching” matches as many characters as possible. For example, `a#` matches the entire string `aaaaaa`.

Example

If `a+` (minimal matching plus) is the find string and `EE` is the replacement string, PWB replaces `aaaaa` with `EEEEEEEEE` because at each occurrence of `a`, PWB immediately replaces it with `EE`.

However, if `a#` (maximal matching plus) is the find string, PWB replaces the same string with `EE` because it matches the entire string `aaaaa` at once and replaces that string with `EE`.

Decorated Names



This appendix discusses decorated names for functions in C++ programs. You must use a decorated name when you need to specify a C++ function name as it is known internally by the linker.

B.1 Overview

Functions in C++ programs are known internally by their decorated names. A decorated name is created by the compiler during compilation of the function definition or prototype. In most circumstances, you do not need to know the decorated name of a function. However, some uses of function names require you to specify the decorated name of a C++ function. For example, the **EXPORTS**, **IMPORTS**, and **FUNCTIONS** statements in a module-definition (.DEF) file require decorated names for C++ functions. These and other .DEF file statements are described in Chapter 16.

Format of a Decorated Name

A decorated name is a string generated by the compiler. It contains the following information:

- The function name.
- The class that the function is a member of, if it is a member function. This may include the class that encloses the function's class, and so on.
- The types of the function's parameters.
- The calling convention.
- The return type of the function.

The function and class names are expressed literally in the string. The rest of the string is a code that has internal meaning only for the compiler and linker.

Note The decoration for a C identifier, in contrast to a C++ identifier, consists only of a leading underscore.

Viewing Decorated Names

The decorated name of a function is not generated until compilation. After the program is compiled, the linker uses the decorated form of the name. Use the `LINK /MAP` option to create a map file that shows decorated names. You can control the map output as follows:

- Specify `/MAP` with no qualifier to get a map file that contains public symbols listed by name and by address. The map file gives the decorated form of C++ function names in each of these lists.
- Specify `/MAP:ADDRESS` to get the same map file that `/MAP` creates but without the list of symbols sorted by name.
- Specify `/MAP:FULL` to add the undecorated form of each name to the map file produced by `/MAP`. The undecorated form is given after the decorated name. This option also adds the contributions of object files to segments.

To see the undecorated form of the decorated name, you must use `/MAP:FULL`. Note that inline functions do not generate entries in a map file. For more information on `LINK` and the `/MAP` option, see Chapter 14.

Examples

The following are examples of decorated names and their undecorated versions:

```
?calc@@YAHH@Z
    int __near __cdecl calc(int)

?getMonth@Date@@QACHXZ
    public: int __near __pascal Date::getMonth(void)__near
```

These names are from the map file that is produced when `/MAP:FULL` is specified to the linker.

B.2 Getting and Specifying a Decorated Name

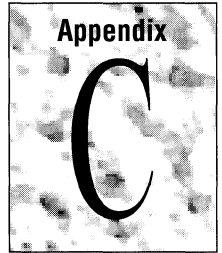
Some uses of C++ function names require that you specify the name in its decorated form. Decorated names are required in the **FUNCTIONS**, **EXPORTS**, and **IMPORTS** statements in a `.DEF` file. This section describes how to get and use a decorated name.

For example, the `FUNCTIONS` statement in a `.DEF` file accepts one or more names of functions that are to be placed in a specified order or assigned to a segment or an overlay. To assign a C++ function using the `FUNCTIONS` statement, you must give the function's decorated name. However, the decorated name is not known until after compilation. Therefore, you must use the following procedure to get and use decorated names in a `.DEF` file:

1. Compile the object files for your program.
2. Create a `.DEF` file that does not yet specify the C++ functions.
3. Link your program using a form of the `/MAP` option.
4. Examine the map file to learn the decorated names of the functions.
5. Specify the decorated names in the places where you want to use the functions in the `.DEF` file.
6. Relink your program.

Warning If you change the function name, class, calling convention, any parameter, or the return type, the old decorated name is no longer valid. You must repeat this procedure and use the new version of the decorated name everywhere it is specified.

United States ASCII Character Chart (Code Page 437)



ASCII Codes

Ctrl	Dec	Hex	Char	Code
~@	0	00		NUL
^A	1	01	☐	SOH
~B	2	02	☐	STX
^C	3	03	☐	ETX
~D	4	04	☐	EOT
^E	5	05	☐	ENQ
~F	6	06	☐	ACK
^G	7	07	☐	BEL
~H	8	08	☐	BS
^I	9	09	☐	HT
~J	10	0A	☐	LF
^K	11	0B	☐	VT
~L	12	0C	☐	FF
^M	13	0D	☐	CR
~N	14	0E	☐	SO
^O	15	0F	☐	SI
~P	16	10	☐	DLE
^Q	17	11	☐	DC1
~R	18	12	☐	DC2
^S	19	13	☐	DC3
~T	20	14	☐	DC4
^U	21	15	☐	NAK
~V	22	16	☐	SYN
^W	23	17	☐	ETB
~X	24	18	☐	CAN
^Y	25	19	☐	EM
~Z	26	1A	☐	SUB
^[27	1B	☐	ESC
~\	28	1C	☐	FS
^]	29	1D	☐	GS
~_	30	1E	☐	RS
^`	31	1F	☐	US

Dec	Hex	Char
32	20	!
33	21	"
34	22	#
35	23	\$
36	24	%
37	25	&
38	26	'
39	27	(
40	28)
41	29	*
42	2A	+
43	2B	,
44	2C	-
45	2D	.
46	2E	/
47	2F	0
48	30	1
49	31	2
50	32	3
51	33	4
52	34	5
53	35	6
54	36	7
55	37	8
56	38	9
57	39	:
58	3A	;
59	3B	<
60	3C	=
61	3D	>
62	3E	?
63	3F	

Dec	Hex	Char
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D]
94	5E	^
95	5F	_

Dec	Hex	Char
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	
125	7D	}
126	7E	~
127	7F	Δ†

† ASCII code 127 has the code DEL. Under DOS, this code has the same effect as ASCII 8 (BS).
The DEL code can be generated by the CTRL + BKSP key combination.

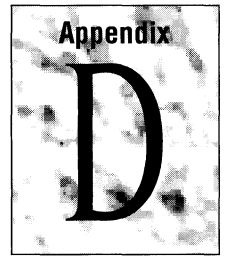
Dec	Hex	Char
128	80	Ç
129	81	ü
130	82	ë
131	83	Š
132	84	à
133	85	â
134	86	ä
135	87	å
136	88	æ
137	89	ë
138	8A	e
139	8B	i
140	8C	T
141	8D	i
142	8E	ñ
143	8F	ê
144	90	È
145	91	É
146	92	À
147	93	ó
148	94	ô
149	95	õ
150	96	ü
151	97	ÿ
152	98	ö
153	99	ü
154	9A	ü
155	9B	ç
156	9C	ê
157	9D	ÿ
158	9E	ÿ
159	9F	ÿ

Dec	Hex	Char
160	A0	ä
161	A1	i
162	A2	ö
163	A3	ü
164	A4	ñ
165	A5	ñ
166	A6	e
167	A7	e
168	A8	ç
169	A9	r
170	AA	ç
171	AB	½
172	AC	¼
173	AD	i
174	AE	«
175	AF	»
176	B0	⋮
177	B1	⋮
178	B2	⋮
179	B3	
180	B4	†
181	B5	‡
182	B6	¶
183	B7	¶
184	B8	¶
185	B9	¶
186	BA	¶
187	BB	¶
188	BC	¶
189	BD	¶
190	BE	¶
191	BF	¶

Dec	Hex	Char
192	C0	L
193	C1	L
194	C2	T
195	C3	T
196	C4	-
197	C5	†
198	C6	‡
199	C7	¶
200	C8	¶
201	C9	¶
202	CA	¶
203	CB	¶
204	CC	¶
205	CD	=
206	CE	¶
207	CF	¶
208	D0	¶
209	D1	¶
210	D2	¶
211	D3	¶
212	D4	¶
213	D5	F
214	D6	¶
215	D7	¶
216	D8	¶
217	D9	J
218	DA	r
219	DB	■
220	DC	■
221	DD	■
222	DE	■
223	DF	■

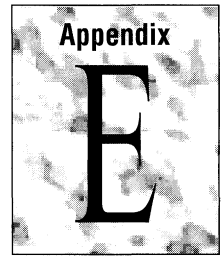
Dec	Hex	Char
224	E0	α
225	E1	β
226	E2	γ
227	E3	π
228	E4	Σ
229	E5	σ
230	E6	μ
231	E7	τ
232	E8	θ
233	E9	θ
234	EA	Ω
235	EB	δ
236	EC	φ
237	ED	φ
238	EE	ε
239	EF	π
240	F0	≡
241	F1	±
242	F2	∫
243	F3	∫
244	F4	∫
245	F5	J
246	F6	∫
247	F7	∫
248	F8	∫
249	F9	.
250	FA	.
251	FB	√
252	FC	∫
253	FD	∫
254	FE	∫
255	FF	∫

Multilingual ASCII Character Chart (Code Page 850)



0		32		64	Ⓔ	96	`	128	Ç	160	á	192	L	224	Ó
1	Ⓢ	33	!	65	Ⓐ	97	a	129	ü	161	í	193	Ⓘ	225	ß
2	Ⓣ	34	"	66	B	98	b	130	é	162	ó	194	T	226	ô
3	♥	35	#	67	C	99	c	131	â	163	ú	195	†	227	ò
4	♦	36	\$	68	D	100	d	132	ä	164	ñ	196	—	228	õ
5	♣	37	%	69	E	101	e	133	à	165	ñ	197	†	229	ö
6	♠	38	&	70	F	102	f	134	ã	166	≡	198	ã	230	μ
7	•	39	'	71	G	103	g	135	ç	167	•	199	ã	231	þ
8	◼	40	(72	H	104	h	136	ê	168	¿	200	Ⓛ	232	þ
9	◊	41)	73	I	105	i	137	ë	169	Ⓢ	201	Ⓛ	233	ú
10	⊗	42	*	74	J	106	j	138	è	170	↵	202	≡	234	û
11	♂	43	+	75	K	107	k	139	ï	171	½	203	Ⓙ	235	ü
12	♀	44	,	76	L	108	l	140	î	172	¼	204	Ⓛ	236	ý
13	♯	45	_	77	M	109	m	141	ì	173	¡	205	=	237	ÿ
14	♪	46	.	78	N	110	n	142	ñ	174	«	206	Ⓛ	238	˘
15	✳	47	/	79	O	111	o	143	ñ	175	»	207	Ⓢ	239	˙
16	▶	48	0	80	P	112	p	144	é	176	▒	208	Ⓢ	240	˚
17	◀	49	1	81	Q	113	q	145	æ	177	▒	209	Ⓢ	241	±
18	‡	50	2	82	R	114	r	146	ff	178	▒	210	ê	242	=
19	!!	51	3	83	S	115	s	147	ô	179		211	ë	243	¼
20	¶	52	4	84	T	116	t	148	ö	180	†	212	è	244	¶
21	⊗	53	5	85	U	117	u	149	ò	181	á	213	í	245	⊗
22	—	54	6	86	V	118	v	150	û	182	â	214	î	246	÷
23	‡	55	7	87	W	119	w	151	ù	183	à	215	ï	247	˘
24	↑	56	8	88	X	120	x	152	ÿ	184	Ⓢ	216	ÿ	248	•
25	↓	57	9	89	Y	121	y	153	ö	185	Ⓛ	217	J	249	˙
26	→	58	:	90	Z	122	z	154	ü	186	Ⓛ	218	Γ	250	˙
27	←	59	;	91	[123	{	155	ø	187	Ⓙ	219	■	251	¡
28	└	60	<	92	\	124		156	£	188	≡	220	■	252	£
29	+	61	=	93]	125	}	157	Ø	189	Ç	221	¡	253	£
30	▲	62	>	94	^	126	~	158	×	190	¥	222	ì	254	■
31	▼	63	?	95	_	127	△	159	f	191	Γ	223	■	255	

Key Codes



Key Codes

Key	Scan Code		ASCII or Extended†			ASCII or Extended† with SHIFT			ASCII or Extended† with CTRL			ASCII or Extended† with ALT		
	Dec	Hex	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
ESC	1 01	27	1B	ESC	27	1B	ESC	27	1B	ESC	1 01	NUL	§	
!	2 02	49	31	!	33	21	!				120 78	NUL		
@	3 03	50	32	@	64	40	@	3 03	NUL		121 79	NUL		
#	4 04	51	33	#	35	23	#				122 7A	NUL		
\$	5 05	52	34	\$	36	24	\$				123 7B	NUL		
%	6 06	53	35	%	37	25	%				124 7C	NUL		
^	7 07	54	36	^	94	5E	^	30 1E	RS		125 7D	NUL		
&	8 08	55	37	&	38	26	&				126 7E	NUL		
*	9 09	56	38	*	42	2A	*				127 7F	NUL		
(10 0A	57	39	(40	28	(128 80	NUL		
)	11 0B	48	30)	41	29)				129 81	NUL		
_	12 0C	45	2D	_	95	5F	_	31 1F	US		130 82	NUL		
=	13 0D	61	3D	=	43	2B	=				131 83	NUL		
BKSP	14 0E	8	08		8	08		127 7F			14 0E	NUL	§	
TAB	15 0F	9	09		15	0F	NUL	148 94	NUL	§	15 A5	NUL	§	
Q	16 10	113	71	q	81	51	Q	17 11	DC1		16 10	NUL		
W	17 11	119	77	w	87	57	W	23 17	ETB		17 11	NUL		
E	18 12	101	65	e	69	45	E	5 05	ENQ		18 12	NUL		
R	19 13	114	72	r	82	52	R	18 12	DC2		19 13	NUL		
T	20 14	116	74	t	84	54	T	20 14	SO		20 14	NUL		
Y	21 15	121	79	y	89	59	Y	25 19	EM		21 15	NUL		
U	22 16	117	75	u	85	55	U	21 15	NAK		22 16	NUL		
I	23 17	105	69	i	73	49	I	9 09	TAB		23 17	NUL		
O	24 18	111	6F	o	79	4F	O	15 0F	SI		24 18	NUL		
P	25 19	112	70	p	80	50	P	16 10	DLE		25 19	NUL		
[26 1A	91	5B	[123	7B	{	27 1B	ESC		26 1A	NUL	§	
]	27 1B	93	5D]	125	7D	}	29 1D	GS		27 1B	NUL	§	
ENTER	28 1C	13	0D	CR	13	0D	CR	10 0A	LF		28 1C	NUL	§	
L CTRL	28 1C	13	0D	CR	13	0D	CR	10 0A	LF		166 A6	NUL	§	
R CTRL	29 1D													
A	30 1E	97	61	a	65	41	A	1 01	SOH		30 1E	NUL		
S	31 1F	115	73	s	83	53	S	19 13	DC3		31 1F	NUL		
D	32 20	100	64	d	68	44	D	4 04	EOT		32 20	NUL		
F	33 21	102	66	f	70	46	F	6 06	ACK		33 21	NUL		
G	34 22	103	67	g	71	47	G	7 07	BEL		34 22	NUL		
H	35 23	104	68	h	72	48	H	8 08	BS		35 23	NUL		
J	36 24	106	6A	j	74	4A	J	10 0A	LF		36 24	NUL		
K	37 25	107	6B	k	75	4B	K	11 0B	VT		37 25	NUL		
L	38 26	108	6C	l	76	4C	L	12 0C	FF		38 26	NUL		
:	39 27	59	3B	:	58	3A	:				39 27	NUL	§	
"	40 28	39	27	"	34	22	"				40 28	NUL	§	
~	41 29	96	60	~	126	7E	~				41 29	NUL	§	
L SHIFT	42 2A													
\	43 2B	92	5C	\	124	7C		28 1C	FS					
Z	44 2C	122	7A	z	90	5A	Z	26 1A	SUB		44 2C	NUL		
X	45 2D	120	78	x	88	58	X	24 18	CAN		45 2D	NUL		
C	46 2E	99	63	c	67	43	C	3 03	ETX		46 2E	NUL		
V	47 2F	118	76	v	86	56	V	22 16	SYN		47 2F	NUL		
B	48 30	98	62	b	66	42	B	2 02	STX		48 30	NUL		
N	49 31	110	6E	n	78	4E	N	14 0E	SO		49 31	NUL		
M	50 32	109	6D	m	77	4D	M	13 0D	CR		50 32	NUL		
, <	51 33	44	2C	,	60	3C	<				51 33	NUL	§	
. >	52 34	46	2E	.	62	3E	>				52 34	NUL	§	
/ ?	53 35	47	2F	/	63	3F	?				53 35	NUL	§	
GRAY /£	53 35	47	2F	/	63	3F	?	149 95	NUL		164 A4	NUL		

Key	Scan Code	ASCII or Extended†			ASCII or Extended† with SHIFT			ASCII or Extended† with CTRL			ASCII or Extended† with ALT		
	Dec Hex	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
R SHIFT	54 36												
* PRTSC	55 37	42	2A	*	PRTSC		††	114	72	0			
L ALT	56 38												
R ALT£	56 38												
SPACE	57 39	32	20	SPC	32	20	SPC	32	20	SPC	32	20	SPC
CAPS	58 3A												
F1	59 3B	59	3B	NUL	84	54	NUL	94	5E	NUL	104	68	NUL
F2	60 3C	60	3C	NUL	85	55	NUL	95	5F	NUL	105	69	NUL
F3	61 3D	61	3D	NUL	86	56	NUL	96	60	NUL	106	6A	NUL
F4	62 3E	62	3E	NUL	87	57	NUL	97	61	NUL	107	6B	NUL
F5	63 3F	63	3F	NUL	88	58	NUL	98	62	NUL	108	6C	NUL
F6	64 40	64	40	NUL	89	59	NUL	99	63	NUL	109	6D	NUL
F7	65 41	65	41	NUL	90	5A	NUL	100	64	NUL	110	6E	NUL
F8	66 42	66	42	NUL	91	5B	NUL	101	65	NUL	111	6F	NUL
F9	67 43	67	43	NUL	92	5C	NUL	102	66	NUL	112	70	NUL
F10	68 44	68	44	NUL	93	5D	NUL	103	67	NUL	113	71	NUL
F11£	87 57	133	85	E0	135	87	E0	137	89	E0	139	8B	E0
F12£	88 58	134	86	E0	136	88	E0	138	8A	E0	140	8C	E0
NUM	69 45												
SCROLL	70 46												
HOME	71 47	71	47	NUL	55	37	7	119	77	NUL			
HOME£	71 47	71	47	E0	71	47	E0	119	77	E0	151	97	NUL
UP	72 48	72	48	NUL	56	38	8	141	8D	NUL§			
UP£	72 48	72	48	E0	72	48	E0	141	8D	E0	152	98	NUL
PGUP	73 49	73	49	NUL	57	39	9	132	84	NUL			
PGUP£	73 49	73	49	E0	73	49	E0	132	84	E0	153	99	NUL
GRAY -	74 4A				45	2D	-						
LEFT	75 4B	75	4B	NUL	52	34	4	115	73	NUL			
LEFT£	75 4B	75	4B	E0	75	4B	E0	115	73	E0	155	9B	NUL
CENTER	76 4C				53	35	5						
RIGHT	77 4D	77	4D	NUL	54	36	6	116	74	NUL			
RIGHT£	77 4D	77	4D	E0	77	4D	E0	116	74	E0	157	9D	NUL
GRAY +	78 4E				43	2B	+						
END	79 4F	79	4F	NUL	49	31	1	117	75	NUL			
END£	79 4F	79	4F	E0	79	4F	E0	117	75	E0	159	9F	NUL
DOWN	80 50	80	50	NUL	50	32	2	145	91	NUL§			
DOWN£	80 50	80	50	E0	80	50	E0	145	91	E0	160	A0	NUL
PGDN	81 51	81	51	NUL	51	33	3	118	76	NUL			
PGDN£	81 51	81	51	E0	81	51	E0	118	76	E0	161	A1	NUL
INS	82 52	82	52	NUL	48	30	0	146	92	NUL§			
INS£	82 52	82	52	E0	82	52	E0	146	92	E0	162	A2	NUL
DEL	83 53	83	53	NUL	46	2E	.	147	93	NUL§			
DEL£	83 53	83	53	E0	83	53	E0	147	93	E0	163	A3	NUL

- † Extended codes return 0 (NUL) or E0 (decimal 224) as the initial character. This is a signal that a second (extended) code is available in the keystroke buffer.
- § These key combinations are only recognized on extended keyboards.
- £ These keys are only available on extended keyboards. Most are in the Cursor/Control cluster. If the raw scan code is read from the keyboard port (60h), it appears as two bytes (E0h) followed by the normal scan code. However, when the keypad ENTER and / keys are read through the BIOS interrupt 16h, only E0h is seen since the interrupt only gives one-byte scan codes.
- †† Under DOS, SHIFT+PRTSCR causes interrupt 5, which prints the screen.

Glossary

386 enhanced mode A mode in Windows that runs on the 80386 and 80486 processors. It provides access to extended memory and the ability to run non-Windows applications. This and standard mode are both referred to as protected mode in Windows and offer more capability than Windows real mode.

8086 family of processors All processors in the Intel 8086 family, including the 8086, 80286, 80386, and 80486 CPU chips.

8087 family of math processors All math processors (also called math coprocessors) in the Intel 8087 family, including the 8087, 80287, and 80387 chips. These processors perform high-speed floating-point and binary-coded-decimal number processing. The 80486 chip includes a math processor.

8087 window The CodeView window in which the floating-point math processor's registers are displayed. This window remains empty until a math processor instruction is executed. If the program uses the Microsoft math processor emulator library, the contents of the emulated math processor's registers are displayed.

A

actual parameter See "argument."

adapter A printed-circuit card that plugs into a computer and controls a device, such as a video display or a printer.

address The memory location of a data item or procedure, or an expression that evaluates to an address. In CodeView, the expression can represent just the offset (a default segment is assumed), or it can be in *segment:offset* format.

address range A range of memory bounded by two addresses.

anonymous allocation Assignment to a segment at link time.

ANSI (American National Standards Institute) The institute responsible for defining programming-language standards to promote portability of languages between different computer systems.

ANSI character set An 8-bit character set that contains 256 characters. See "ASCII character set."

API (application programming interface) A set of system-level routines that can be used in an application program for tasks such as input, output, and file management. In a graphics-oriented operating environment like Microsoft Windows, high-level support for video graphics output is part of the API.

argc The conventional name for the first argument to the **main** function in a C source program (an integer specifying the number of arguments passed to the program from the command line).

argument A value passed to a routine or specified with an option in the command line for a utility. Also called an actual parameter. See also "parameter."

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 of the same type.

ASCII character set The American Standard Code for Information Interchange 8-bit character set, consisting of the first 128 (0 to 127) characters of the ANSI character set. The term ASCII characters is sometimes used to mean all 256 characters defined for a particular system, including the extended ASCII characters. ASCII values represent letters, digits, special symbols, and other characters. See also “extended ASCII.”

ASCII file See “text file.”

.ASM The extension for an assembly-language source file.

Assembly mode The mode in which CodeView displays the assembly-language equivalent of the machine code being executed. CodeView disassembles the executable file in memory to obtain the code.

automatic data segment See “DGROUP.”

available memory The portion of conventional memory not used by system software, TSR utilities, or other programs.

B

.BAK The extension that is often used to indicate a backup file.

.BAS The extension for a Basic language source file.

base name The part of a filename before the extension, usually 1 to 8 characters. For example, README is the base name of the filename README.TXT.

.BAT The extension for a DOS batch file.

batch file A file containing operating-system commands that can be run from the command line. Also called a command file.

binary file A file that contains numbers in binary, machine-readable form. For example, an executable file is a binary file.

binary operator An operator that takes two operands.

BIOS (basic input/output system) The code built into system memory that provides hardware interface routines for programs. You can trace into the BIOS with CodeView when using Assembly mode.

.BMP The extension for a bitmap file.

breakpoint A specified address where program execution halts. CodeView interrupts execution when the program reaches the address where a breakpoint is set. See also “conditional breakpoint.”

.BSC The extension for a database file for use with the Source Browser. A .BSC file is created by BSCMAKE.

buffer An area in memory that holds data temporarily, most often during input/output operations.

C

.C The extension for a C source file.

call gate A special descriptor-table entry that describes a subroutine entry point rather than a memory segment. A far call to a call gate selector transfers to the entry point specified in the call gate. This is a feature of the 80286–80486 hardware and is typically used to provide a transition from a lower privilege state to a higher one.

case sensitivity The distinction made between uppercase and lowercase letters. For example, “MyFile” and “MYFILE” are considered to be different strings in a case-sensitive situation but are understood to be the same string if case is not sensitive.

CGA (color graphics adapter) A video adapter capable of displaying text characters or graphics pixels in low resolution in up to 16 colors.

character string A sequence of bytes treated as a set of ASCII letters, numbers, and other symbols. A character string is often enclosed in single quotation marks (' ') or double quotation marks (" ").

child process A process created by another process (its parent process).

click To press and release quickly one of the mouse buttons (usually the left button) while pointing the mouse pointer to an object on the screen.

clipboard A temporary storage area for text. The clipboard is used for cut, copy, and paste operations.

.COB The extension for a COBOL source file.

code symbol The address of a routine.

.COM The extension for a DOS executable file that contains a single segment. Tiny-model programs have a .COM extension. See also “tiny memory model.”

command An instruction you use to control a computer program, such as an operating system or application.

command file A file containing operating-system commands that can be run from the command line. If the file’s extension is .BAT, the command file contains DOS commands. Also called a batch file.

command file (in NMAKE) A text file containing input expected by utilities such as NMAKE.

compact memory model A program with one code segment and multiple data segments.

compile To translate programming language statements into a form that can be executed by the computer.

conditional breakpoint A breakpoint that is taken when a specified expression becomes nonzero (true). A conditional breakpoint is evaluated after every instruction is executed unless an address is also specified. Formerly called tracepoint and watchpoint.

constant A value that does not change during program execution.

constant expression Any expression that evaluates to a constant. It may include integer constants, character constants, floating-point constants, enumeration constants, type casts to integral and floating-point types, and other constant expressions. It cannot include a variable or function call.

conventional memory The first 640K (or sometimes 1MB) of memory under MS-DOS. Also called low memory.

coprocessor See “8087 family of math processors.”

.CPP The extension for a C++ source file.

CPU (central processing unit) The main processor in a computer. For example, the CPU that receives and carries out instructions in the PC/AT is an 80286 processor. See also “8086 family of processors.”

CS:IP The address of the current program location. This is the address of the next instruction to be executed. CS is the value of the Code Segment register, and IP is the value of the Instruction Pointer register.

cursor The thin blinking line or other character that represents the location of typed input or mouse activity.

.CXX The extension for a C++ source file.

D

.DAT The extension that is often used to indicate a data file.

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.

.DBG The extension for a file that is created by LINK when the /CO and /TINY options are used. The file contains symbolic debugging information.

debugger A program that allows the programmer to execute a program one line or instruction at a time. The debugger displays the contents of registers and memory to help locate the source of problems in the program. An example is the Microsoft CodeView debugger.

debugging information Symbolic information used by a debugger, especially information in the Microsoft Symbolic Debugging Information format that is used by the Microsoft CodeView debugger.

.DEF The extension for a module-definition file.

default data segment See “DGROUP.”

default library A standard library that contains routines and data for a language. The language’s compiler embeds the name of the default library in the object file in a COMMENT record. The embedded name tells LINK to search the default library automatically.

DGROUP The group that contains the segments called `_DATA` (initialized data), `CONST` (constant data), `_BSS` (uninitialized data), and `STACK` (the program’s stack). Also called default (or automatic) data segment.

dialog box A box that appears when you choose a command that requires additional information.

disassemble To translate binary machine code into the equivalent assembly-language representation. Also called unassemble.

disassembly The assembly-language representation of machine code, obtained by disassembling the machine code.

.DLL The extension for a dynamic-link library.

DLL A dynamic-link library.

.DOC The extension that is often used to indicate a document file.

DOS application A program that runs only under DOS. A DOS executable file contains a header and one contiguous block of segments.

DOS-extended An application that is able to be run by the DOS Extender in extended or expanded memory.

DOS Extender A program that lets an application run in extended or expanded memory.

DOS session Under Windows, a full-screen emulation of the MS-DOS environment started using the DOS Prompt in the Program Manager Main Group. The DOS Prompt program item starts a copy of the MS-DOS command interpreter (COMMAND.COM).

double precision A real (floating-point) numeric value that occupies eight bytes of memory. Double-precision values are accurate to 15 or 16 digits.

DPMI A server that provides extended or expanded memory. Examples of DPMI servers include a DOS session under Windows and Microsoft’s MSDPMI.EXE.

drag To move the mouse while holding down one of its buttons.

dump To display the contents of memory at a specified memory location.

dynamic link A method of postponing the resolution of external references until load time or run time. A dynamic link allows the called routines to be created, distributed, and maintained independently of their callers.

dynamic-link library A file, usually with a .DLL extension, that contains the binary code for routines and data that are linked to a program at run time.

E

EGA (enhanced graphics adapter) A video adapter capable of displaying all the modes of the color graphics adapter (CGA) plus additional modes in medium resolution in up to 64 colors.

EMM386.EXE An example of a VCPI server. EMM386.EXE simulates expanded memory in extended memory for an 80386 or higher processor.

EMS Expanded Memory Specification. See “expanded memory.”

emulator A floating-point math package that provides software emulation of the operations of a math processor.

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 memory area, defined by the operating system, that stores environment variables and their values.

environment variable A string associated with an identifier and stored by the operating system.

Environment variables are defined by the SET command. The identifier and the string associated with it can be used by a program.

.ERR The extension for a file of error-message text or error output.

error code See “exit code.”

escape sequence A specific combination of an escape character (often a backslash) followed by a character, keyword, or code. Escape sequences often represent white space, nongraphic characters, or literal delimiters within strings and character constants.

.EXE One of the extensions for an executable file, which is a file that can be loaded and executed by the operating system.

executable file A program ready to be run by an operating system, usually with one of the extensions .EXE, .COM, or .BAT. When the name of the file is typed at the system prompt, the statements in the file are executed.

exit code An integer returned by a program to the operating system or the program’s caller after completion to indicate the success, failure, or status of the program. Also called a return code or error code.

Exit code also refers to the executable code that a compiler places in every program to terminate execution of the program. This code typically closes open files and performs other housekeeping chores. When a program terminates in CodeView, the current line is in the exit code. No source code is shown since none is available. See also “startup code.”

expanded memory Memory above 640K made available to real-mode programs and controlled through paging by an expanded memory manager.

expanded memory emulator A device driver that allows extended memory on computers with an 80286 or later processor to behave like expanded memory.

expanded memory manager (EMM) A device driver for controlling expanded memory.

explicit allocation Assignment to a segment at compile time.

expression A combination of operands and operators that yields a single value.

extended ASCII ASCII codes between 128 and 255. The meanings of extended ASCII codes differ depending on the system.

extended dictionary A summary of the definitions contained in all modules of a standard library. LINK uses extended dictionaries to search libraries faster.

extended memory Memory above either 640K or 1 megabyte made available to protected-mode programs on computers with an 80286 or later processor. Extended memory is used by Windows in standard mode or 386 enhanced mode.

extended memory manager A device driver for controlling extended memory, for example, HIMEM.SYS for Windows.

extender-ready See “DOS-extended.”

extension One, two, or three characters that appear after a period (.) following the base name in a filename. For example, .TXT is the extension of the filename README.TXT. A filename does not necessarily have an extension. Sometimes the extension is considered to include the preceding period.

external reference A routine or data item declared in one module and referenced in another.

F

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. Also called a segmented address. See also “address” and “near address.”

FAT (file allocation table) The standard file system for MS-DOS.

fatal error An error that causes a program to terminate immediately.

.FD The extension for a declaration file (a type of include file) in FORTRAN.

.FI The extension for an interface file (a type of include file) in FORTRAN.

file handle A value returned by the operating system when a file is opened and used by a program to refer to the file when communicating to the system. Under MS-DOS, COMMAND.COM opens the first five file handles as **stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn**.

filename A string of characters identifying a file on disk, composed of a base name optionally followed by a period (.) and an extension. A filename may be preceded by a path. For example, in the filename README.TXT, .TXT is the extension and README is the base name.

fixup The linking process that resolves a reference to a relocatable or external address.

flags register 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 CodeView, the current values of the most commonly used

bits of the flags register are shown at the bottom of the Register window.

flat memory model A nonsegmented memory model that can address up to four gigabytes of memory.

flipping A screen-exchange method that uses the video pages of the CGA or EGA to store both the debugging and output screens. When you request the other screen, the two video pages are exchanged. See also “screen exchange” and “swapping.”

.FOR The extension for a FORTRAN source file.

formal parameter See “parameter.”

frame The segment, group, or segment register that specifies the segment part of an address.

full-screen application A program that runs under Windows but cannot run in a window.

function A routine that returns a value.

function call An expression that invokes a function and passes arguments (if any) to the function.

G

gigabyte (GB) 1024 megabytes or 1,073,741,824 bytes (2 to the power of 30).

global symbol A symbol that is available throughout the entire program. In CodeView, function names are always global symbols. See also “local symbol.”

grandparent process The parent of a process that created a process.

group A collection of segments having the same segment base address.

H

.H The extension for an include (or header) file in C.

HELPPFILES The environment variable used by a program to find .HLP files.

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 easily converted to and from binary, the base-2 numbering system the computer itself uses.

highlight To select an area in a text box, window, or menu as a command or as text to be deleted or copied. A highlight is shown in reverse-video or a bright color.

high memory Memory between the 640K of conventional memory and the 1MB limit of a PC’s address space.

HIMEM.SYS An example of an XMS server. HIMEM.SYS manages extended memory for an 80286 or higher processor.

.HLP The extension for a help file created by HELPMMAKE.

HPFS (high-performance file system) An installable file system that uses disk caching and that allows filenames to be longer and to contain certain nonalphanumeric characters.

huge memory model A program with multiple code and data segments, and data items that can exceed 64K.

I

identifier A name that identifies a register or a location in memory and usually represents a program element such as a constant, variable, type, or routine. The terms identifier and symbol are used synonymously in most documentation.

IEEE format (Institute for Electrical and Electronic Engineers) A standard for representing floating-point numbers, performing math with them, and handling underflow/overflow conditions. The 8087 family of math processors and the Microsoft emulator library implement this format.

import library A library, created by IMPLIB, that contains entry points in DLLs. It does not contain the actual code for routines and data. An import library is used to resolve references at link time in the same way as a standard library; each is a type of static library. See “dynamic-link library” and “standard library.”

.INC The extension for an include file in Microsoft Macro Assembler.

include file A file that is merged into a program’s source code with a file-inclusion command. In C, this command is the **#include** preprocessor directive. In FORTRAN, it is the **INCLUDE** keyword or the **\$INCLUDE** meta-command. In Microsoft Macro Assembler, the equivalent command is the **INCLUDE** directive. In a .DEF file, the **INCLUDE** statement performs this action. In an NMAKE makefile, it is the **!INCLUDE** directive.

.INI The extension for an initialization file.

INIT The environment variable usually used by a program to find an initialization file.

installable file system A file system that exists in addition to the FAT file system.

integer In CodeView and the C language, a whole number represented as a 16-bit two’s complement binary number that has a range from -32,768 through +32,767. See also “long integer.”

interoverlay call A call from a function in one overlay to a function in another overlay, represented internally by an entry in a thunk table.

interrupt call A machine-level procedure that can be called to execute a BIOS, DOS, or other function. You can trace into BIOS interrupt-service routines with CodeView, but not into the DOS interrupt (0x21).

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 library.

K

kilobyte (K) 1024 bytes (2 to the power of 10).

L

label An identifier representing an address.

large memory model A program with multiple code and data segments.

LIB The environment variable used by LINK to find default libraries.

.LIB The extension for a static library.

library A collection of routines or data made available to one or more programs through static or dynamic linking.

LIM EMS Lotus/Intel/Microsoft Expanded Memory Specification.

LINK The environment variable used by LINK for command-line options.

linking The process in which the linker resolves all external references by searching the run-time and user libraries and then computes absolute offset addresses for these references. The linking process results in a single executable file.

list file A text file of information produced by a utility such as LIB. See “map file.”

listing A generic term for a map, list, or cross-reference file.

.LNK The extension that is often used to indicate a response file.

load library A static library specified to the linker as an object file, causing all modules in the library to be linked into the program. See “static library.”

local symbol An identifier that is visible only within a particular routine. See “global symbol.”

Local window The CodeView window in which the local variables for the current routine are displayed.

logical segment A segment defined in an object module. Each physical segment other than DGROUPE contains exactly one logical segment, except when you use the **GROUP** directive in a Microsoft Macro Assembler module. (Linking with the **/PACKC** option can also create more than one logical segment per physical segment.)

long integer In CodeView and the C language, a whole number represented 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.”

low memory See “conventional memory.”

.LRF The extension that is often used to indicate a response file. PWB creates response files with the .LRF extension.

.LST The extension that is often used to indicate a list file.

l-value An expression (such as a variable name) that refers to a single memory location and is required as the left operand of an assignment operation or the single operand of a unary operator. For example, x_1 is an l-value, but x_1+x_2 is not.

M

machine code A series of binary numbers that a processor executes as program instructions. See also “disassemble.”

macro A block of text or instructions that has been assigned an identifier. For example, you can create a macro that contains a set of functions that you perform repeatedly and assign the macro to a single keystroke.

.MAK The extension that is often used to indicate a makefile or description file.

.MAP The extension for a map file.

map file A text file of information produced by a utility such as LINK. Also called a list file or listing.

math coprocessor See “8087 family of math processors.”

MB Megabyte.

MDI Multiple Document Interface.

medium memory model A program with multiple code segments and one data segment.

megabyte (MB) 1024 kilobytes or 1,048,576 bytes (2 to the power of 20).

memory model A convention for specifying the number of code and data segments in a program. Memory models include tiny, small, medium, compact, large, huge, and flat.

memory-resident program See “terminate-and-stay-resident.”

menu bar The bar at the top of a display containing menus.

Mixed mode The CodeView source display mode that shows each source line of the program

being debugged, followed by a disassembly of the machine code generated for that source line. This mode combines Source mode and Assembly mode.

modification time See “time stamp.”

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.

module (in LIB) See “object module.”

module-definition file A text file, usually with a .DEF extension, that describes characteristics of a program. A module-definition file is used by LINK and by IMPLIB.

monochrome adapter A video adapter capable of displaying only in medium resolution in one color. Most monochrome adapters display text only; individual graphics pixels cannot be displayed.

mouse pointer The reverse-video or colored square that moves to indicate the current position of the mouse. The mouse pointer appears only if a mouse is installed.

MS32EM87.DLL A DLL required by the DOS Extender. The SYSTEM environment variable must be set to the directory that contains this file.

MS32KRNL.DLL A DLL required by the DOS Extender. The SYSTEM environment variable must be set to the directory that contains this file.

MSDPMI An example of a DPMI server. MSDPMI supports 32-bit interrupt services.

MSDPMI.INI The initialization file used by MSDPMI.

multitasking operating system An operating system in which two or more programs or threads can execute simultaneously.

N

NAN An acronym for “not a number.” The math processors generate NANs when the result of an operation cannot be represented in IEEE format.

near address A memory location specified by only the offset from the start of the segment. A near address requires only two bytes. See also “address” and “far address.”

newline character The character used to mark the end of a line in a text file, or the escape sequence (\n in C language) used to represent this character.

null character The ASCII character whose value is 0, or the escape sequence (\0 in C language) used to represent this character.

null pointer A pointer to nothing, expressed as the integer value 0.

O

.OBJ The extension for an object file produced by a compiler or assembler.

object file A file produced by compiling or assembling source code, containing relocatable machine code.

object module A group of routines and data items stored in a standard library, originating from an object file. See also “standard library.”

object module format The specification for the structure of object files. Microsoft languages conform to the Microsoft Relocatable Object-Module Format (OMF), which is based on the Intel 8086 OMF.

offset The number of bytes from the beginning of a segment or other address to a particular byte.

OMF Object module format.

Output screen The CodeView screen that contains program output. To switch to this screen, choose the Output command from the View menu or press F4.

overlay A program component loaded into memory only when needed.

P

packaged function A function that exists in an object file as a COMDAT record. Packaged functions allow function-level linking. Functions that are not packaged can be linked only at the object level.

parameter A data item expected by a routine or information expected in the command line for a utility. Also called a formal parameter. See also “argument.”

parent process A process that creates another process, called the child process.

.PAS The extension for a Pascal source file.

path A specification of the location of a file or a directory. A path consists of one or more directory names and may include a drive (or device) specification. For example, C:\PROJECT\PROJLIBS is the path to a sub-directory called PROJLIBS in a directory called PROJECT that is located on the C drive. Sometimes “path” refers to multiple path specifications, each separated by a semicolon (;). In certain circumstances, a path specification must include a trailing backslash; for example, specify C:\PROJECT\PROJLIBS\ to tell LINK the location of the PROJLIBS directory containing additional libraries.

.PCH The extension for a precompiled C header (or include) file.

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.

PID (process identification number) A unique code that the operating system assigns to a process when the process is created. The PID may be any value except 0.

pointer A variable containing an address or offset.

pop-up menu A menu that appears when you click the menu title with the mouse or press the ALT key and the first letter of the menu 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.

privileged mode A special execution mode (also known as ring 0) supported by the 80286–80486 hardware. Code executing in this mode can execute restricted instructions that are used to manipulate system structures and tables. Device drivers run in this mode.

procedure A routine that does not return a value.

procedure call A call to a routine that performs a specific action.

process Generally, any executing program or code unit. This term implies that the program or unit is one of a group of processes executing independently.

processor See “CPU (central processing unit).”

program step To trace the next source line in Source mode or the next instruction in Mixed mode or 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 also “trace.”

protected mode The operating mode of the 80286–80486 processors that allows the operating system to protect one application from another.

protected mode (in Windows) Either of two modes in Windows 3.0: standard mode or 386 enhanced mode. See also “standard mode” and “386 enhanced mode.”

Q

.QLB The extension for a Quick library.

R

radix The base of a number system. In CodeView, numbers can be entered in three radices: 8 (octal), 10 (decimal), or 16 (hexadecimal). The default radix is 10.

RAM Random access memory. Usually refers to conventional memory.

.RC The extension for a resource script file. An .RC file defines resources for an application such as icons, cursors, menus, and dialog boxes. The Microsoft Windows Resource Compiler compiles an .RC file to create an .RES file.

real mode The operating mode of the 80286–80486 processors that runs programs designed for the 8086/8088 processor. All programs for the DOS environment run in real mode.

real mode (in Windows) An operating mode that provides compatibility with versions of Windows applications prior to 3.0. Real mode is the only mode of Windows 3.0 available for computers with less than 1 megabyte of extended memory.

redirection The process of causing a command or program to take its input from a file or device other than the keyboard (standard input), or causing the output of a command or program to be sent to a file or device other than the display (standard output). The operating-system redirection symbols are the greater-than (>) and less-than (<) signs.

The same symbols are used in the CodeView Command window to redirect input and output of the debugging session. In addition, the equal sign (=) can be used to redirect both input and output.

Register window The CodeView window in which the CPU registers and the bits of the flags register are displayed.

registers Memory locations in the processor that temporarily store data, addresses, and logical values. See also “flags register.”

regular expression A text expression that specifies a pattern of text to be matched (as opposed to matching specific characters). CodeView supports a subset of the regular-expression characters used in the XENIX and UNIX operating systems. PWB supports both the full UNIX syntax and an extended Microsoft syntax for regular expressions.

relocatable Not having an absolute address.

.RES The extension for a file produced by the Microsoft Windows Resource Compiler from an .RC file.

response file A text file containing input expected by utilities such as LINK and LIB. Commonly used extensions for response files include .LRF, .LNK, and .RSP.

return code See “exit code.”

ROM Read-only memory.

root In an overlaid DOS program, the part of the program that always remains in memory. Also called the root overlay.

routine A generic term for a procedure, function, or subroutine.

.RSP The extension that is often used to indicate a response file.

RTF Rich text format.

run-time error A math or logic error that occurs during execution of a program. A run-time error often results in termination of the program.

S

.SBR The extension for a file used by BSCMAKE to create a .BSC file.

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 also "flipping" and "swapping."

scroll To move text up, down, left, and right in order to see parts that cannot fit on the screen.

segment A section of memory containing code or data, limited to 64K for 16-bit segments or 4 gigabytes for 32-bit segments. Also refers to the starting address of that memory area.

segmented executable file The executable file format of a Windows application or DLL. A segmented executable file contains a DOS header, a new .EXE header, and multiple relocatable segments.

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.

shell To gain access to the operating-system command line without actually leaving the PWB or CodeView environment or losing the current context. You can execute operating-system commands and then return to the environment.

single precision A real (floating-point) value that occupies four bytes of memory. Single-precision values are accurate to six or seven decimal places.

small memory model A program with one code segment and one data segment.

SMARTDRV.SYS A driver that creates a disk cache in extended or expanded memory.

source file A text file containing the high-level description that defines a program.

Source mode The mode in which CodeView displays the source code that corresponds to the machine code being executed.

stack A dynamically expanding and shrinking 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 routine's local and temporary variables, arguments, and control information.

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. A trace of the stack shows the currently active functions and

the values of their arguments. See also “stack frame.”

standard error The device to which a program sends error messages. COMMAND.COM opens standard error with a file handle named **stderr**. The default device is the display (CON). Standard error cannot be redirected.

standard input The device from which a program reads input. COMMAND.COM opens standard input with a file handle named **stdin**. The default device is the keyboard (CON). Standard input can be redirected using a redirection symbol (<).

standard library A library created by LIB that contains compiled routines and data. It is used to resolve references at link time.

standard mode The normal Windows 3.0 operating mode that runs on the 80286–80486 processors. This and 386 enhanced mode are both referred to as protected mode in Windows and offer more capability than Windows real mode.

standard output The device to which a program sends output. COMMAND.COM opens standard output with a file handle named **stdout**. The default device is the display (CON). Standard output can be redirected using a redirection symbol.

startup code The code placed at the beginning of a program to control execution of the program code. When CodeView is loaded, the first source line executed runs the entire startup code. If you switch to Assembly mode before executing any code, you can trace through the startup code. See also “exit code.”

static library A library used for resolving references at link time. A static library can be either a standard library or an import library. See also “standard library” and “import library.”

static linking The combining of multiple object and library files into a single executable file with all external references resolved at link time.

status bar The bar at the bottom of the CodeView or PWB display containing status information and command buttons or a short description of the dialog or menu item currently displayed.

stderr See “standard error.”

stdin See “standard input.”

stdout See “standard output.”

string A contiguous sequence of characters, often identified by a symbolic name as a constant or variable.

structure A set of elements which may be of different types, grouped under a single name. See also “user-defined type.”

structure member One of the elements of a structure.

stub file A DOS executable file added to the beginning of a segmented executable file. The stub is invoked if the file is executed under DOS.

subroutine A unit of FORTRAN code terminated by the **RETURN** statement. Program control is transferred to a subroutine with a **CALL** statement.

swapping A screen-exchange method that uses buffers to store the CodeView display and program output screens. When you request the other screen, the two buffers are exchanged. See also “flipping” and “screen exchange.”

symbol See “identifier.”

symbolic debugging information See “debugging information.”

.SYS The extension for a system file or device driver.

SYSTEM An environment variable used by the DOS Extender to find the files MS32EM87.DLL and MS32KRNL.DLL.

T

TEMP The environment variable usually used by a program to find the directory in which to create temporary files. Other programs use the TMP variable in a similar way.

temporary file A file that is created for use by a command while it is running. The file is usually deleted when the command is completed. Most programs create temporary files in the directory indicated by the TMP or TEMP environment variable.

terminate-and-stay-resident (TSR) A DOS program that remains in memory and is ready to respond to an interrupt.

ternary operator An operator that takes three operands. For example, the C-language ? operator.

text file A file containing only ASCII characters in the range of 1 to 127.

thread An operating-system mechanism that allows more than one path of execution through the same instance of a program.

thread ID The name or handle of a particular thread within a process.

thread of execution The sequence of instructions executed by the CPU in a single logical stream. In DOS, there is only one thread of execution.

thunk An interoverlay call in an overlaid DOS program.

time stamp The time of the last write operation to the file. Sometimes the term time stamp refers to the combination of the date and time of the last write operation. Also called modification time.

tiny memory model A program with a single segment holding both code and data, limited to 64K, with the extension .COM.

TMP The environment variable usually used by a program to find the directory in which to create temporary files. Other programs use the TEMP variable in a similar way.

.TMP The extension that is often used to indicate a temporary file.

toggle A feature with two states. Often used to describe a command that turns a feature on if it is off, and off if it is on. When used as a verb, “toggle” means to reverse the state of a feature.

TOOLS.INI A file that contains initialization information for Microsoft tools such as PWB, CodeView, and NMAKE.

trace To execute a single line or instruction. The next source line is traced in Source mode and the next instruction is traced 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. CodeView is ready to execute the next instruction inside the call. See also “program step.”

tracepoint (obsolete) A breakpoint that is taken when an expression, variable, or range of memory changes. This is now a type of conditional breakpoint. See also “conditional breakpoint.”

TSR See “terminate-and-stay-resident.”

.TXT The extension for a text file.

type cast An operation in which a value of one type is converted to a value of a different type.

type casting Including a type specifier in parentheses in front of an expression to indicate the type of the expression's value.

U

unary operator An operator that takes a single operand.

unassemble To translate binary machine code into the equivalent assembly-language representation. Also called disassemble.

unresolved external A reference to a global or external variable or function that cannot be found either in the modules being linked or in the libraries linked with those modules. An unresolved reference causes a fatal link error.

user-defined type A data type defined by the user. See also "structure."

V

variable A value that may change during program execution.

VCPI Virtual Control Program Interface

VCPI server A server that provides expanded memory. An example of a VCPI server is Microsoft's EMM386.EXE.

VGA (video graphics adapter) A video adapter capable of displaying both text and graphics at medium to high resolution in up to 256 colors.

virtual memory A memory management system that provides more memory to a program than is actually in the system. Virtual memory can consist of a file on disk, extended memory, or expanded memory.

W

watchpoint (obsolete) A breakpoint that is taken when an expression becomes true (nonzero). This

is now a type of conditional breakpoint. See also "conditional breakpoint."

wildcard A character that represents one or more matching characters. DOS wildcards (* and ?) in a filename specification are expanded by COMMAND.COM.

Windows application A program that runs only under Windows.

X

XMS Extended Memory Standard (or Specification). See "extended memory."

XMS server A server that provides extended memory. An example of an XMS server is Microsoft's HIMEM.SYS.

Index

- ! (exclamation point)
 - command modifiers, NMAKE, 662
 - HELPMAKE command, 719
 - preprocessing directives, NMAKE, 688
 - replacing text, PWB, 94
 - Shell Escape command, CodeView, 423, 468–469
- ! command, CodeView, 423, 468–469
- " command, CodeView, 423, 470
- " (quotation marks)
 - character strings, 805
 - CodeView syntax, 340
 - LINK syntax, 565
 - long filenames, NMAKE, 654
 - module statement syntax, 610–611
 - Pause command, CodeView, 423, 470
- # (number sign)
 - custom builds, 59
 - HELPMAKE syntax, 712–713
 - inference rules, NMAKE, 681
 - makefile comments, NMAKE, 654
 - substituting for equal sign, CL, 492
 - Tab Set command, CodeView, 423, 470
 - TOOLS.INI file syntax, 652
 - user-defined macros, NMAKE, 669
- # command, CodeView, 423, 470
- \$ (dollar sign)
 - environment variables, NMAKE, 679
 - filename macros, NMAKE, 672–673
 - literal characters, NMAKE, 653
 - regular expressions, PWB, 93
 - user-defined macros, NMAKE, 669
- % (percent sign)
 - file specifier, NMAKE, 653
 - Filename-Parts Syntax, PWB, 265–266
- & (ampersand)
 - C address operator, 414–415
 - CodeView, 405
 - operations line, extending, 702
- () (parentheses)
 - balancing, PWB, 192–193
 - searching, PWB, 91
- * (asterisk)
 - Comment command, CodeView, 423, 471
 - Copy command, LIB, 705
 - deleting watch expressions, CodeView, 461
 - (asterisk) (*continued*)
 - filename macros, NMAKE, 672–673
 - hyperlink, Microsoft Advisor, 764
 - regular expressions, PWB, 93
 - SBRPACK syntax, 740
 - wildcard operator
 - HELPMAKE syntax, 711
 - NMAKE, 653–654
 - UNDEL syntax, 749
 - * command, CodeView, 423, 471
- + (plus sign)
 - Add command, LIB, 703–704
 - concatenating help files, 772
 - LINK syntax, 565, 567
 - searching, PWB, 92
- , (comma)
 - argument separator, CodeView, 352–353
 - CodeView operator, 405
 - field separation, LIB, 699
 - LINK syntax, 565
 - with context operator, CodeView, 421–422
- (dash)
 - character classes, PWB, 92
 - CL syntax, 488
 - command modifier, NMAKE, 647, 661
 - Delete command, LIB, 700, 704
 - HELPMAKE options, 711
- (minus sign), NMAKE options, 688–689
- * Move command, LIB, 705
- + Replace command, LIB, 704–705
- . command, CodeView, 423, 471
- . (period)
 - Current Location command, CodeView, 423, 471
 - dot directives, NMAKE, 687
 - inference rules, NMAKE, 681
 - line number specifier, CodeView, 365
 - LINK syntax, 565
- ... (ellipsis)
 - call tree, PWB, 101
 - menu commands, PWB, 80, 82, 126
- / command, CodeView, 423, 472–473
- / (slash)
 - CL syntax, 488
 - command line, NMAKE, 647
 - HELPMAKE options, 711

/ (slash) *(continued)*

- LINK syntax, 575
- Search command, CodeView, 361, 423, 472–473
- /2 option, CodeView, 338
- /25 option, CodeView, 338–339
- 386 enhanced mode defined, 803
- /43 option, CodeView, 338–339
- /50 option, CodeView, 339
- 7 command, CodeView, 423, 473–474
- /8 option, CodeView, 338
- 8086 instruction mnemonics, assembling, 424–426
- 8086 processors defined, 803
- 8087 command, CodeView, 373–374, 423, 473–474
- 8087 processors defined, 803
- 8087 window, CodeView
 - defined, 803
 - function, 355–356
 - opening, 374
 - overview, 348
- 8259 interrupt trapping, 341–342
- 8514 display, specifying, CodeView, 338–339
- : (colon)
 - appending device names, CL, 497–498
 - CodeView operator, 405
 - Delay command, CodeView, 423, 475
 - dependency, NMAKE, 656–657
 - dot directives, NMAKE, 687
 - HELPMAKE commands, 722
 - inference rules, NMAKE, 681
 - macro substitution, NMAKE, 677
 - module statement syntax, 610
 - target separator, NMAKE, 655
- : command, CodeView, 423, 475
- :: (scope operator), CodeView precedence, 406
- :< (base operator), CodeView precedence, 406
- :< command, HELPMAKE, 723
- :> command, HELPMAKE, 723
- = command, CodeView, 424, 477
- = (equal sign)
 - module statement syntax, 610
 - Redirect Input and Output command, CodeView, 424, 477
 - substituting for number sign, CL, 492
- ? command, CodeView, 424, 477–478
- ? (question mark)
 - call tree, PWB, 100
 - decorated names, C++, 409
 - Display Expression command, CodeView, 424, 477–478

? (question mark) *(continued)*

- filename macros, NMAKE, 672–673
- SBRPACK syntax, 740
- wildcard operator
 - HELPMAKE syntax, 711
 - NMAKE, 653–654
 - UNDEL, 749
- /? option
 - BSCMAKE, 737
 - CVPACK, 745
 - EXEHDR, 631
 - EXP, 750
 - HELPMAKE, 715
 - IMPLIB, 747
 - LIB, 702
 - LINK, 593
 - NMAKE, 650
 - RM, 748
 - SBRPACK, 740
 - UNDEL, 749
- ?: (conditional operator), CodeView, 405
- ?? command, CodeView, 424, 478–479
- \ (backslash)
 - HELPMAKE syntax, 720–721
 - line continuation character, NMAKE, 655, 660, 669
 - line continuation character, PWB, 115–117, 136
 - LINK syntax, 567
 - regular expressions, PWB, 96
 - Screen Exchange command, CodeView, 424, 479
- \ command, CodeView, 424, 479
- \\ formatting attribute, HELPMAKE, 721
- @ (at sign)
 - BSCMAKE syntax, 738
 - command files, NMAKE, 651
 - command modifier, NMAKE, 661
 - filename macros, NMAKE, 672–673
 - LINK syntax, 573
 - local contexts, HELPMAKE, 720
 - naming registers, CodeView, 401, 419
 - Redraw command, CodeView, 424, 479
- @ command, CodeView, 424, 479
- < (less than operator), Redirect Input command, CodeView, 340, 475
- <> (angle brackets)
 - command buttons, PWB, 80, 82
 - inline files, NMAKE, 664–665
- < command, CodeView, 340, 475
- [] (brackets)
 - balancing, PWB, 192–193
 - call tree, PWB, 100

- > (DOS redirection symbol), HELPMMAKE syntax, 713
- > (greater than operator), Redirect Output command, CodeView, 340, 424, 476
- > command, CodeView, 340, 424, 476
- >> (help delimiter), HELPMMAKE, 726, 728
- ;(semicolon)
 - comments, PWB, 136
 - LINK syntax, 565, 571
 - TOOLS.INI file syntax, 652
- ^ (caret)
 - literal characters, NMAKE, 653
 - regular expressions, PWB, 92, 95
 - user-defined macros, NMAKE, 669
- _ (underscore)
 - macros, NMAKE, 668
 - regular expressions, PWB, 93
 - symbol format, CodeView, 409
- { } (braces)
 - context operator, CodeView, 405–407, 421–422
 - key box, PWB, 120
 - RTF formatting codes, 726
 - specifying search path, NMAKE, 660
- ~ (tilde), menu command, PWB, 126

A

- A command, CodeView, 422, 424–426, 436–437
- \a formatting attribute, HELPMMAKE, 718, 721
- /A option, NMAKE, 648
- About command
 - CodeView, 374–375
 - PWB, 78
- /AC option
 - CL, described, 488–490
 - HELMMAKE, 712
- Access control, CodeView, 409
- Activating, windows, PWB, 262–263
- Actual parameters. *See* Arguments
- Adapters defined, 803
- Add command, LIB, 703–704
- Add Watch command, CodeView, 364, 423
- Add Watch dialog box, CodeView, 364
- Add Watch Expression command, CodeView, 460
- Adding
 - breakpoints, CodeView, 367
 - commands, PWB Run menu, 125–127
 - custom sections, PWB, 58–59

- Adding (*continued*)
 - files, PWB, 44, 47–48, 50
 - Program Item, PWB, 66
 - watch expressions, CodeView, 364, 460
- Address ranges
 - CodeView expressions, 402–403, 420–421
 - defined, 803
- Addresses
 - CodeView expressions, 401–402, 419–420
 - defined, 803
 - variables, debugging assembly code, 414
- AFLAGS options macro, NMAKE, 676
- /AH option, CL, 488–490
- AH register, CodeView syntax, 419, 450
- /AL option, CL, 488–490
- AL register, CodeView syntax, 419, 450
- Aliasing, optimization, CL options, 531–532, 536
- Aligning tabs, PWB, 297
- All Files command, PWB, 72
- All Windows command, PWB, 77
- alloc_text pragma, 599, 626
- Allocated functions, overlaid DOS programs, 599
- Allocating
 - memory, MOVE, 602–603
 - registers, CL options, 533
 - space, LINK, 577–578
- Alphabetic characters, predefined expression syntax, 778, 780, 785
- Alphanumeric characters, predefined expression syntax, 778, 780, 785
- Alternation, regular expression syntax, 778, 780–781, 787
- ALTGR key, enabling, 276
- /AM option, CL, 488–490
- Ampersand (&)
 - C address operator, 414–415
 - CodeView, 405
 - operations line, extending, 702
- Angle brackets (< >)
 - command buttons, PWB, 80, 82
 - inline files, NMAKE, 664–665
- Animate command, CodeView, 362–363, 369, 422, 432–433, 453
- ANNUITY1.C sample program, 29
- Anonymous allocation defined, 803
- ANSI
 - additional C features, 550–552
 - defined, 803

- ANSI escape sequence, CodeView expressions, 408
- API defined, 803
- Appending DOS device names, CL option, 497–498
- Application programming interface defined, 803
- Applications
 - inserting text, module-definition files, 613–614
 - specifying
 - module-definition files, 611–612
 - protected mode, 617
 - real mode, 617
- APPOADER statement, module-definition files, 609, 615
- Arg function, PWB, 151, 156–157
 - executing, 106–108
 - getting Help, 762
 - replacing text, 94
- argc defined, 803
- Arguments
 - CodeView
 - entering, 352
 - format, 352–353
 - setting, 363–364
 - command line, 337
 - defined, 803
 - functions, PWB, 156–157
 - module statement syntax, 610
 - numeric in LINK, 576
- argv defined, 803
- Arrange command
 - CodeView, 373–374
 - PWB, 77, 145
- Arrangewindow function, PWB, 151, 157
- Arrays
 - debugging assembly language, 415
 - expanding and contracting, CodeView, 367–368, 478–479
- AS command macro, NMAKE, 675
- /AS option, CL, 488–490
- ASCII
 - HELPMAKE format, 728
 - memory format, CodeView, 356–357
- ASCII characters defined, 804
- Askexit switch, PWB, 263, 267
- Askrtm switch, PWB, 263, 267
- .ASM files defined, 804
- Assemble command, CodeView, 422, 424–426, 436–437
- Assembling, 8086 instruction mnemonics, 424–426
- Assembly files, creating listing, CL, 501–505
- Assembly language
 - debugging, 412–415
 - specifying, CL option, 542
 - translating source code, CL, 501–505
- Assembly mode defined, 804
- Assign function, PWB
 - described, 151, 157–158
 - executing, 108
 - key assignment, changing, 121
 - switch settings, changing, 124
- Asterisk (*)
 - Comment command, CodeView, 423, 471
 - Copy command, LIB, 705
 - deleting watch expressions, CodeView, 461
 - filename macros, NMAKE, 672–673
 - hyperlink, Microsoft Advisor, 764
 - match character, regular expression syntax, 779
 - regular expressions, PWB, 93
 - SBRPACK syntax, 740
 - wildcard operator
 - HELPMAKE syntax, 711
 - NMAKE, 653–654
 - UNDEL syntax, 749
- /AT option, CL, 488–490
- At sign (@)
 - BSCMAKE syntax, 738
 - command files, NMAKE, 651
 - command modifier, NMAKE, 661
 - filename macros, NMAKE, 672–673
 - LINK syntax, 573
 - local contexts, HELPMAKE, 720
 - naming registers, CodeView, 401, 419
 - Redraw command, CodeView, 424, 479
- Attributes
 - formatting in HELPMAKE, 718, 721
 - segment, defining with module statements, 618–619
- Auto option, Language command, CodeView, 370
- AUTOEXEC.BAT
 - HELPPFILES environment variable, 771
 - PWB configuration, 137
- Autoload switch, PWB, 263, 268
- Automatic indentation, PWBC switches, 312
- Autosave switch, PWB, 122, 263, 269
- Autostart entry, TOOLS.INI file, CodeView, 330
- Autostart macro, PWB, 224
- AUX, CL options, appending to, 497–498
- Available memory
 - defined, 804
 - overlaid DOS programs, 597
- AX register, CodeView syntax, 419, 450

B

- \b formatting attribute, HELPMMAKE, 719, 721, 727
- /B option
 - CodeView, 338–339
 - LINK, 576
 - NMAKE, 648
- Backing up files, PWB, 95, 303, 747–750
- Backslash (\)
 - escape, regular expression syntax, 780, 786
 - HELMMAKE syntax, 720–721
 - line continuation character
 - NMAKE, 655, 660, 669
 - PWB, 115–117, 136
 - LINK syntax, 567
 - match character, regular expression syntax, 779
 - regular expressions, PWB, 96
 - Screen Exchange command, CodeView, 424, 479
- Backtab function, PWB, 127–128, 151, 159
- Backup files, 95, 303, 747–750
- Backup switch, PWB, 263, 269–270
- .BAK files defined, 804
- .BAS files defined, 804
- Base names
 - Curfilenam predefined macro, PWB, 225
 - defined, 804
 - Shortnames switch, PWB, 296
- Base operator (:<), CodeView precedence, 406
- __based keyword
 - /A options, CL, 489
 - allocated functions, 599
 - enabling, CL options, 550
 - ordering functions, module-definition files, 626
 - segmenting data, CL option, 530
- Basic Compiler, NMAKE macros, 675–676
- .BAT files defined, 804
- Batch files
 - backing up C files, PWB, 95
 - building browser database, PWB, 104–105
 - defined, 804
 - executing CL, 490
- /batch option, CL, 490
- /BATCH option, LINK, 576
- BC command macro, NMAKE, 675
- BC command, CodeView, 422, 426–427
- BD command, CodeView, 422, 427–428
- BE command, CodeView, 422, 428–429
- Beep switch, PWB, 263, 270
- Begfile function, PWB, 151, 159
- Begline function, PWB, 151, 159–160
- BFLAGS options macro, NMAKE, 676
- BH register, CodeView syntax, 419, 450
- Binary files defined, 804
- Binary operators defined, 804
- BIOS defined, 691, 804
- Bit rate, remote debugging, 396
- BL command, CodeView, 422, 429
- BL register, CodeView, 419, 450
- Black, color value, 273
- Blank lines, command lines, NMAKE, 660
- Blue, color value, 273
- /Bm option, CL, 491
- .BMP files defined, 804
- Bold text, HELPMMAKE formatting
 - QuickHelp, 721
 - rich text format, 726
- Boolean switches, PWB, 122, 266
- Box Mode command, PWB, 73, 143
- Boxes, command execution, PWB, 80
- BP command, CodeView, 384–385, 422, 429–432
 - display radix, 444–445
 - line numbers, 400
- BP register, CodeView syntax, 419, 450
- Braces ({ })
 - context operator, CodeView, 405–407, 421–422
 - key box, PWB, 120
 - RTF formatting codes, 726
 - specifying search path, NMAKE, 660
- Brackets ([])
 - balancing, PWB, 192–193
 - call tree, PWB, 100
 - character class, regular expression syntax, 778, 780–781, 786
 - match character, regular expression syntax, 779
- Breakpoint Clear command, CodeView, 422, 426–427
- Breakpoint Disable command, CodeView, 422, 427–428
- Breakpoint Enable command, CodeView, 422, 428–429
- Breakpoint List command, CodeView, 422, 429
- Breakpoint Set command, CodeView, 384–385, 422, 429–432, 444–445
- Breakpoints
 - CodeView
 - clearing, 426–427
 - disabling, 427–428
 - enabling, 428–429
 - listing, 429
 - saving, 344
 - setting, 326, 365–366, 384–385, 429–432
 - defined, 804

Bright Back check box, PWB, customizing colors, 125

Bright Fore check box, PWB, customizing colors, 125

Browcase switch, PWB, 309

Browdbase switch, 309–310

Brown, color value, 273

Browse menu, PWB, 76, 145, 200

BROWSE option, BSCMAKE, 732

Browse Options command, PWB, 75

Browser Database Maintenance Utility. *See* BSCMAKE

Browser database, PWB

- building
 - BSCMAKE, 731–734
 - combined, 106
 - described, 58, 101
 - non-PWB projects, 104–106
- creating, 97–98
- estimating file size, 103–104
- finding symbol definitions, 101–103
- makefiles, 61
- menu commands, 76
- overview, 731
- specifying switches, 310

Browser Information Compactor. *See* SBRPACK

Browser information files, PWB

- browser database, 97–98
- building browser database, non-PWB, 105
- estimating size, 103–104

Browser Output command, PWB, 77, 102

Browser, source. *See* Source browser

.BSC files

- defined, 804
- PWB, 97, 103–104

BSCMAKE

- building a database, 731–734
- command line, 735–738
- options, 732, 736–737
- overview, 731
- response files, 738
- syntax, 735
- system requirements, 734

BSCMAKE command

- non-PWB, 105
- PWB, 97–98

BSCMAKE.EXE, 734

BSCMAKEEV.EXE, 734

Buffers

- CodeView command window, 353
- decompression, specifying size, 332
- defined, 804

Bugs. *See* Debugging

Build command, PWB, 74, 144

Build Results command, PWB, 77

Build Results window, PWB

- clearing, 162
- described, 261
- Nextmsg function, 188–189
- retaining results, 289

Build switch, PWB, 263

Build:message switch, tagged expressions, 784

Building

- browser database, PWB
 - BSCMAKE, 731–734
 - combined, 106
 - creating, 97–98
 - described, 58, 101
 - non-PWB projects, 104–106
- canceling, `_pwbcancelbuild` macro, 229–230
- customized PWB projects, 58–61
- described, PWB, 56–58
- menu commands, PWB, 74
- multimodule programs, PWB, 45–46
- programs, NMAKE, 646
- `_pwbbuild` macro, 229
- targets
 - NMAKE, 648
 - PWB, 163–164

Buttons

- command execution, PWB, 80–82
- hyperlinks
 - index screens, 764
 - navigating with, 759–761

BX register, CodeView syntax, 419, 450

BY operator, CodeView, 405, 414–415

Bytes, displaying, CodeView, 356–357

C

`:c` command, HELPMMAKE, 722

C Compiler Options command, PWB, 75

C Compiler, NMAKE macros, 675–676

C expression evaluator

- choosing, 403–404
- defined, 399
- overview, 399
- using, 404–407

- .C files defined, 804
- /C option
 - CL, 325, 491
 - CodeView, 338, 340
 - HELPMMAKE, 712
 - NMAKE, 648
- /c option, CL
 - described, 491
 - option interactions, 496–497
- C++ Compiler Options command, PWB, 75
- C++ Compiler, NMAKE command macro, 676
- C++ expression evaluator
 - choosing, 403–404
 - overview, 399
 - using, 404–415
- Caches, overlaid DOS programs, 598, 602–603
- Call gates defined, 804
- Call Tree command, PWB, 76, 145
- Call trees, PWB, 86, 99–101
- Calling conventions, CL options, 516–520
- Calling functions, CodeView expressions, 405
- Calls menu, CodeView, 372–373
- Cancel function, PWB, 151, 160
- Canceling
 - background search, `_pwbcancelsearch` macro, 230–231
 - builds, `_pwbcancelbuild` macro, 229–230
 - print operations, `_pwbcancelprint` macro, 230
- Cancelsearch function, PWB, 151, 161
- Caret (^)
 - character ranges, regular expression syntax, 779
 - line beginning, regular expression syntax, 778–781, 786
 - literal characters, NMAKE, 653
 - regular expressions, PWB, 92, 95
 - user-defined macros, NMAKE, 669
- Cascade command, PWB
 - described, 77
 - predefined macros, 145
- Cascading window arrangements, `_pwbcascade` macro, 231
- Case sensitivity
 - browser database, 309
 - CodeView
 - commands, 368, 371, 417
 - expression evaluators, 405
 - options, 445–447
 - search option, 361
 - defined, 804
 - global searches, in Microsoft Advisor, 766
 - IMPLIB, 747
- Case sensitivity (*continued*)
 - LIB options, 701
 - PWB
 - options, 141–142
 - search functions, 270–271
- Case switch, PWB, 263, 270–271
- .category command, HELPMMAKE, 722
- CC command macro, NMAKE, 675
- `__cdecl` keyword
 - calling conventions, CL options, 516–518
 - enabling, CL options, 550
 - symbol format, CodeView, 409
- Cdelete function, PWB, 151, 161
- CFLAGS options macro, NMAKE, 676
- CGA defined, 805
- CGA displays, suppressing snow, CodeView
 - option, 341
- CH register, CodeView syntax, 419, 450
- Character range, regular expression syntax, 779
- Character strings defined, 805
- Characters
 - ASCII, defined, 804
 - case distinction, LINK option, 585
 - changing type, CL option, 526–527
 - classes
 - PWB, 92
 - regular expression syntax, 778, 780–781, 786–787
 - controlling, HELPMMAKE, 712
 - deleting, PWB, 161, 167–168, 209–210
 - inserting, PWB, 173–174
 - makefiles, NMAKE, 653
 - matching, regular expression syntax, 779
 - predefined expression syntax, 778, 785
 - searching, 91
 - special, NMAKE, 668
- Check box, PWB, 82
- `check_pointer` pragma, /ZR option, CL, 556–557
- `check_stack` pragma, removing stack probes, 518–520
- Child process defined, 805
- CL
 - batch files, executing, 490
 - calling conventions, 516–518, 520
 - command line, 485–486
 - compiling without linking, 491
 - constants, defining, 491–493
 - data allocation, 523–524
 - data threshold, setting, 522
 - debugging, 553
 - DOS device names, 497–498
 - entry/exit codes, optimizing, 515, 522–523

CL (*continued*)

- environment variables
 - NO87, 513
 - specifying options, 557–559
- filename extensions
 - processing, 486
 - specifying, 496
- files
 - assembly, listing, 501–503, 505
 - browser, generating, 507–508
 - executable, creating, 486–487
 - machine-code, listing, 502–503, 505
 - map, creating, 505–507
 - optimizing size, 538
 - renaming, 498–499
 - source, listing, 501
- floating-point math operations, 508–513
- function-level linking, 524
- function prototypes listing, 552–553
- intrinsic functions, generating, 534–535
- language specification, 542
- linker-control options, 527
- macros, defining, 491–493
- memory capacity, increasing, 491
- naming conventions, 518
 - Pascal, 552
 - segments, 528–529
- optimizing
 - aliasing, 531–532
 - execution time, 539
 - exit sequence, 538
 - file size, 538
 - float consistency, 537–538
 - frame sorting, 539
 - inline expansion control, 532
 - intrinsic function generation, 534–535
 - loops, 535–537
 - maximum, 539
 - p-code quoting, 533
 - post-code generation, 536–537
 - register allocation, 533
 - subexpressions, 533
 - turning off, 533
- options, 323–324, 488–557
- output files, setting alternates, 495
- p-code, removing entry points, 520–521, 533
- paths, specifying, 496
- preprocessing
 - copying output, 493–494
 - output file, creating, 540
 - preserving comments, 491

CL (*continued*)

- processor-specific instructions, 514–515
- segments, 529–530
- specifying entry tables, 521
- stacks
 - checking, 518–520
 - size, setting, 494
- syntax, 488, 557
- warning level, setting, 544–545
- CL register, CodeView syntax, 419, 450
- Class Hierarchy command, PWB
 - described, 76
 - function, 145
- CLASS keyword, module-definition files, 619–620
- Class Tree command, PWB, 76, 145
- Classes
 - characters, regular expression syntax, 778, 780–781, 786–787
 - CodeView accessibility, 409–411
- Clearing breakpoints in CodeView, 367, 426–427
- Clearmsg function, PWB, 151, 162
- Clearsearch function, PWB, 151, 162
- Click defined, 805
- Clipboard defined, 805
- Clipboard Results command, PWB, 77
- Close All command, PWB
 - described, 77
 - predefined macros, 145
- Close command
 - CodeView, 373–374
 - PWB, 72, 77, 142–145
- Close Project command, PWB, 74
- Closefile function, PWB, 151, 163
- Closing
 - files, PWB, 72, 233
 - Help files
 - PWB, 213–214
 - QuickHelp, 769
 - menus, PWB, 79
 - projects, PWB, 234
 - windows, PWB, 220, 232–233
- CLRFILE.CV4, CodeView, 343–344, 360, 370–371
- CLRFILE.CVW, CodeView, 343–344, 360, 370–371
- !CMDSWITCHES preprocessing directive,
 - NMAKE, 688–689
- /CO option, LINK, 324–325, 577, 743
- .COB files defined, 805
- COBFLAGS options macro, NMAKE, 676
- COBOL command macro, NMAKE, 675
- COBOL Compiler, NMAKE macros, 675–676

- Code
 - inline, debugging, 322
 - inserting, HELPMAKE rich text format, 726
 - invariant, removing, 535–536
 - memory model, 489–490
 - p-code
 - native entry points, 520–521
 - optimizing, 538
 - quoting, 533
 - specifying, 521
 - tracing to native code, 391–392
 - searching, PWB, 85
 - source, displaying, 350, 457–460
- Code links, HELPMAKE, 718
- Code pointers, memory model codes, CL options, 489–490
- Code segments
 - defining attributes, module-definition files, 618, 620
 - memory models, CL options, 488–490
 - naming, CL option, 528–530
 - overlaid DOS programs, 598, 603–605
- CODE statement, module-definition files, 609, 618, 620
- Code symbols defined, 805
- CodeView
 - access control, 409
 - animating, 432–433
 - arguments
 - entering, 352
 - format, 352
 - setting, 363–364
 - breakpoints
 - clearing, 426–427
 - disabling, 427–428
 - editing, 367
 - enabling, 428–429
 - listing, 429
 - setting, 326, 365–366, 384–385, 429–432
 - case sensitivity
 - commands, 371
 - expression evaluators, 405
 - options, 445–447
 - CL options, 553
 - command line, 336–343
 - commands
 - copying text, 353
 - described, 460
 - entering, 352
 - executing, 340, 433–434
 - compacting files with CVPACK, 743–744
 - CodeView (*continued*)
 - compatibility, MOVE, 603
 - configuring, 329–330
 - contracting elements, 367–368, 478–479
 - CURRENT.STS, PWB, 138
 - CVW
 - commands, 382–385
 - compared to CV, 377–382
 - multiple applications, 380–382
 - multiple instances, 379–380
 - running, 378–379
 - techniques, 386–388
 - debugging
 - p-code, 389–393
 - assembly language, 412–415
 - PWB programs, 28–34
 - displays
 - black-and-white, 339
 - line-display mode, 339
 - overview, 345–347
 - redrawing, 479
 - screen exchange, 341, 371, 445–447, 479
 - specifying, 338–339
 - suppressing snow, 341
 - dynamic-link libraries, loading, 328, 342, 363–364
 - editing, 360
 - execution
 - controlling, 386
 - speed of, 453
 - terminating, 387–388
 - expanding elements, 367–368, 478–479
 - expression evaluators
 - choosing, 403–404, 454–455
 - defined, 399
 - listing, 370
 - numbers, 407–408
 - operators, 405–407
 - string literals, 408
 - symbol format, 409
 - expressions
 - See also* Expressions
 - address ranges, 403, 420–421
 - addresses, 401–402, 420
 - C++, 409–415
 - line numbers, 400, 418
 - overview, 399
 - registers, 401
 - flags, changing values, 450–452
 - functions
 - listing, 435–436
 - tracing, 452–453

CodeView (*continued*)

Help

See also Microsoft Advisor
getting, 756–765
structure, 755

Help menu, 757

identifying bugs, 325–326

installing, 327–329

interrupt trapping, 341–342

interrupting execution, 387

LINK option, 577

loading symbolic information, 342

locating bugs, 326

memory

comparing, 437–438

dumping, 438–439

entering data, 440–441

filling, 441–442

format, 356–357

management of, 336

moving blocks of, 442–443

searching, 443–444

viewing, 455–457

menus

Calls menu, 372–373

Data menu, 364–368

Edit menu, 360

File menu, 358–360

Help menu, 374–375

Options menu, 368–372

Run menu, 362–364

Search menu, 361–362

Windows menu, 373–374

module statement keywords, 621

modules

configuring, 363–364

listing, 463

mouse, disabling, 342

options, 338–344, 396, 445–447

preparing programs, 321–325

printing, 359

PWB menu commands, 74

quitting, 360

radix, 444–445

registers, changing values, 450–452

remote debugging

overview, 393

requirements, 393–395

starting a session, 397–398

syntax, 396

restarting, 362, 436–437

CodeView (*continued*)

searching, 361–362

shell escape, 468–469

slow motion execution, 363, 369

source code, displaying, 457–460

source files

loading, 359

opening, 358

state file

overview, 344

toggling status, 343

syntax, 336–343

CVW commands, 382–385

expressions, 400–403, 417–421

regular expressions, 779

TOOLS.INI file entries, 330–336

TOOLS.INI file entries, 330–336

trace speed, 453

variables

listing, 369

local, 354

program, 351

viewing output, 374

watch expressions

adding, 364, 460

deleting, 365, 461

listing, 465–466

setting, 326–327

windows

8087 window, 355–356

Command window, 351–354, 417

described, 350

Help window, 357–358

Local window, 354

Memory windows, 356–357

navigation, 349

opening, 373

overview, 347–349

Register window, 354–355

Source windows, 350

Watch window, 350–351

/CODEVIEW option, LINK, 577

CodeView Options command, PWB, 75

Colon (:)

appending device names, CL, 497–498

CodeView operator, 405

Delay command, CodeView, 423, 475

dependency, NMAKE, 656–657

dot directives, NMAKE, 687

HELPMAKE commands, 722

inference rules, NMAKE, 681

- Colon (:) (*continued*)
 - macro substitution, NMAKE, 677
 - module statement syntax, 610
 - target separator, NMAKE, 655
- Color entry, TOOLS.INI file, CodeView, 330–331
- Color graphics adapter defined, 805
- Color switch, PWB, 263, 271–273
- Colors
 - customizing, PWB, 124–125
 - setting, CodeView, 370–371
 - specifying, PWB, 271–273, 313
 - values, 272
- Colors command
 - CodeView, 368, 370–371
 - PWB, 75
- Colors dialog box, CodeView, 370–371
- .COM files defined, 805
- Combining libraries, 704
- Combo box, PWB, 81–82
- Comma (,.)
 - argument separator, CodeView, 352–353
 - CodeView operator, 405
 - field separation, LIB, 699
 - LINK syntax, 565
 - with context operator, CodeView, 421–422
- !command
 - CodeView, 452
 - HELPMMAKE, 719
- Command buffer, using CodeView, 353
- Command button, PWB, 82
- Command command, CodeView, 373–374
- .command command, HELPMMAKE, 722
- Command files
 - defined, 805
 - NMAKE, 651
- Command lines
 - BSCMAKE, 735–738
 - CL, 485–486
 - CodeView, 336–343
 - CVPACK, 744
 - EXEHDR, 629–631
 - EXP, 750
 - IMPLIB, 746–747
 - LIB, 698
 - LINK, 564–572
 - MOVE, 604–605
 - NMAKE
 - command file, 651
 - commands, 660
 - described, 647
 - Command lines (*continued*)
 - NMAKE (*continued*)
 - macros, defining, 669–670
 - suppressing, 688
 - PWB, 141–142
 - RM, 748–749
 - SBRPACK, 740–741
 - UNDEL, 749
 - Command modifiers, NMAKE, 661–662
 - Command shell, DOS Shell command, CodeView, 359
 - Command window, CodeView
 - command format, 417
 - function, 351–353
 - opening, 374
 - overview, 347
 - COMMAND.COM file handles, 808
 - Command-line arguments, NMAKE, 651
 - Commands
 - CodeView
 - copying text for, 353
 - CVW, 382–385
 - Data menu, 364–368
 - described, 424–479
 - Edit menu, 360–362
 - entering, 352
 - executing, 340, 433–434
 - File menu, 358–360
 - for Windows applications, 377
 - format, 352–353
 - Help menu, 374
 - Options menu, 368–372
 - Run menu, 362–363
 - Windows menu, 373
 - defined, 805
 - HELPMMAKE, 713–714, 722–724
 - LIB, 702–705
 - NMAKE
 - displaying, 649
 - exit codes, 662–664
 - inline files, 664–667
 - macros, 675
 - modifiers, 661–662
 - predefined inference rules, 684–685
 - suppressing display, 650
 - syntax, 660–661
 - PWB
 - choosing, 78–79
 - cursor movement, 154–155
 - Edit menu, 73
 - executing, 78–82, 142, 170, 219

Commands (*continued*)

- PWB (*continued*)
 - File menu, 72
 - Help menu, 78
 - Options menu, 75
 - predefined, 142–146
 - Project menu, 74
 - Run menu, 74, 125, 127
 - Search menu, 73
 - Window menu, 77
- QuickHelp, 770
- Comment command, CodeView, 423, 471
- .comment command, HELPMMAKE, 722
- Comment line, custom builds in PWB, 59
- Comments
 - makefiles, 654
 - preserving, CL, 491
 - TOOLS.INI file, 136, 329–330
- Common expressions, optimizing, CL option, 533
- Compact memory model defined, 805
- Compacting files, CVPACK, 743–744
- Compatibility, floating-point math operations, 512
- Compile command, PWB, 144
- Compile File command, PWB, 74
- Compile function, PWB, 150, 163–164
- Compiler options
 - changing, PWB, 52–56
 - debugging considerations, 323–324
 - listing, 525
- Compilers
 - changing options, PWB, 52–56
 - increasing capacity, CL, 491
 - menu commands, PWB, 75
 - optimizing, CL options, 531–539
 - options, 56
- Compiling
 - debugging considerations, 323
 - defined, 805
 - files, PWB, 234
 - for debugging, CL option, 553
 - overlays, 599–600
 - without linking, CL, 491
- Compressing
 - help database, 711–712
 - keywords, HELPMMAKE option, 712–713
- CON, CL options, appending to, 497–498
- Concatenating help files, 772
- Conditional breakpoints defined, 805
- Conditional operator (?), CodeView, 405
- CONFIG.SYS
 - editing, PWB, 66
 - memory management, CodeView, 336
 - PWB configuration, 137
- Configuring CodeView
 - modules, 363–364
 - TOOLS.INI, 329–330
- Consistency checks, LIB, 700
- Constant expressions defined, 805
- Constants
 - defined, 805
 - defining, CL, 492–493
- Constructors, using C++ expressions, 410–411
- Contents command
 - CodeView, 374–375
 - PWB, 78, 146, 757
- .context command, HELPMMAKE, 716–717, 720, 722, 726
- Context operator ({ }), CodeView, 405–407, 421–422
- Context prefixes, HELPMMAKE, 729
- contextstring command, HELPMMAKE, 718–719
- Contracting, elements in CodeView, 367–368, 478–479
- Control characters, specifying, HELPMMAKE, 712
- Control library, selecting, CL options, 527
- Conventional memory
 - browser database, 733–734, 737
 - defined, 805
- Conventions
 - CL options
 - calling, 516, 520
 - naming, 518, 529, 552
 - document, xxv
- Conversion functions, using C++ expressions, 410–411
- Coprocessors
 - defined, 803
 - displaying registers, CodeView, 355–356
 - floating-point math, 509–511, 513
- Copy command
 - CodeView, 360
 - LIB, 705
 - PWB, 73, 143
 - MS-DOS
 - concatenating help databases, 710–711
 - concatenating help files, 772
- Copy function, PWB, 151, 164–165
- Copying
 - files, PWB, 95
 - object modules, 705

- Copying (*continued*)
 - preprocessor output, CL, 493–494
 - text
 - CodeView commands, 353
 - Microsoft Advisor, 761
 - QuickHelp, 771
 - Copyright message, suppressing. *See* /NOLOGO
 - option
 - COUNT sample program, PWB, 41–63, 97–103
 - /CP option, LINK, 577–578
 - /CPARM option, LINK, 577–578
 - /CPARMAXALLOC option, LINK, 577–578
 - CPP command macro, NMAKE, 676
 - .CPP files defined, 805
 - CPPFLAGS options macro, NMAKE, 676
 - CPU defined, 805
 - CRC command macro, NMAKE, 676
 - Creating
 - backup files, 747–750
 - browser database, PWB, 97–98
 - call tree, PWB, 99–101
 - executable files, CL, 486–487
 - import libraries, IMPLIB, 745–746
 - inline files, NAME, 665
 - library files, 700, 703
 - map files
 - CL, 505–507
 - LINK, 582–583
 - module-definition files, 600–601
 - overlaid programs
 - LINK, 598–601
 - module-definition files, 619–620
 - MOVE, 598–601
 - packaged functions, CL options, 524
 - preprocessor-output files, 540
 - projects, PWB, 42
 - pseudofiles, in PWB, 187–188, 245
 - segmented files, LINK, 564
 - Cross-reference listing, LIB, 705–706
 - CS command, CodeView, 422
 - CS register, CodeView syntax, 419, 450
 - CS:IP
 - defined, 805
 - saving, CodeView, 344
 - C_Softer switch, PWB, 312
 - C_suffices switch, PWB, 312–313
 - Curdate function, PWB, 151, 165
 - Curday function, PWB, 151, 165
 - Curfile predefined macro, PWB, 222, 224–225
 - Curfileext predefined macro, PWB, 222, 225–226
 - Curfilenam predefined macro, PWB, 222, 225–226
 - Current date, PWB, 165
 - Current Location command, CodeView, 423, 471
 - CURRENT.STS
 - CodeView
 - overview, 344
 - saving, 360
 - toggleing status of, 343
 - PWB, 138
 - Cursor
 - defined, 805
 - PWB commands, 154–155
 - shape of, in PWB, 273
 - Cursormode switch, PWB, 263, 273
 - Curtime function, PWB, 151, 166
 - Customize Project Template command, PWB, 75
 - Customize Run Menu command, PWB, 74
 - Cut command, PWB, 73
 - predefined macros, 143
 - CV. *See* CodeView
 - Cvdlpath entry, TOOLS.INI file, CodeView, 330–331
 - CVPACK
 - command line, 744
 - exit codes, 745
 - Help, 745
 - options, 744
 - overview, 743–744
 - syntax, 744
 - CVW
 - See also* CodeView
 - commands, 382–385
 - compared to CV, 377
 - debugging techniques, 386–388
 - multiple applications, debugging, 379–382
 - running, 378–379
 - CX register, CodeView syntax, 419, 450
 - CXX command macro, NMAKE, 676
 - .CXX files defined, 806
 - CXXFLAGS options macro, NMAKE, 676
 - Cyan, color value, 272
- ## D
- /D option
 - CL, 429–432, 492–493
 - HELPMMAKE, 714
 - NMAKE, 648
 - PWB, 141
 - d. context prefix, HELPMMAKE, 729
 - /DA option, PWB, 141
 - Dark gray, color value, 273

- Dash (–)
 - character classes, PWB, 92
 - character ranges, regular expression syntax, 779
 - CL syntax, 488
 - command line, NMAKE, 647
 - command modifier, NMAKE, 661
 - Delete command, LIB, 700, 704
 - HELPMMAKE options, 711
- .DAT files defined, 806
- Data
 - dumping, CodeView, 438–439
 - entering, CodeView, 440–441
 - exporting, module-definition files, 623
 - importing, module-definition files, 624–625
 - moving blocks, CodeView, 442–443
- Data allocation, CL options, 523–524
- Data menu, CodeView, 364–368
- Data pointers, memory model codes, CL options, 489–490
- Data segment register, LINK, 579
- Data segments
 - defining attributes, module-definition files, 618–620
 - loading data, LINK, 579
 - memory models, CL options, 488–490
 - naming, CL option, 528–530
 - overlaid DOS programs, 598, 603–605
 - packing, LINK, 588–589
- DATA statement, module-definition files, 609, 618–620
- Data symbol defined, 806
- Data threshold, setting, CL option, 522
- Database
 - browser. *See* Browser database
 - help
 - context prefixes, 729
 - creating, 711–712
 - decoding, 713–714
 - overview, 710–711
- Date, current in PWB, 165
- .DBG files defined, 806
- Dblclick switch, PWB, 263, 274
- Debug command, PWB, 74
- Debug options, finding symbols, PWB, 101
- Debugger defined, 806
- Debugging
 - See also* CodeView
 - assembly language, 412–415
 - CL options, 553
 - CodeView options, 338–344
 - Debugging (*continued*)
 - CVW
 - commands, 382–385
 - compared to CV, 377–382
 - multiple applications, 380–382
 - multiple instances, 378–380
 - techniques, 386–388
 - identifying bugs, 326
 - locating bugs, 326
 - makefiles, NMAKE, 648–649
 - p-code, 389–393
 - programs
 - preparing, 321–325
 - PWB, 28–34
 - remote
 - bit rate, 396
 - options, 396
 - overview, 393
 - requirements, 393–395
 - starting a session, 397–398
 - syntax, 396
 - specifying libraries, LINK, 566
 - syntax, TOOLS.INI file entries, 330–336
 - watch expressions, 326–327
- Debugging information
 - See also* Symbolic Debugging Information Compactor. *See* CVPACK
- Decoding, HELPMMAKE options, 713–714
- Decompressing
 - help database, 714
 - help files, 332
- Decorated names
 - debugging considerations, 324
 - overview, 789–790
 - specifying, 790–791
- .DEF files. *See* Module-definition files
- Default keys, 142–150
- Default libraries
 - defined, 806
 - LINK, 568, 583–584
- Default values, LINK, 573
- deffile field, LINK, 570–572
- Define Mark command, PWB, 73
- DEFINED operator, NMAKE, 690–691
- Defining
 - constants, CL, 492–493
 - macros, CL, 492–493
- Deflang switch, PWB, 263, 274
- Defwinstyle switch, PWB, 263, 275
- Delay command, CodeView, 423, 475

- Delete command
 - LIB, 700, 704
 - PWB
 - described, 73
 - predefined macros, 143
- Delete function, PWB, 151, 166
- Delete Watch command, CodeView, 364–365, 423
- Delete Watch dialog box, CodeView, 365
- Delete Watch Expressions command, CodeView, 461
- DELETED directory, backup utilities, 747–748
- Deleting
 - breakpoints, CodeView, 367
 - characters, PWB, 161, 167–168, 209–210
 - files
 - during debugging session, 359
 - EXP, 750
 - PWB, 48
 - RM, 748–749
 - lines, PWB, 176–177
 - marks, PWB, 178–179
 - object modules, 704
 - text, PWB, 166, 232
 - watch expressions, CodeView, 365, 461
- Delimiters
 - help (>>), 726, 728
 - regular expressions, PWB, 91
- Dependencies
 - executing, NMAKE, 648
 - PWB programs, 45, 48
- Dependency lines
 - defined, 655
 - dependents, NMAKE, 659–660
- Dependency tree
 - building, NMAKE, 655
 - updating dependents, NMAKE, 659
- Dependent files
 - defined, 646
 - dependency lines, NMAKE, 655
- Dependents
 - dependency lines, NMAKE, 659–660
 - inferred, NMAKE, 659, 685–686
- Dereference Global Handle command, CodeView, 463–464
- Dereference Local Handle command, CodeView, 466–467
- Dereferencing memory handles, CodeView, 386
- Description blocks
 - dependency lines, 655
 - NMAKE, 656–660
- DESCRIPTION statement, module-definition files, 609, 613–614
- Destructors, using C++ expressions, 410–411
- DGROUP defined, 806
- DH register, CodeView syntax, 419, 450
- DI register, CodeView syntax, 419, 450
- Dialog boxes
 - CodeView, getting Help, 761
 - defined, 806
 - HELPMAKE context prefix, 729
 - PWB
 - default key assignments, 150
 - displaying, 284–285, 288, 294
 - function, 80–82
 - getting Help, 761
 - Help, 758, 764
- Dictionaries, extended, suppressing, in LIB, 701
- Digits, predefined expression syntax, 778, 780, 785
- DIR command, PWB, 94–95
- Directives
 - dot, NMAKE, 687–688
 - preprocessing, NMAKE, 688, 690–692
- Directories
 - ignoring, CL option, 545
 - listing C files, PWB, 94
 - searching, CL options, 525–526
- Disable Mouse in CodeView option, 342
- Disabling
 - breakpoints, CodeView, 367, 427–428
 - mouse, CodeView option, 342
- Disassembling defined, 806, 818
- DISCARDABLE keyword, module-definition files, 620
- Display
 - CodeView
 - black-and-white display, 339
 - line-display mode, 339
 - arranging, 327
 - memory format, 356–357
 - overview, 345–347
 - redrawing, 479
 - screen exchange, 341, 343, 371, 445–447, 479
 - specifying, 338–339
 - suppressing snow, 341
 - PWB
 - height, 283
 - screen, 67–68
 - specifying color, 271–273
 - width, 306
- Display Expression command, CodeView, 424, 477–478

- Display modules, listing, CodeView, 383
 - DL register, CodeView syntax, 419, 450
 - .DLL files
 - See also* Dynamic-link library
 - defined, 806
 - Dlllibs switch, PWB, 310–311
 - .DOC files defined, 806
 - Dollar sign (\$)
 - end of line, regular expression syntax, 778, 780–781, 786
 - environment variables, NMAKE, 679
 - filename macros, NMAKE, 672–673
 - literal characters, NMAKE, 653
 - match line end, regular expression syntax, 779
 - reference to tagged expressions, regular expression syntax, 780, 787
 - regular expressions, PWB, 93
 - user-defined macros, NMAKE, 669
 - DOS
 - device names, appending, 497–498
 - Help, getting, 768–769
 - managing memory, browser database, 733–734
 - overlays, LINK, 566
 - session defined, 806
 - DOS applications
 - defined, 806
 - module-definition files, 607–608
 - overlaid
 - LINK, 597–601
 - MOVE, 597–603
 - DOS executable files, EXEHDR output, 632–633
 - DOS Extender defined, 806
 - DOS Protected-Mode Interface server, CodeView, 336
 - DOS redirection symbol (>), HELPMMAKE syntax, 713
 - DOS Shell
 - command
 - CodeView, 358–359
 - PWB, 72, 142–143
 - creating, PWB, 214–215, 257–258
 - DOS-extended defined, 806
 - Doslibs switch, PWB, 310–311
 - /DOSS option, LINK, 561, 578
 - /DOSSEG option, LINK, 561, 578
 - Dot commands, HELPMMAKE, 713–714, 722–724
 - Dot directives, NMAKE, 687–688
 - Double precision defined, 806
 - Down function, PWB, 151, 167
 - DPMI defined, 806
 - DPMI server. *See* DOS Protected-Mode Interface server
 - Dragging defined, 807
 - /DS option
 - HELMMAKE, 714, 773
 - LINK, 579
 - PWB, 141
 - DS register
 - CodeView syntax, 419, 450
 - LINK, 579
 - /DSALLOC option, LINK, 579
 - /DSALLOCATE option, LINK, 579
 - /DT option, PWB, 141
 - /Du option, HELPMMAKE, 714
 - Dumping
 - defined, 807
 - math registers, CodeView, 473–474
 - memory, CodeView, 438–439
 - DW operator, CodeView, 405, 414–415
 - DX register, CodeView syntax, 419, 450
 - /DY option, LINK, 561, 579
 - Dynamic address, viewing memory, CodeView, 357
 - Dynamic Data Exchange, debugging, 379–382
 - Dynamic links defined, 807
 - /DYNAMIC option, LINK, 561, 579, 601–602
 - Dynamic overlays, MOVE, 604–605
 - Dynamic-link libraries
 - debugging p-code, 389–390
 - default names, PWB switches, 310–311
 - defined, 807
 - EXEHDR output, 635
 - initialization routine, debugging, 381–382
 - LINK object files, 563
 - listing modules, CodeView, 383, 463
 - loading symbolic information, CodeView, 342
 - loading, CodeView, 363–364
 - module-definition files, LINK, 570, 613–614
 - optimizing entry/exit codes, CL options, 515
 - protected mode, specifying, 617
 - real mode, specifying, 617
 - searching, module statements, 622
 - specifying, module-definition files, 612–613
 - values, CodeView, 328
- ## E
- E command, CodeView, 422, 433, 453
 - :e command, HELPMMAKE, 722
 - e. context prefix, HELPMMAKE, 729

- /E option
 - CL, 493
 - HELPMAKE, 711–712
 - LINK, 580
 - NMAKE, 648, 678, 680
 - PWB, 142
- EAX register, CodeView syntax, 419, 450
- EBP register, CodeView syntax, 419, 450
- EBX register, CodeView syntax, 419, 450
- ECX register, CodeView syntax, 419, 450
- __edata, LINK, 578
- EDI register, CodeView syntax, 419, 450
- Edit Breakpoints command, CodeView, 364, 367
- Edit Breakpoints dialog box, CodeView, 367
- Edit menu
 - CodeView, 360
 - PWB
 - described, 73
 - functions, 143
 - predefined macros, 143
- Edit Project command, PWB, 74
- Editing
 - breakpoints, CodeView, 367
 - CONFIG.SYS, PWB, 66
 - files, Editreadonly switch, PWB, 275–276
 - macros, PWB, 111
 - Noedit function, PWB, 190
 - projects, PWB, 47–49, 55
 - repeat function, PWB, 205
 - text, menu commands, PWB, 73
- Editor, PIF, starting PWB, 66
- Editor Settings command, PWB, 75, 767
- Editreadonly switch, PWB, 263, 275
- EDX register, CodeView syntax, 419, 450
- EGA defined, 807
- /Ei option, BSCMAKE, 736
- /El option, BSCMAKE, 736
- Ellipsis (...)
 - call tree, PWB, 101
 - menu command, PWB, 80, 82, 126
- !ELSE preprocessing directive, NMAKE, 689
- !ELSEIF preprocessing directive, NMAKE, 690
- !ELSEIFDEF preprocessing directive, NMAKE, 690
- !ELSEIFNDEF preprocessing directive, NMAKE, 690
- /Em option, BSCMAKE, 736
- Emacscdel function, PWB, 151, 167–168
- Emacsnewl function, PWB, 151, 168
- Embedding text strings, CL option, 544
- EMM386.EXE
 - CodeView, 336, 338
 - defined, 807
- EMM.386.SYS, CodeView, 336, 338
- Emulator library, floating-point math, CL options, 509–514
- Emulators defined, 807
- Enablealtgr switch, PWB, 263, 276
- Enabling breakpoints, CodeView, 367, 428–429
- Encoding, HELPMAKE options, 712–713, 727
- __end, LINK, 578
- .end command, HELPMAKE, 722
- Endfile function, PWB, 151, 168
- !ENDIF preprocessing directive, NMAKE, 690
- Endline function, PWB, 151, 169
- English word, predefined expression syntax, 778, 780, 785
- Enhanced graphics adapter defined, 807
- Entab switch
 - PWB, 263, 276–277
 - white space, 127–128
- Enterinsmode switch, PWB, 263, 277–278
- Enterlogmode switch, PWB, 263, 278–279
- Enterselmode switch, PWB, 263, 278
- Entry codes
 - optimizing, 515
 - Windows functions
 - customizing, 515
 - generating, 522–523
- Entry tables, specifying, CL option, 521
- Envcursave switch, PWB, 137, 263, 279
- Environment function, PWB, 151, 169–170
- Environment strings defined, 807
- Environment tables
 - defined, 807
 - saving, in PWB, 279–280
- Environment variables
 - defined, 807
 - HELPPFILES
 - defined, 809
 - Help file location, 771
 - opening Help files, 769
 - restricting global search, 767
 - INCLUDE, BSCMAKE, 736
 - INIT
 - defined, 810
 - remote debugging, 396
 - LIB, 810

Environment variables (*continued*)

- LINK
 - clearing, 594
 - defined, 810
 - setting, 593–594
 - macros, NMAKE, 678
 - menu commands, PWB, 75
 - NO87, 513
 - PATH, installing CodeView, 327
 - PWB
 - function, 169–170
 - starting, 67
 - TOOLS.INI file, 137
 - specifying options, CL, 557–559
 - SYSTEM, 817
 - TEMP, 817
 - TMP, 817
- Environment Variables command, PWB, 75
- Envprojsave switch, PWB, 137, 263, 280
- /EP option, CL, 494
- Equal sign (=)
 - module statement syntax, 610
 - Redirect Input and Output command, CodeView, 424, 477
 - substituting for number sign, 492
- /Er option, BSCMAKE, 736
- .ERR files defined, 807
- Error bit, LINK, clearing with EXEHDR, 631, 634–636
- Error checking, turning off, NMAKE, 661
- Error codes
 - CVPACK, 745
 - defined, 807
 - LIB, 708
 - LINK, 596
 - NMAKE
 - described, 696
 - from commands, 662–663
 - ignoring, 649, 687
 - SBRPACK, 741
 - Windows applications, optimizing, 515
 - Windows functions
 - customizing, 515
 - generating, 522–523
- Error messages, Help, 764
- Error numbers, HELPMMAKE context prefix, 729
- Error output, NMAKE, 650
- !ERROR preprocessing directive, NMAKE, 689
- Errors
 - building a PWB program, 46, 50
 - defined, 815

Errors (*continued*)

- Help, getting, 764
 - menu commands, PWB, 74
 - resolving references, LINK, 569
 - simulating in CodeView, 385, 464–465
- /Es option, BSCMAKE, 736
- ES register, CodeView syntax, 419, 450
- Escape sequence
 - CodeView expressions, 408
 - defined, 807
- Escapes, regular expression syntax, 778, 780–781, 786–787
- ESI register, CodeView syntax, 419, 450
- ESP register, CodeView syntax, 419, 450
- Eval entry, TOOLS.INI file
 - CodeView, 328–331, 403–404
 - remote debugging, 393–395
- Examine Symbols command, CodeView, 423, 467–468
- Exception-mask bits, 8087 command, CodeView, 473–474
- Exclamation point (!)
 - command modifiers, NMAKE, 662
 - HELPMMAKE command, 719
 - preprocessing directives, NMAKE, 688
 - replacing text, PWB, 94
 - Shell Escape command, CodeView, 423, 468–469
- .EXE files
 - defined, 807
 - overlaid DOS programs, 597
- Executable files
 - adding to, module-definition files, 614
 - creating, CL, 486–487
 - defined, 807
 - EXEHDR output, 632–636
 - format, 631
 - Header Utility. *See* EXEHDR
 - LINK, 563
 - renaming, CL, 499
- .execute command, HELPMMAKE, 722
- Execute command, PWB, 74
- Execute Commands option, CodeView, 340
- Execute function, PWB, 108, 151, 170
- EXECUTEONLY keyword, module-definition files, 621
- EXECUTEREAD keyword, module-definition files, 621
- Executing
 - commands, PWB, 78–82, 142, 219
 - functions, PWB, 106–108, 170
 - macros, PWB, 106–108

Execution

- controlling, CodeView, 386
- model, specifying, CodeView, 333
- time, optimizing, CL option, 539

exefile field, LINK, 566

EXEHDR

- application type, setting, 630
- command line, 629–631
- DLL output, 635
- error bits, clearing, 631, 634–636
- executable-file format, 631
- exports tables, 636, 638
- heap allocation, 630
- Help, 630
- memory allocation, 630
- output
 - DOS executable files, 632–633
 - segmented executable files, 634–636
 - verbose output, 637–639
- overview, 629
- relocations, 639
- segment tables, 635–638
- syntax, 629–631

/EXEPACK option, LINK

- debugging considerations, 325
- described, 580

EXETYPE statement

- module-definition files, 609, 615–616
- segmented files, LINK, 564

EXIST operator, NMAKE, 690–691

Exit codes

- CVPACK, 745
- defined, 807
- LIB, 708
- LINK, 596
- NMAKE
 - described, 696
 - from commands, 662–663
 - ignoring, 649, 687
- SBRPACK, 741
- Windows
 - applications, optimizing, 515
 - functions, customizing, 515

Exit command

- CodeView, 358, 360
- PWB, 72, 142–143

Exit function, PWB, 151, 171

Exit sequence, optimizing, CL option, 538

Exiting

- CodeView, 360
- PWB, 47, 171, 251

EXP

- command line, 750
- options, 750
- overview, 743, 747–748
- syntax, 750

Expanded memory

- defined, 807
- overlaid DOS programs, 598

Expanded memory

- emulator defined, 808
- manager defined, 808

Expanding elements in CodeView, 367–368, 478–479

Expansion, inline, controlling, CL option, 532

Explicit allocation

- defined, 808
- ordering functions, module-definition files, 626

Explicit links, HELPMMAKE, 718

Explicitly allocated functions, overlaid DOS programs, 599

__export keyword, entry/exit code

- generating, 522–523
- optimizing, 515

Export ordinals, searching for, module-definition files, 622

EXPORTS statement, module-definition files, decorated names, 609, 623, 790–791

Exports tables, EXEHDR output, 636, 638

Expression evaluators, CodeView

- choosing, 403–404, 454–455
- defined, 399
- listing, 370
- numbers, 407–408
- operators, 404–407
- specifying, 331
- string literals, 408
- symbol format, 409

Expressions

- address ranges, 402–403, 420–421
- addresses, 401–402, 420
- C++, in CodeView, 409–411
- constant, defined, 805
- defined, 808
- displaying, CodeView, 477–478
- editing, CodeView, 351
- line number, 400, 418
- live, creating, 357
- overview, CodeView, 399
- predefined. *See* Predefined expressions
- preprocessing directives, NMAKE, 690–692
- registers, 401, 419

- Expressions (*continued*)
 - regular. *See* Regular expressions
 - setting breakpoints, CodeView, 365
 - tagged. *See* Tagged expressions
 - watch. *See* Watch expressions
 - Expunging files. *See* EXP
 - Extended ASCII defined, 808
 - Extended dictionaries
 - defined, 808
 - resolving references, LINK, 584
 - suppressing, in LIB, 701
 - Extended libraries, resolving references, LINK, 569
 - Extended memory
 - browser database, 733–737
 - defined, 808
 - Keepmem switch, PWB, 285
 - overlaid DOS programs, 598
 - Extended memory manager
 - CodeView, 336
 - defined, 808
 - Extending operations line, 702
 - Extension switches, PWB, 265
 - Extensions
 - autoloading, PWB, 131, 268
 - Curfileext predefined macro, PWB, 225
 - default
 - CL, 495
 - PWB, 274
 - defined, 808
 - IMPLIB, 746
 - language, CL options, 550–552
 - LINK, 565–567
 - loading, PWB, 286
 - processing, CL, 486
 - specifying, CL, 496
 - External names, restricting length, CL option, 525
 - External references defined, 808
- F**
- /f option, CL, 494–495
 - /F option
 - CL, 494, 527
 - CodeView, 338, 341
 - LINK, 580–581
 - NMAKE, 648
 - RM, 748
 - F1 key, Help, 757
 - /Fa option, CL
 - option interactions, 495–497
 - translating source code, 501–502
 - Factor switch, PWB, 263, 280–281
 - Far address defined, 808
 - Far calls
 - LINK, 584, 587–588
 - overlaid DOS programs, 603–604
 - Far functions, optimizing entry/exit codes, CL
 - options, 515
 - __far keyword, CL
 - /A options, 489
 - accepting, 555
 - data allocation, 523–524
 - enabling options, 550
 - /FARCALL option, LINK, 580–581
 - /FARCALLTRANSLATION option, LINK, 580–581
 - Fast functions, PWB switches, 280
 - __fastcall keyword
 - calling conventions, CL options, 520
 - enabling, 550
 - naming conventions, 518
 - symbol format, CodeView, 409
 - Fastfunc switch, PWB, 263, 281–282
 - FAT defined, 808
 - Fatal errors
 - defined, 808
 - simulating, CodeView, 385, 464–465
 - /FBr option, BSCMAKE, 732
 - /FBx option, BSCMAKE, 732
 - /Fc option, CL
 - option interactions, 496–497
 - output files, 495
 - translating source code, 503–505
 - .FD files defined, 808
 - /Fe option, CL
 - option interactions, 496
 - rename executable file, 499
 - specifying output files, 495
 - FFLAGS options macro, NMAKE, 676
 - .FI files defined, 808
 - \fi formatting code, HELPMMAKE, 726
 - Fields
 - BSCMAKE, syntax, 735
 - LIB, specifying, 699–705
 - LINK
 - deffile, 570
 - exefile, 566
 - libraries, 567–570
 - mapfile, 567
 - objfiles, 565
 - overview, 564–572
 - SBRPACK, syntax, 740

- File allocation table defined, 808
- File Expunge Utility. *See* EXP
- File handle defined, 808
- File Header Utility. *See* EXEHDR
- File history
 - maximum files, setting, 300
 - PWB, 72
- File menu
 - CodeView, 358–360
 - PWB
 - described, 72
 - predefined macros, 142
- File Removal Utility. *See* RM
- File Undelete Utility. *See* UNDEL
- filename! command, HELPMMAKE, 718
- Filename extensions
 - autoloading, PWB, 268
 - Curfileext predefined macro, PWB, 225
 - default
 - CL, 495
 - PWB, 274
 - defined, 808
 - IMPLIB, 746
 - LINK, 565–567
 - loading, PWB, 286
 - processing, CL, 486
- Filename macros, NMAKE, 672–673
- Filename-extension tags, TOOLS.INI file, PWB, 132–133
- Filename-Parts Syntax, PWB switches, 265–266
- Filenames
 - alternate, setting, CL, 495
 - base names
 - Curfilenam predefined macro, PWB, 225
 - defined, 804
 - Shortnames switch, PWB, 296
 - defined, 808
 - macros, NMAKE, 672–673
 - path specification, CL, 496
 - precompiled headers, 546
 - predefined expression syntax, 778, 780, 785
 - specifying
 - HELMMAKE, 712–714
 - LINK, 566–567
 - NMAKE, 648
 - syntax
 - LINK, 565
 - NMAKE, 663–664
- Files
 - adding, PWB, 44, 48, 50
 - assembly. *See* Assembly files
- Files (*continued*)
 - backup, 95, 303, 747–750
 - batch, executing CL, 490
 - browser, generating from CL, 507–508
 - closing, 72, 163, 233
 - CodeView requirements, 328–329
 - command
 - defined, 805
 - NMAKE, 651
 - compacting for CodeView, CVPACK, 743–744
 - compiling, PWB, 234
 - creating, PWB, 245
 - deleting
 - during debugging session, 359
 - PWB, 48
 - RM, 748–749
 - editing, Editreadonly switch, PWB, 275
 - estimating size, PWB, 103–104
 - executable. *See* Executable files
 - expunging, 750
 - finding, PWB, 72
 - Help. *See* Help files
 - include. *See* Include files
 - inline, NMAKE, 664–667
 - inserting, module statements, 627
 - library. *See* Library files
 - listing, PWB, 94
 - loading, PWB, 142
 - machine-code, creating listing, CL, 502–505
 - makefiles. *See* Makefiles
 - map
 - creating, CL, 505–507
 - LINK, 582–583
 - module-definition. *See* Module-definition files
 - moving
 - PWB, 49
 - RM, 748–749
 - naming, SBRPACK, 740
 - object. *See* Object files
 - opening, PWB, 72, 141, 191, 289–290
 - optimizing size, CL option, 538
 - output
 - alternate, CL, 495
 - LINK, 566
 - packing, SBRPACK, 739–740
 - preprocessing output, creating, 540
 - printing
 - CodeView, 359
 - PWB, 194–195
 - project file list, PWB, 43–44
 - relocatable, LINK, 563

Files (*continued*)

- remote debugging, 393–395
 - removing library name, CL option, 553–554
 - response
 - BSCMAKE, 738
 - LINK, 573–575
 - restoring, UNDEL, 749
 - saving
 - Autosave switch, PWB, 269
 - PWB, 72, 209, 255–256, 300
 - searching, PWB, 86–90
 - segmented. *See* Segmented executable files
 - source. *See* Source files
 - specifying type, HELPMMAKE, 713
 - startup, PWB, 137
 - state. *See* State file
 - status, PWB, 138–139
 - temporary, LINK, 595
 - truncated, BSCMAKE, 733–735
- Filetab switch, PWB, 127–128, 263, 282
- Find command
- CodeView, 361
 - PWB, 72–73, 87–90
- Find Dialog box, CodeView, 361
- Finding
- files, PWB, 72
 - symbol definitions, PWB, 98–103
 - text in PWB, 91–93
 - Mreplace function, 185
 - Mreplaceall function, 185–186
 - Qreplace function, 202
 - Replace function, 205–207
- FIXED keyword, module-definition files, 621
- Fixup defined, 808
- /Fl option, CL
- combined listing, 502–503
 - option interactions, 496
 - output file, 495
- Flags
- 8087 command, CodeView, 473–474
 - changing values, CodeView, 450–452
 - displaying value, CodeView, 354–355
 - register defined, 809
- Flat memory model defined, 809
- Flipping screen exchange
- CodeView, 341, 445–447
 - defined, 809
- Float consistency
- improving, CL options, 537–538
 - optimizing, CL option, 537–538

Floating-point math

- compatibility between, 512
 - intrinsic functions, generating, 534–535
 - library selection, CL options, 509–514
 - specifying, CL options, 508
- Flow control statements, 112–114
- /Fm option, CL
- mapfile, 505–507, 527
 - option interactions, 496
 - output file, 495
- /Fo option, CL, 495, 498
- FOR command macro, NMAKE, 676
- .FOR files defined, 809
- Foreign makefiles in PWB, 61–63
- Format
- commands, CodeView, 352–353, 417
 - decorated names, 789
 - executable files, 631
 - HELMMAKE
 - described, 716
 - QuickHelp, 716–724
 - rich text format, 725–727
 - memory, changing, 356–357
- Formatting attributes, HELPMMAKE, 718, 721
- Formatting codes, rich text format, HELPMMAKE, 726
- Formatting text, HELPMMAKE topics, 721
- FORTTRAN Compiler, NMAKE macros, 676
- __fortran keyword, CL
- calling conventions, 516–518
 - enabling, 550
- /Fp option, CL, 495, 546
- /FPa option, CL, 508–509, 527
- /FPc option, CL, 508–509, 527
- /FPc87 option, CL, 508–510, 527
- /FPi option, CL, 508–513, 527
- /FPi87 option, CL, 508–513, 527
- /FR option
- BSCMAKE, 732
 - CL, 495–496, 507–508
- /Fr option
- BSCMAKE, 732
 - CL, 495–496, 507–508
- Frame sorting, CL option, 539
- Frames defined, 809
- .freeze command, HELPMMAKE, 722
- Friction switch, PWB, 264, 282–283
- /Fs option, CL
- described, 495, 501
 - option interactions, 496–497

Full build, building a database, BSCMAKE, 733, 735

Full-screen application defined, 809

Function calls

defined, 809

replacing, CL option, 534–535

Function Hierarchy command, PWB

described, 76

function, 145

Functions

allocated, overlaid DOS programs, 599

calling, CodeView expressions, 405

defined, 809

exporting, module-definition files, 623

importing, module-definition files, 624–625

intrinsic, CL options, 534–535

listing, CodeView, 435–436

far, optimizing entry/exit codes, 515

ordering, module-definition files, 626–627

packaged

CL options, 524

overlaid DOS programs, 599

prototypes, listing, Cl option, 552–553

PWB

Arg, 94, 106–108

Assign, 108, 121–122, 124

Backtab, 127–128

call tree, 99–101

closing, 220

cursor-movement commands, 154–156

executing, 106–108

functions, 150–154, 170–221

Linsert, 108

listing references, 102

mark, 86

menu commands, 142, 144–146

Meta, 107–108

Mgrep, 87

modifying, 181

Msearch, 87

Paste, 94, 108

Prompt, 116–117

Psearch, 87, 107

tabs, 127–129

Tell, 108

tracing, CodeView, 452–453

FUNCTIONS statement

module-definition files, 609, 626–627, 790–791

overlaid DOS programs, 600–601

overlay number, LINK, 564, 598

G

G command, CodeView, 422, 433–434

:g command, HELPMMAKE, 723

/G option, CodeView, 338, 341

/G0 option, CL, 514–515

/G1 option, CL, 514–515

/G2 option, CL, 514–515

/G3 option, CL, 514–515

/G4 option, CL, 514–515

/GA option, CL, 515

/Gc option, CL, 516–518

/GD option, CL, 515

/Gd option, CL, 516–518

/GE option, CL, 515–516

/Ge option, CL, 518–520

/GEa option, CL, 515

/GEd option, CL, 515

/GEe option, CL, 515

/GEf option, CL, 515

/GEm option, CL, 515

/GEr option, CL, 515

/GEs option, CL, 515

Gigabyte defined, 809

Global contexts, help files, linking, 719–720

Global heaps, listing memory objects, CodeView, 382, 462–463

Global memory handles, converting to pointers, 463–464

Global Search command, PWB, 78, 766–767

Global symbols

defined, 809

searching for, CodeView, 406–407

GlobalLock routine, locking memory handles, 386

/Gn option, CL, 520–521

Go command, CodeView, 422, 433–434

Goto command, PWB, predefined macros, 144

Goto Definition command, PWB

described, 76

finding symbols, 98–99

function, 145

Goto Error command, PWB, 74

Goto Mark command, PWB, 73

Goto Match command, PWB

described, 73

predefined macros, 144

Goto Reference command, PWB, 76, 145

/Gp option, CL, 521

/Gq option, CL, 521–522

/Gr option, CL, 520

Grandparent process defined, 809

Graphic function, PWB, 151, 172
 Gray, dark, color value, 273
 Greater-than operator (>), Redirect Input
 command, CodeView, 340, 424, 476
 Green, color value, 273
 Group defined, 809
 /Gs option, CL, 518–520
 /Gt option, CL, 522
 /GW option, CL, 522–523
 /Gw option, CL, 522–523
 /Gx option, CL, 523–524
 /Gy option, CL, 524, 599

H

H command, CodeView, 422, 434
 .H files defined, 809
 /H option
 CL, 525
 CVPACK, 745
 IMPLIB, 747
 LIB, 701
 Handlers, symbol, specifying, 334–336
 /HE option, LINK, 581
 /HEA option, EXEHDR, 630
 Header files, unreferenced symbols, packing files,
 739
 Headers. *See* EXEHDR
 Heap allocation, setting in EXEHDR, 630
 /HEAP option, EXEHDR, 630
 Heaps
 global, listing memory objects, 382, 462–463
 local, listing memory objects, 383
 overlaid DOS programs, 598, 602
 size, specifying, 617–618
 HEAPSIZE statement, module-definition files,
 609, 617–618
 Height switch, PWB, 264, 283
 /HEL option, EXEHDR, 630
 Help
 See also CodeView; Help files; Microsoft Advisor;
 QuickHelp
 displaying in PWB, 198, 237, 239
 getting
 CodeView, 756–765
 HELPMMAKE, 714
 index table, PWB, 239–240
 load state, PWB, 235
 next topic, PWB, 198–199, 236
 previous topic, PWB, 237

Help (*continued*)
 searching, PWB, 199, 240
 structure, CodeView, 755
 switches, 313–315
 topic selection, PWB switch, 315
 topic, PWB, 238
 Help command
 CodeView, 373–374, 422, 434
 PWB, 77
 Help database
 compressing, 711–712
 context prefixes, 729
 creating, 711–712
 decoding, 713–714
 decompressing, 714
 overview, 710–711
 Help delimiters (>>), HELPMMAKE, 726, 728
 Help File Maintenance Utility. *See* HELPMMAKE
 Help files
 closing
 PWB, 213–214
 QuickHelp, 769
 concatenating, 772
 creating, 711–712
 decoding, 713–714
 decompressing, specifying buffer size, 332
 formats
 described, 716
 minimally formatted ASCII, 728
 QuickHelp, 719–724
 rich text format, 725–727
 listing
 CodeView, 333
 PWB, 314, 772
 QuickHelp, 772
 locking, 713
 managing, 771–773
 opening
 Microsoft Advisor, 765–766
 PWB, 213–214
 QuickHelp, 769
 overview, 710–711
 requirements, CodeView, 328–329
 specifying, 713
 splitting, 773
 topics, defining, 716–717
 Help menu
 CodeView, 374, 757
 PWB, 78, 146, 757

- Help on Help command
 - CodeView, 374–375
 - PWB
 - described, 78, 758
 - predefined macros, 146
- /HELP option
 - BSCMAKE, 736
 - CL, 525
 - CVPACK, 745
 - EXEHDR, 630
 - EXP, 750
 - HELPMMAKE, 715
 - IMPLIB, 747
 - LIB, 701
 - LINK, 581
 - NMAKE, 648
 - RM, 748
 - SBRPACK, 740
 - UNDEL, 749
 - using, 768
- Help window
 - CodeView
 - function, 357–358
 - opening, 374
 - overview, 348
 - using, 760–761
 - PWB
 - default key assignments, 149
 - using, 760–761
 - setting size, 313–314
- Helpautosize switch, PWB, 313–314
- Helpbuffer entry, TOOLS.INI file, CodeView, 330, 332
- helpfile! contextstring command, HELPMMAKE, 718
- Helpfiles entry, TOOLS.INI file, CodeView, 330, 333
- HELPPFILES environment variable
 - defined, 809
 - Help file location, 771
 - opening Help files, 769
 - restricting global search, 767
- Helpfiles switch, PWB, 314, 767
- Helplist switch, PWB, 315
- HELPMMAKE
 - compatibility, 709
 - context prefixes, 729
 - decoding, 713–714
 - defining topics, 716–717
 - dot commands, 722–724
 - encoding, 711–713, 727
- HELPMMAKE (*continued*)
 - formats
 - described, 716
 - minimally formatted ASCII, 728
 - QuickHelp, 716–724
 - rich text, 725–726
 - rich text format, 726–727
 - specifying, 713
 - formatting attributes, 718–721
 - formatting text, 721
 - getting Help, 714
 - global contexts, 719–720
 - local contexts, 720
 - options
 - decoding, 713–714
 - encoding, 712–713
 - overview, 710–711
 - syntax, 711–715
- Helpwindow switch, PWB, 315
- Hexadecimal numbers
 - predefined expression syntax, 778, 785
 - predefined expressions syntax, 780
- Hexadecimal defined, 809
- /HI option, LINK, 581
- High memory defined, 809
- /HIGH option, LINK, 581–582
- Highlight defined, 809
- Highlighting search strings in PWB, 210
- Hike switch, PWB, 264, 283–284
- HIMEM.SYS
 - CodeView, 336
 - defined, 809
 - .HLP files defined, 809
- Home function, PWB, 151, 172
- Horizontal Scrollbars command, CodeView, 368, 370
- HPFS defined, 809
- Hscroll switch, PWB, 264, 284
- __huge keyword, CL
 - /A options, 489
 - enabling, 550
- Huge memory model defined, 809
- Hyperlinks, Microsoft Advisor
 - index screens, 764
 - navigating with, 759–761

I

- I command, CodeView, 422, 434–435
- :i command, HELPMMAKE, 723
- \i formatting attribute, HELPMMAKE, 721

- `\i` formatting code, HELPMMAKE, 726
- `/I` option
 - CL, 525–526
 - CodeView, 338, 341–342
 - LIB, 701
 - NMAKE, 649
 - RM, 748
- Icons, screen display, PWB, 68
- Identifiers
 - case distinction, LINK option, 585
 - C/C++, predefined expression syntax, 778, 780, 785
 - defined, 809
 - searching, PWB, 93
- IEEE format defined, 810
- `!IF` preprocessing directive, NMAKE, 689
- `!IFDEF` preprocessing directive, NMAKE, 689
- `!IFNDEF` preprocessing directive, NMAKE, 689
- `.IGNORE` dot directive, NMAKE, 687
- `/IGNORECASE` option, LIB, 701
- IMPLIB
 - case sensitivity, 747
 - command lines, 746–747
 - LINK import libraries, 568
 - module-definition files, 609
 - options, 747
 - overview, 743, 745–747
 - syntax, 746–747
- Implicit links, HELPMMAKE, 719–720
- Import libraries
 - combining, 704
 - creating, IMPLIB, 745–747
 - defined, 810
 - linking, 568
- Import Library Manager. *See* IMPLIB
- IMPORTS statement, module-definition files, 609, 624–625, 790–791
- Improving float consistency, CL options, 537–538
- `.INC` files defined, 810
- `#include` directive, PWB project dependencies, 45
- INCLUDE environment variable, BSCMAKE, 736
- Include files
 - browser database, PWB, 104
 - BSCMAKE, 737
 - defined, 810
 - finding symbols, PWB, 101
 - project dependencies, PWB, 45, 48
 - search directory, CL option, 525–526
 - unreferenced symbols, packing files, 739
- `!INCLUDE` preprocessing directive, NMAKE, 689
- INCLUDE statement
 - module-definition files, 609, 627
 - overlaid DOS programs, 600–601
- Incremental build, building a database, BSCMAKE, 733–735
- Indenting
 - automatic, PWBC switches, 312
 - command line, NMAKE, 660
 - dependency lines, NMAKE, 655
 - text
 - HELPMMAKE, 726
 - PWB, 296
- Index command
 - CodeView, 374
 - PWB, 78, 146, 757
- Index screens, Microsoft Advisor, 762, 764
- Indirection register, debugging assembly language, 414
- `/INF` option, LINK, 582
- Inference rules, NMAKE
 - commands, 660
 - inferred dependents, 685–686
 - precedence, 686–687
 - predefined, 684–685
 - search paths, 682
 - syntax, 680–682
 - user-defined, 682–684
- Inferred dependents, NMAKE
 - dependency line, 659
 - inference rules, 685–686
- Infinite loops, terminating execution, 387–388
- `/INFO` option, LINK, 562, 582, 601–602
- Infodialog switch, PWB, 264, 285–286
- Information function, PWB, 151, 173
- `/INFORMATION` option, LINK, 562, 582
- Inheritance, using C++ expressions, 410
- Inherited macros, NMAKE, 679–680
- `.INI` files defined, 810
- INIT environment variable
 - defined, 810
 - PWB, starting, 67
 - remote debugging, 396
- INIT environment variable, PWB, 137
- Initialization routine, debugging, 381–382
- Initialize function, PWB, 151, 172–173
- Inline code, debugging, 322
- Inline expansion, control, CL option, 532
- Inline files, NMAKE
 - creating, 664–666
 - multiple, NMAKE, 667
 - reusing, 666–667

Input
 LINK, 564–572
 redirecting, CodeView, 475, 477
 Insert function, PWB, 151, 173–174
 Insert mode, toggling, in PWB, 174, 277
 Inserting
 characters, PWB, 173–174
 files, module statements, 627
 lines, PWB, 177–178
 RTF formatting codes, HELPMMAKE, 726
 space, PWB, 215–216
 Insertmode function, PWB, 151, 174
 Installable file system defined, 810
 Installing CodeView, 327–329
 Instruction sets, generating, CL option, 514–515
 int, searching, PWB, 91–93
 Integers defined, 810
 Interoverlay calls
 defined, 810
 limiting, LINK, 579
 Interrupt call defined, 810
 __interrupt keyword, CL, enabling, 550
 Interrupt number, MOVE, 604
 Interrupting, CodeView, 387, 470, 475
 Interrupts, trapping, CodeView, 341–342
 Intrinsic functions, calling, CodeView expressions,
 405
 Invariant code, removing, CL option, 535–536
 I/O privilege mechanism defined, 810
 Italics, HELPMMAKE formatting
 QuickHelp format, 721
 rich text format, 726
 /Iu option, BSCMAKE, 737

J

/J option, CL, 526–527
 Justifying tagged expressions, 785

K

K command, CodeView, 435–436
 /K option
 HELPMMAKE, 712–713
 NMAKE, 649
 RM, 748
 KEEP, inline files, NMAKE, 665
 Keepmem switch, PWB, 264, 285
 Key assignments, PWB
 cursor movement commands, 154–155
 default, 146–150
 described, 107, 119–121, 134–135

Key assignments (*continued*)
 Graphic function, 172
 menu commands, 142–146
 Unassigned function, 218
 Key Assignments command, PWB, 75
 Key box, assigning key function, PWB, 120
 Keyboard
 choosing commands, 78
 executing PWB commands, 78–79
 hyperlinks, activating, 759
 navigation in CodeView, 349
 shortcut keys, PWB, 79
 Keys
 shortcut, PWB, 79
 TOOLS.INI syntax, PWB, 135
 Keywords
 /A options, CL, 489
 compressing, HELPMMAKE option, 712–713
 Help, getting, 762–763
 specific keyword, 489
 Kilobyte defined, 810

L

L command, CodeView, 422, 436–437
 :l command, HELPMMAKE, 723
 /L option
 CL, 380–382
 CodeView, 338
 HELPMMAKE, 713
 /L options, CodeView, 342
 Label defined, 810
 Label/Function command, CodeView, 361–362
 Language command, CodeView, 368, 370
 Language dialog box, CodeView, 370
 Language extensions, enabling, CL options,
 550–552
 Language Options command, PWB, 75
 Large memory model defined, 810
 Lastproject switch, PWB, 141, 264, 285–286
 Lastselect function, PWB, 151, 175
 Lasttext function, PWB, 151, 175–176
 /Ld option, CL, 527
 Ldelete function, PWB, 152, 176–177
 Least-recently-used (LRU) algorithm, overlaid
 DOS programs, 598
 Leaving
 CodeView, 360
 PWB, 171, 251
 Left function, PWB, 152, 177
 .length command, HELPMMAKE, 723, 726

Less than operator (<), Redirect Output command,
CodeView, 340, 475

\li formatting code, HELPMMAKE, 726

/LI option, LINK, 582

LIB

- case sensitivity, 701
- combining libraries, 704
- command line, 698
- commands, 702–705
- consistency checks, 700
- cross-reference listing, 705–706
- error codes, 708
- extended dictionaries, suppressing, 701
- fields, specifying, 699–705
- file compatibility, 697–698
- Help, 701
- library files, 700, 703
- LINK import libraries, 568
- object modules
 - adding, 703
 - copying, 705
 - deleting, 704
 - distinguishing, 698
 - moving, 705
 - replacing, 704–705
- options, 700–701
- output library, 706–707
- overview, 697–698
- page size, 701–702
- prompts, 698–699
- response file, 699
- syntax, 699–705

LIB environment variable defined, 810

.LIB files defined, 810

LibEntry routine, debugging, 381–382

Libraries

See also LIB

combining, 704

default

defined, 806

LINK, 568, 583–584

defined, 810

import. *See* Import libraries

load, defined, 811

managing, with LIB, 697–698

math operations, CL options, 508–513

MOVE, 600

output, 706

quick, LINK, 590–591

removing name, CL options, 553–554

searching for files, LINK, 570, 583–584

Libraries (*continued*)

selecting, CL options, 527

specifying, LINK, 566–570

standard, defined, 816

static, defined, 816

switches, 310–312

libraries field, LINK, 567–570

Library files

creating, 700, 703

PWB, 43–44

Library Manager. *See* LIB

Library modules. *See* Object modules

Library routines, overlaid DOS programs, 600

LIBRARY statement

file extension, LINK, 566

module-definition files, 609

segmented files, LINK, 564

LIM EMS defined, 810

.line command, HELPMMAKE, 723

Line continuation character

NMAKE, 655, 660, 669

PWB, 115, 117, 136

#line directives, CL

adding to output, 493–494

preprocessor-output files, 540

\line formatting code, HELPMMAKE, 727

Line Mode command, PWB, 73, 143

Line numbers, CodeView expressions, 400, 418

/LINE option, LINK, 582

Line selection mode, setting in PWB, 241

Line-display mode, setting, CodeView, 339

LINENUMBERS option, LINK, 582

Lines

blank in NMAKE, command line, 660

deleting, PWB, 176–177

inserting, PWB, 177–178

multiple statements, debugging, 322

trailing, display mode, in PWB, 301–302

LINK

case distinction, 585

CL linking options, 527–528

command line, 564–568, 570–572

data segments loading, 579

debugging, 577

decorated names, 790

defaults, 573

deffile field, 570

environment variable, 593–594, 810

error bits, clearing with EXEHDR, 631, 634–636

exefile field, 566

exit codes, 596

- LINK (*continued*)
- far calls, 580–581, 584
 - Help, 581
 - interoverlay calls, limiting, 579
 - invoking, from CL, 486
 - libraries, 567–570
 - libraries field, 567–570
 - map files, 582–583
 - mapfile field, 567
 - module-definition files, 600–601, 609
 - new features, 561–562
 - objfiles field, 565
 - optimizing relocation table, 580
 - options
 - debugging considerations, 323–325
 - described, 575–576
 - ordering functions, module-definition files, 626–627
 - ordering segments, 578, 585
 - output files, 563–564
 - overlaid programs, 597–602
 - overview, 563
 - packing
 - code segments, 587–588
 - data segments, 588–589
 - prompts, 573–576
 - PWB menu commands, 75
 - response files, 573–575
 - running, 572
 - searching
 - for object files, 565
 - libraries, 583–584
 - space allocation, 577–578
 - syntax, 564–572, 575
 - temporary files, 595
- LINK environment variable, 593–594, 810
- /link option, CL, 527
- LINK Options command, PWB, 75
- Linker. *See* LINK
- Linking
 - debugging considerations, 323–325
 - defined, 810, 816
 - function-level, CL options, 524
 - topics, HELPMMAKE, 717–719
- Link-time possibilities, CL options, 508
- Linsert function, PWB, 108, 152, 177–178
- List box, PWB, 81
- I. lst command, HELPMMAKE, 723
- List files defined, 810
- List References command, PWB, 76, 145
- List Watch command, CodeView, 423
- List Watch Expressions command, CodeView, 465–466
- Listing
 - assembly files, CL, 501–503, 505
 - C files, PWB, 94
 - CodeView, breakpoints, 429
 - compiler options, 525
 - cross references, in LIB, 705–706
 - defined, 811
 - expression evaluators, CodeView, 370
 - function prototypes, CL option, 552–553
 - functions, CodeView, 435–436
 - Help files, 333, 772
 - machine-code file, CL, 502–505
 - Microsoft Advisor topics, 765
 - modules, CodeView, 463
 - project files, PWB, 43–44
 - references, PWB, 102
 - source files, CL, 501
 - watch expressions, CodeView, 365, 465–466
- Literal characters
 - makefiles, NMAKE, 653
 - searching, PWB, 91
- Live expressions, CodeView, 357
- /Ln option, CL, 528
- .LNK files defined, 811
- Load command, CodeView, 362–364
- Load dialog box, CodeView, 363–364
- Load libraries
 - defined, 811
 - specifying, LINK, 566
- Load Other Files option, CodeView, 342
- Load switch, PWB, 264, 286
- __loadds keyword, naming conventions, CL option, 528–529
- Loader, replacing, module-definition files, 615
- Loading
 - data, LINK, 579
 - source files, CodeView, 359
 - symbolic information, CodeView, 342
- LOADONCALL keyword, module-definition files, 621
- Load-time relocation table, optimizing, LINK, 580
- LOC. *See* local transport layer
- Local command, CodeView, 373–374
- Local contexts, help files, linking, 720
- Local heaps, listing memory objects, CodeView, 383, 462
- Local memory handles, converting to pointers, 466–467
- Local Options command, CodeView, 368–369

Local Options dialog box, CodeView, 369
 Local symbols
 building a database, BSCMAKE, 732, 736
 defined, 811
 Local variables, listing, CodeView, 354, 369
 Local window, CodeView
 defined, 811
 function, 354
 opening, 374
 overview, 348
 LocalLock routine, locking memory handles, 386
 Locking
 help files, 713
 memory handles, 386
 Log command, PWB
 described, 73
 predefined macros, 144
 Log Search Complete dialog box, PWB, 88
 Logged search, PWB, 86, 178, 241–242, 278
 Logical segment defined, 811
 Logsearch function, PWB, 152, 178
 Long filenames, NMAKE, 654
 Long integer defined, 811
 Long jumps, MOVE programs, 604
 longjump function, MOVE programs, 604
 Loops
 infinite, terminating execution, 387–388
 optimizing, CL options, 535–537
 Low memory defined, 811
 /Lr option (CL), 528
 .LRF files defined, 811
 LRU algorithm, overlaid DOS programs, 598
 .LST files defined, 811
 l-value defined, 811
 /Lw option, CL, 527

M

:m command, HELPMMAKE, 723
 m. context prefix, HELPMMAKE, 729
 /M option
 CL, 429–432
 CodeView, 338, 342
 CVPACK, 745
 LINK, 583
 NMAKE, 649
 PWB, 142
 /MA option
 CL, 528
 EXEHDR, 630
 Machine code
 defined, 811
 translating source code, CL, 502–505
 Macro Assembler, NMAKE options macro, 676
 Macros
 changing key assignment, PWB, 119–121, 135
 commands, NMAKE, 675
 debugging, programming considerations, 322
 defined, 811
 defining
 CL, 492–493
 PWB, 157–158
 environment variables, NMAKE, 678
 executing, PWB, 106–108, 170
 filename, NMAKE, 672–673
 flow control statements, PWB, 112–114
 inherited, NMAKE, 679–680
 inheriting, NMAKE, 650
 key assignments, PWB, 146–150
 null, NMAKE, 670
 options, NMAKE, 676
 overview
 NMAKE, 667–668
 PWB, 109
 precedence, NMAKE, 680
 predefined, PWB, 142–146, 222–263
 recording, PWB, 109–112, 203–204, 252–253
 recursion, NMAKE, 674–675
 .SBR files, PWB, 104
 shortcut keys, PWB, 79
 special, NMAKE, 671–672
 substitution, NMAKE, 677–678
 TOOLS.INI syntax, PWB, 134
 undefined
 NMAKE, 670
 PWB, 224, 226
 user input statements, PWB, 114–117
 user-defined, NMAKE, 668–670
 using, NMAKE, 671
 macros field, NMAKE, 647
 Magenta, color value, 273
 main function, calling conventions, CL options, 517
 .MAK files
 See also Makefiles
 defined, 811
 MAKE recursion macro, NMAKE, 674
 MAKEDIR recursion macro, NMAKE, 674
 Makefiles
 association with .PIF files, 67
 build process, 56, 58

- Makefiles (*continued*)
 - building browser database, non-PWB projects, 104–105
 - contents, 653–654
 - customizing, 58–61
 - debugging, 648–649
 - defined, 646
 - dependency lines, 655
 - loading, PWB, 142
 - non-PWB, 61–63
 - opening, 141
 - sample, 694–696
 - sequence of operations, 692–694
- MAKEFLAGS recursion macro, NMAKE, 674
- Map files
 - creating
 - CL, 505–507
 - LINK, 582–583
 - defined, 811
- /MAP option, LINK, 562, 583
- mapfile field, LINK, 567
- .mark command, HELPMMAKE, 723
- Mark file, PWB menu commands, 73
- Mark function, PWB, 86, 152, 178–179
- Markfile switch, PWB, 122, 264, 287
- Marks
 - manipulating, in PWB, 178–179
 - saving, in PWB, 287
- MASM
 - CL options, 528
 - debugging assembly language, 412–415
 - radix, 444–445
- Match case, search option, CodeView, 361
- Match Case command, PWB, 76
- Matches, searching, PWB, 87–90
- Matching
 - characters, regular expression syntax, 779
 - regular expressions, 307–308, 788
- Math coprocessors
 - defined, 803
 - displaying registers, CodeView, 355–356
 - dumping register contents, 473–474
- Math, floating-point operations, CL options, 508–512
- /MAX option, EXEHDR, 630
- Maximize command
 - CodeView, 373–374
 - PWB
 - described, 77
 - predefined macros, 145
- Maximize function, PWB, 152, 179–180
- Maximizing windows, PWB, 179–180, 242
- MAXVAL keyword, module-definition files, 617–618
- MC command, CodeView, 422, 437–438
- MD command, CodeView, 422, 438–439, 444–445
- MDC command, CodeView, 439
- ME command, CodeView
 - described, 422, 440–441
 - input radix, 444–445
 - Restart command, and, 436–437
- Medium memory model defined, 811
- Megabyte defined, 811
- Memory
 - CodeView
 - comparing, 437–438
 - displaying, 356–357
 - dumping data, 438–439
 - entering data, 440–441
 - filling, 441–442
 - moving data, 442–443
 - searching, 443–444
 - viewing, 455–457
 - expression evaluator requirements, 403–404
 - extended
 - defined, 808
 - Keepmem switch, PWB, 285
 - LINK options, 591
 - format, changing, 356–357
 - high, defined, 809
 - increasing capacity, CL, 491
 - managing
 - CodeView, 336
 - DOS, 733–734
 - models, MOVE library, 599
 - NMAKE, running, 649
 - overlaid DOS programs, 597
 - space allocation
 - LINK, 577–578
 - MOVE, 602–603
- Memory 1 command, CodeView, 373–374
- Memory 2 command, CodeView, 373–374
- Memory allocation, setting in EXEHDR, 630
- Memory Compare command, CodeView, 422, 437–438
- Memory Dump Code command, CodeView, 439
- Memory Dump command, CodeView, 422, 438–439, 444–445
- Memory Enter command, CodeView
 - described, 422, 440–441
 - input radix, 444–445
 - Restart command, and, 436–437

- Memory Fill command, CodeView, 422, 436–437, 441–442
- Memory handles
 - converting objects to pointers, 462–464, 466–467
 - dereferencing, CodeView, 386
- Memory models
 - defined, 811
 - floating-point options, 513–514
 - segmenting, CL options, 488–490
- Memory Move command, CodeView, 422, 442–443
- Memory objects, listing, CodeView, 382–383, 462
- Memory operators
 - CodeView, 405
 - debugging assembly language, 412–415
- Memory Search command, CodeView, 422
- Memory Window command, CodeView, 369
- Memory Window Options dialog box, CodeView, 369
- Memory windows, CodeView
 - described, 327
 - function, 356–357
 - opening, 374
 - overview, 348
 - saving addresses, 344
 - specifying, 455–457
- Memory1 Window command, CodeView, 368
- Memory2 Window command, CodeView, 368
- Menu bars
 - CodeView, overview, 346
 - defined, 811
 - PWB
 - activating, 180
 - screen display, 67–68
- Menu commands
 - adding, PWB, 125–127
 - Browse menu, PWB, 76, 145, 200
 - choosing, PWB, 78–79
 - Data menu, CodeView, 364–368
 - Edit menu
 - CodeView, 361–362
 - PWB, 73
 - executing, PWB, 78–79
 - File menu
 - CodeView, 358–360
 - PWB, 72, 142
 - Help menu
 - CodeView, 374
 - PWB, 78, 146
 - Help, getting, 764
- Menu commands (*continued*)
 - Options menu
 - CodeView, 368–372
 - PWB, 75
 - predefined macros, PWB, 142–146
 - Project menu, PWB, 74, 144
 - Run menu
 - CodeView, 362–363
 - PWB, 74, 144
 - Search menu, PWB, 73, 144
 - Window menu, PWB, 77, 145
 - Windows menu, CodeView, 373
- Menu items
 - adding, PWB Run menu, 304–306
 - custom, PWB, 260
 - Help, getting, 758
 - HELPMAKE context prefix, 729
 - Menukey function, PWB, 152, 180
- Menus
 - Browse menu, PWB, 76, 145, 200
 - Calls menu, CodeView, 372–373
 - closing, PWB, 79
 - Data menu, CodeView, 364–368
 - Edit menu
 - CodeView, 360
 - PWB, 73, 143
 - File menu
 - CodeView, 358–360
 - PWB, 72, 142
 - Help menu
 - CodeView, 374, 757
 - PWB, 78, 146, 757
 - menu bars, CodeView, 346
 - Options menu
 - CodeView, 368–372
 - PWB, 75
 - Project menu, PWB, 74, 144
 - Run menu
 - CodeView, 362–363
 - PWB, 74, 144, 260, 304–306
 - screen display, PWB, 67–68
 - Search menu
 - CodeView, 361–362
 - PWB, 73, 144
 - Window menu, PWB, 77, 145
 - Windows menu, CodeView, 373
- Merge command, PWB, 72
- Message classes, CodeView options, 429–432
- Message function, PWB, 152, 180
- Message numbers, HELPMAKE context prefix, 729

- !MESSAGE preprocessing directive, NMAKE, 689
- Messages, Windows types and class, 384–385
- Meta function, PWB, 107–108, 152, 181
- Metacharacters, searching, PWB, 91
- MF command, CodeView, 422, 436–437, 441–442
- Mgrep function, PWB, 87, 152, 181–182
- Mgreplist macro, PWB, 224, 226
- MI option, EXEHDR, 630
- Microsoft Advisor
 - copying text, 761
 - error Help, 764
 - global searches, 766–767
 - help files
 - concatenating, 772
 - listing, 772
 - managing, 771–773
 - opening, 765–766
 - splitting, 773
 - Help menus, 757
 - Help, getting, 756–765
 - hyperlinks, 759–761
 - index, 762
 - keyword Help, 762–763
 - menu items, 757
 - mouse functions, 757
 - pasting text, 761
 - Pwbhelp function, 198
 - structure, 755
- Microsoft Basic Compiler, NMAKE macros, 675–676
- Microsoft Browser Database Maintenance Utility. *See* BSCMAKE
- Microsoft Browser Information Compactor. *See* SBRPACK
- Microsoft C Compiler, NMAKE macros, 675–676
- Microsoft COBOL Compiler, NMAKE macros, 675–676
- Microsoft Debugging Information Compactor. *See* CVPACK
- Microsoft EXE File Header Utility. *See* EXEHDR
- Microsoft File Expunge Utility. *See* EXP
- Microsoft File Removal Utility. *See* RM
- Microsoft File Undelete Utility. *See* UNDEL
- Microsoft FORTRAN Compiler, NMAKE macros, 676
- Microsoft Import Library Manager. *See* IMPLIB
- Microsoft Library Manager. *See* LIB
- Microsoft Macro Assembler
 - CL options, 528
 - NMAKE macros, 675–676
- Microsoft Overlaid Virtual Environment. *See* MOVE
- Microsoft Pascal Compiler, NMAKE macros, 676
- Microsoft Program Maintenance Utility. *See* NMAKE
- Microsoft Programmer's Workbench. *See* PWB
- Microsoft Relocatable Object-Module Format (OMF), LINK, 563
- Microsoft Resource Compiler, NMAKE macros, 676
- Microsoft Segmented Executable Linker. *See* LINK
- Microsoft Static Overlay Manager, 600, 604–605
- Microsoft Symbolic Debugging Information. *See* Symbolic Debugging Information
- Microsoft Windows. *See* Windows
- Microsoft Word, rich text format, HELPMMAKE, 725–726
- /MIN option, EXEHDR, 630
- Minimally formatted ASCII, HELPMMAKE, 713, 716, 728
- Minimize command
 - CodeView, 373–374
 - PWB
 - described, 77
 - predefined macros, 145
- Minimize function, PWB, 152, 182–183
- Minimizing windows in PWB, 243–244
- /MINIMUM option, CVPACK, 745
- Minus sign (–), NMAKE, 688–689
- Mixed mode defined, 812
- Mlines function, PWB, 152, 183
- MM command, CodeView, 422, 442–443
- Mnemonics, assembling, 424–426
- Model entry, TOOLS.INI file
 - CodeView, 327–332
 - debugging p-code, 389–390
 - remote debugging, 393–395
- Models, memory. *See* Memory models
- Modes, specifying, module-definition files, 617
- Module Outline command, PWB
 - described, 76
 - function, 145
- Module statements, module-definition files
 - adding to executable files, 614
 - exporting functions, 623
 - files, specifying, 612–613
 - heap size, 617–618
 - importing functions, 624–625
 - inserting files, 627
 - inserting text, 613–614
 - mode, specifying, 617

Module statements, module-definition files
(*continued*)

- operating system, specifying, 615–616
- ordering functions, 626–627
- purpose, 609
- replacing loader, 615
- reserved words, 611
- searching DLLs, 622
- segment attributes, 618–620
- specifying files, 611–612
- stack size, 617
- syntax, 610

Module-definition files

- defined, 806, 812
- LINK, 564, 570, 598–601
- module statements
 - adding to executable files, 614
 - exporting functions, 623
 - files, specifying, 612–613
 - heap size, 617–618
 - importing functions, 624–625
 - inserting files, 627
 - inserting text, 613–614
 - mode, specifying, 617
 - operating system, specifying, 615–616
 - ordering functions, 626–627
 - purpose, 609
 - replacing loader, 615
 - reserved words, 611
 - searching DLLs, 622
 - segment attributes, 618–620
 - specifying files, 611–612
 - stack size, 617
 - syntax, 610
- MOVE, 598, 604
- new features, 607–608
- overview, 608
- PWB, 43

Modules

- configuring, CodeView, 363–364
- defined, 812
- listing, CodeView, 383, 463
- naming, CL option, 528–530
- object
 - adding, in LIB, 703
 - distinguishing, 698

Monitors

- CodeView
 - black-and-white display, 339
 - line-display mode, 339
 - redrawing, 479

Monitors (*continued*)CodeView (*continued*)

- screen exchange, 341–343, 371, 445–447, 479
- specifying, 338–339
- suppressing snow, 341
- PWB, specifying color, 271–273
- remote debugging, 396

Monochrome adapter defined, 812**Mouse**

- choosing commands, 78
- disabling, CodeView option, 342
- enabling, PWB, 288
- executing PWB commands, 78–79
- Help, getting, 756–757
- hyperlinks, activating, 759

Mouse pointer defined, 812**Mousemode switch, PWB, 264, 288****MOVABLE keyword, module-definition files, 621****MOVE**

- advantages, 605
- dynamic and static overlays, 604–605
- library, 599
- library routines, 600
- overlaid programs, 597–601
 - compatibility, 603
 - memory allocation, 602–603
 - segments, 603–604

Move command

- CodeView, 373–374
- LIB, 705
- PWB, 77, 145

_moveinit routine, 602–603**_movepause routine, 602–603****_moveresume routine, 602–603****_movesetcache routine, 602–603****_movesetheap, 602****Movewindow function, PWB, 152, 183–184****Moving**

files

- PWB, 49
- RM, 748–749
- memory blocks, CodeView, 442–443
- object modules, 705
- windows, PWB, 183–184, 244

Mpage function, PWB, 152, 184**Mpara function, PWB, 152, 184–185****MPC, debugging p-code, 390****/Mq option, CL, 528****Mreplace function, PWB, 152, 185, 289–290****Mreplaceall function, PWB, 152, 185–186****MS command, CodeView, 443–444**

MS32EM87.DLL defined, 812
 MS32KRNL.DLL defined, 812
 MS-DOS. *See* DOS
 MSDPMI defined, 812
 MSDPMI.EXE, CodeView, 336
 MSDPMI.INI defined, 812
 Msearch function, PWB, 87, 152, 186
 Msgdialog switch, PWB, 264, 288
 Msgflush switch, PWB, 264, 289
 Mtlibs switch, PWB, 310–311
 Multimodule programs, PWB
 building, 45–46
 compiler options, 53–56
 creating projects, 42
 editing, 47–49, 55
 extending projects, 58–61
 non-PWB makefiles, 61–63
 overview, 41–42
 project contents, 43–44
 project dependencies, 45, 48
 using existing projects, 49
 Multiple applications, debugging, 379–382
 MULTIPLE keyword, module-definition files, 621
 Multitasking operating system defined, 812
 Mword function, PWB, 152, 187

N

N command, CodeView, 422, 444–445
 :n command, HELPMMAKE, 724
 n. context prefix, HELPMMAKE, 729
 /N option
 CodeView, 338, 341–343
 CVPACK, 745
 NMAKE, 649
 /n option, BSCMAKE, 737
 Name decorations
 debugging considerations, 324, 409
 overview, 789–790
 specifying, 790–791
 NAME statement, module-definition files,
 609–612, 630
 Named tags, TOOLS.INI file, PWB, 133–134
 Names
 decorated
 overview, 789–790
 specifying, 790–791
 removing, CL option, 542–544
 restricting length, CL option, 525
 Naming
 files, SBRPACK, 740
 libraries, 706–707
 segments, 322, 528–530
 Naming conventions
 CL options, 518
 Pascal naming, CL option, 552
 segments, CL option, 528–529
 NAN defined, 812
 Native command, CodeView, 368, 372, 391–392
 Native entry
 points, p-code, removing, 520–521
 TOOLS.INI file
 CodeView, 329–336
 remote debugging, 393–395
 Native execution model, specifying, CodeView, 333
 native_caller pragma, p-code entry points,
 removing, 520–521
 Navigation
 CodeView windows, 349
 cursor movement commands, PWB, 154–155
 Microsoft Advisor, 756–765
 QuickSearch, 770
 windows, menu commands, PWB, 77
 /ND option, CL, 528–530
 /NE option, EXEHDR, 630
 Near address defined, 812
 __near keyword, CL
 /A options, 489
 data allocation, 523–524
 enabling, 550
 Negated set, regular expression syntax, 779
 Nested structures, expanding and contracting, 479
 New command, PWB, 72, 77, 142–145
 New line, starting, PWB, 168
 New Project command, PWB, 74
 Newfile function, PWB, 152, 187–188
 NEWFILES keyword, NAME statement,
 module-definition files, 611–612
 /NEWFILES option, EXEHDR, 630
 Newline character
 defined, 812
 module statement syntax, 610
 Newline function, PWB, 152, 188
 Newwindow switch, PWB, 264, 289–290
 .next command, HELPMMAKE, 723
 Next command, PWB
 described, 72, 76, 78
 function, 145
 predefined macros, 142–143, 146

- Next Error command, PWB
 - described, 74
 - multimodule builds, 46
 - predefined macros, 144
- Next Match command, PWB
 - described, 73
 - predefined macro, 144
 - searching, 89
- Nextmsg function, PWB, 152, 188–189, 289–290
- Nextsearch function, PWB, 152, 189–190, 289–290
- /NM option, CL, 528–530
- NMAKE
 - building projects, 45, 58–60
 - command file, 650–651
 - command line
 - command file, 650–651
 - described, 647
 - macros, defining, 669–670
 - suppressing, 688
 - commands
 - exit codes, 662–664
 - inline files, 664–667
 - macros, 675
 - modifiers, 661–662
 - syntax, 660
 - description blocks
 - dependency lines, 655
 - dependents, 659–660
 - overview, 655
 - targets, 656–658
 - directives
 - dot, 687–688
 - preprocessing, 688, 690–692
 - exit codes
 - commands, 662–664
 - described, 696
 - ignoring, 649, 687
 - inference rules
 - inferred dependents, 685–686
 - overview, 680–681
 - precedence, 686–687
 - predefined, 684–685
 - search paths, 682
 - syntax, 681–682
 - user-defined, 682–684
 - macros
 - commands, 675
 - environment variables, 678
 - filename, 672–673
 - inherited, 679–680
 - NMAKE (*continued*)
 - macros (*continued*)
 - null, 670
 - options, 676
 - overview, 667–668
 - precedence, 680
 - recursion, 674–675
 - special, 671
 - substitution, 677–678
 - undefined, 670
 - user-defined, 668–670
 - using, 671
 - makefiles
 - contents, 653–654
 - PWB, 62–63
 - new features, 645
 - options
 - macros, 676
 - turning on, 688–689
 - overview, 646
 - running, 647
 - sample makefile, 694–696
 - sequence of operations, 692–694
 - TOOLS.INI file, 652
 - NMAKE Options command, PWB, 75
 - NMD1PCD.DCC, 389–390
 - NMWOPCD.DCC, 389–390
 - /NO option, EXEHDR, 630
 - NO87 environment variable, CL, 513
 - /NOD option, LINK, 583–584
 - NODATA keyword, module-definition files, 623
 - /NODEFAULTLIBRARYSEARCH option, LINK, 583–584
 - /NOE option
 - LIB, 701
 - LINK, 584
 - Noedit function, PWB, 152, 190
 - /NOEXTDICTIONARY option, LINK, 584
 - /NOEXTDICTIONARY option, LIB, 701
 - /NOF option, LINK, 584
 - /NOFARCALL option, LINK, 584
 - /NOFARCALLTRANSLATION option, LINK, 584
 - /NOG option, LINK, 584
 - /NOGROUP option, LINK, 584
 - /NOGROUPASSOCIATION option, LINK, 584
 - /NOI option
 - IMPLIB, 747
 - LIB, 701
 - LINK, 585

- /NOIGNORECASE option
 - IMPLIB, 747
 - LIB, 701
 - LINK, 585
 - Noise switch, PWB, 264, 290
 - NOKEEP, inline files, NMAKE, 665
 - /NOL option
 - IMPLIB, 747
 - LIB, 701
 - LINK, 585, 594
 - /NOLOGO option
 - BSCMAKE, 737
 - CL, 530
 - CVPACK, 745
 - EXEHDR, 630
 - HELPMMAKE, 715
 - IMPLIB, 747
 - LIB, 701
 - LINK, 585, 594
 - NMAKE, 649
 - SBRPACK, 740
 - /NON option, LINK, 585
 - Non-UNIX regular expression syntax, setting in
 - PWB, 303–304
 - NONDISCARDABLE keyword, module-definition files, 620
 - NONE keyword, module-definition files, 621
 - Nonmaskable-interrupt, CodeView option, 341–342
 - Nonmaskable-Interrupt Trapping option,
 - CodeView, 343
 - NONSHARED keyword, module-definition files, 622
 - /NONULLS option, LINK, 585
 - /NONULLSDOSSEG option, LINK, 585
 - Non-UNIX predefined expressions, syntax, 780
 - Non-UNIX regular expressions
 - matching method, 788
 - syntax, 780, 786
 - /NOPACKC option, LINK, 562, 586
 - /NOPACKCODE option, LINK, 562, 586
 - /NOPACKF option, LINK, 562, 586
 - /NOPACKFUNCTIONS option, LINK, 562, 586
 - /NQ option, CL, 528–530
 - /NT option, CL, 528–530
 - NUL, CL options, appending to, 497–498
 - Null character defined, 812
 - Null macros, NMAKE, 670
 - Null pointer defined, 812
 - Null strings, user-defined macros, NMAKE, 668
 - Number sign (#)
 - custom builds, 59
 - HELPMMAKE syntax, 712–713
 - inference rules, NMAKE, 681
 - makefile comments, NMAKE, 654
 - substituting for equal sign, CL, 492
 - Tab Set command, CodeView, 423, 470
 - TOOLS.INI file syntax, 652
 - user-defined macros, NMAKE, 669
 - Numbers, predefined expression syntax, 778, 785
 - Numeric arguments, LINK, 576
 - Numeric constants, CodeView expression
 - evaluators, 407–408
 - Numeric switches, PWB, 122
 - /NV option, CL, 528–530
- ## O
- O command, CodeView, 422, 445–448
 - /O option
 - CL, 530–531, 539
 - HELPMMAKE, 712, 714
 - /o option, BSCMAKE, 733, 737
 - O3 command, CodeView, 445–447
 - OA command, CodeView, 347, 445–447
 - /Oa option, CL, 530–532
 - OB command, CodeView, 445–447
 - /Ob option, CL, 530, 532
 - .OBJ files defined, 812
 - Object files
 - defined, 812
 - object modules, distinguishing, 698
 - output files, LINK, 563–564
 - overview, 563
 - PWB, 43
 - renaming, CL, 498
 - specifying, LINK, 565
 - Object modules
 - adding, 703
 - copying, 705
 - defined, 698, 812
 - deleting, 704
 - format defined, 563, 812
 - moving, 705
 - replacing, 704–705
 - objfiles field, LINK, 565
 - OC command, CodeView, 445–447
 - /Oc option, CL, 530, 533
 - /Od option, CL, 324, 530, 533
 - /Oe option, CL, 531, 533

- OF command, CodeView, 445–447
- /Of option, CL, 531, 533–534
- /Of- option, CL, 390, 531
- Offset defined, 812
- OFFSET operator, MASM, 414
- /Og option, CL, 531, 533
- OH command, CodeView, 445–447
- /Oi option, CL, 531, 534–535
- OL command, CodeView, 445–447
- /OL option, LINK, 586
- /Ol option, CL, 531, 535–536
- OLD statement, module-definition files, 609, 622
- /OLDOVERLAY option, LINK, 562, 586
- ON command, CodeView, 445–447
- /On option, CL, 531
- /Oo option, CL, 531, 536–537
- /Oo- option, CL, 531, 536–537
- /Op option, CL, 531, 537–538
- Opcodes, frame sorting, CL option, 539
- Open command, PWB, 72
- Open Custom command, PWB, 76
- Open Module command, CodeView, 358–359
- Open Module dialog box, CodeView, 359
- Open Project command, PWB, 74
- Open Source command, CodeView, 358
- Open Source File dialog box, CodeView, 358
- Openfile function, PWB, 152, 191, 289–290
- Opening
 - files, PWB, 72, 141, 191, 289–290
 - Help files
 - Microsoft Advisor, 765–766
 - PWB, 213–214
 - QuickHelp, 769
 - projects
 - automatically, 285–286
 - PWB, 49–50, 195–196
 - source files, CodeView, 358
 - source windows, CodeView, 350
 - windows
 - CodeView, 373
 - PWB, 201, 220, 245–246
- Operating system
 - prompt, DOS Shell command, 359
 - specifying in module-definition files, 615–616
 - tags, TOOLS.INI file, PWB, 132
- Operations line, extending, 702
- Operations, regular expressions, PWB, 94
- Operators
 - flow control, PWB, 112–114
 - functions, using C++ expressions, 412
 - regular expressions, PWB, 91–93, 95
- optimize pragma
 - p-code optimization, 538
 - subexpression optimization, 533
 - unsafe optimizations, CL option, 532
- Optimizing
 - assuming no aliasing, CL option, 531–532
 - common expressions, CL option, 533
 - compiler options, PWB, 52–56
 - debugging considerations, 324
 - entry codes, CL options, 515
 - execution time, CL option, 539
 - exit codes, CL options, 515
 - exit sequence, CL options, 538
 - far calls, LINK, 580–581, 584, 587–588
 - file size, CL option, 538
 - frame sorting, CL option, 539
 - inline expansion control, CL option, 532
 - intrinsic function generation, CL option, 534–535
 - load-time relocation table, LINK, 580
 - loops, CL option, 537
 - maximum optimization, CL, 539
 - p-code, CL option, 533, 538
 - post-code generation, CL option, 536–537
 - register allocation, CL option, 533
 - space, SBPACK, 739–740
 - turning off, CL, 533
- Option button, PWB, 81
- Options
 - BSCMAKE, 732, 736–737
 - CL, 323–324, 488–559
 - CodeView, 338–344, 396, 445–447
 - compiler
 - changing in PWB, 53–56
 - debugging considerations, 323–324
 - CVPACK, 744
 - debug, finding symbols, PWB, 101
 - EXP, 750
 - HELPMAKE
 - decoding, 713–714
 - encoding, 712–713
 - IMPLIB, 747
 - LIB, 701
 - LINK
 - debugging considerations, 323–325
 - described, 575–576
 - new features, 561–562
 - NMAKE
 - macros, 676
 - turning on, 688–689
 - PWB, 141–142
 - RM, 748

- Options (*continued*)
 - SBRPACK, 740
 - UNDEL, 749
 - Options command, CodeView, 422, 445–447
 - options field
 - BSCMAKE, 735
 - NMAKE, 647
 - SBRPACK, 740
 - Options menu
 - CodeView, 368–372
 - PWB, 75
 - /Oq option, CL, 531, 538
 - /Or option, CL, 531, 538
 - Ordering segments, debugging considerations, 322
 - OS command, CodeView, 445–447
 - /Os option, CL, 531, 538
 - OS2INIT.CMD, PWB configuration, 137
 - OS2libs switch, PWB, 310–311
 - /Ot option, CL, 531, 539
 - Output
 - redirecting, CodeView, 476–477
 - viewing, CodeView, 374
 - Output files
 - alternate, setting, 495
 - LINK, 563–566
 - preprocessing, creating, 540
 - Output libraries, LIB, 706–707
 - Output screen defined, 813
 - OV command, CodeView, 445–447
 - /OV option
 - CL, debugging p-code, 390
 - LINK, 562, 587
 - /Ov option, CL, 531, 539
 - /Ov- option, CL, 531, 539
 - Overlaid programs
 - compatibility, MOVE, 603
 - creating
 - LINK, 598–601
 - module-definition files, 619–620
 - MOVE, 598–601
 - interoverlay calls, limiting with LINK, 579
 - linking, 601–602
 - memory allocation, MOVE, 602–603
 - module-definition files, 607–608
 - overlays, 604–605
 - overview, 597
 - segments, 603–604
 - space restrictions, 603
 - specifying, LINK, 570
 - Overlaid Virtual Environment. *See* MOVE
 - Overlay caches, overlaid DOS programs, 598, 602–603
 - Overlay heaps, overlaid DOS programs, 598, 602
 - OVERLAY keyword, module-definition files, 619–620
 - Overlay manager, overlaid DOS programs, 603
 - Overlay number, LINK, 564
 - /OVERLAYINTERRUPT option, LINK, 562, 587
 - Overlays
 - compiling, 599–600
 - defined, 813
 - DOS programs, linking, 566
 - module-definition files, 607–608
 - specifying, MOVE, 604
 - Overloaded functions, using C++ expressions, 411
 - OVL keyword, module-definition files, 619–620
 - /Ow option, CL, 530–532
 - /Ox option, CL, 531, 539
 - /Oz option, CL, 531, 537–538
- ## P
- P command, CodeView, 422, 449, 452–453
 - :p command, HELPMMAKE, 723
 - \p formatting attribute, HELPMMAKE, 719, 721
 - /P option
 - CL, 540
 - CodeView, 396
 - CVPACK, 745
 - EXEHDR, 630
 - LIB, 701–702
 - NMAKE, 649
 - P register, CodeView syntax, 419, 450
 - pack pragma data structure, 554–555
 - Packaged functions
 - creating, CL options, 524
 - defined, 813
 - ordering, module-definition files, 626–627
 - overlaid DOS programs, 599
 - Packaged INCLUDE, ordering, module-definition files, 627
 - /PACKC option, LINK, 587–588
 - /PACKCODE option, LINK, 587–588
 - /PACKD option, LINK, 588–589
 - /PACKDATA option, LINK, 588–589
 - /PACKF option, LINK, 562, 589
 - /PACKFUNCTIONS option, LINK, 562, 589
 - Packing
 - files
 - CVPACK, 743–744
 - SBRPACK, 739–740

Packing (*continued*)

- preventing, BSCMAKE, 732
 - structure members, CL options, 554–555
- Page size, specifying, with LIB, 701–702
- /PAGESIZE option, LIB, 701–702
- \par formatting code, HELPMMAKE, 727
- Paragraphs, setting number, LINK, 577–578
- Parameters defined, 813
- Parent process defined, 813
- Parentheses ()
- balancing, in PWB, 192–193
 - searching, PWB, 91, 93
- .PAS files defined, 813
- PASCAL command macro, NMAKE, 676
- Pascal Compiler, NMAKE macros, 676
- __pascal keyword, CL
- calling conventions, 516–518
 - enabling, 550
- Paste command
- CodeView, 360
 - PWB
 - described, 73
 - predefined macros, 143
- .paste command, HELPMMAKE, 723
- Paste function, PWB
- described, 152, 191–192
 - executing, 108
 - replacing text, 94
- Pasting text
- Microsoft Advisor, 761
 - QuickHelp, 771
- PATH environment variable
- CodeView, installing, 327
 - starting PWB, 67
- Paths
- Curfile predefined macro, PWB, 224
 - defined, 813
 - predefined expression syntax, 778, 780, 786
 - search, NMAKE, 660, 682
 - specifying, 88, 496
- Patterns. *See* Regular expressions
- /PAU option, LINK, 589–590
- Pause command, CodeView, 423, 470
- /PAUSE option, LINK, 589–590
- Pausing, Trace Speed command, CodeView, 369
- Pbal function, PWB, 152, 192–193
- .PCH files defined, 813
- P-code
- debugging, 372, 389–393
 - entry tables, specifying, 521
 - native entry points, removing, 520–521

P-code (*continued*)

- optimizing, CL option, 538
 - quoting, CL option, 533
 - registers, displaying, 355
- Percent sign (%)
- file specifier, NMAKE, 653
 - Filename-Parts Syntax, PWB, 265–266
- Period (.)
- Current Location command, CodeView, 423, 471
 - dot directives, NMAKE, 687
 - inference rules, NMAKE, 681
 - line number specifier, CodeView, 365
 - LINK syntax, 565
 - match character, regular expression syntax, 779
 - wildcard character, regular expression syntax, 778, 781
- /PF option, PWB, 141
- PFLAGS options macro, NMAKE, 676
- Physical segments defined, 813
- PID defined, 813
- PIF files, association with Makefiles, 67
- /PL option, PWB, 141
- \plain formatting code, HELPMMAKE, 727
- Playback macro, PWB, 224
- Plines function, PWB, 152, 193
- Plus sign (+)
- Add command, LIB, 703–704
 - concatenating help files, 772
 - LINK syntax, 565, 567
 - options, NMAKE, 688–689
 - searching, PWB, 92
- /PM option, LINK, 562, 590, 630
- /PMTYPE option
- EXEHDR, 630
 - LINK, 562, 590
- /PN option, PWB, 141
- Pointers
- checking, CL option, 556–557
 - converting global memory handles, 463–464
 - converting local memory handles, 466–467
 - defined, 813
 - expanding and contracting, CodeView, 367–368, 478–479
- .popup command, HELPMMAKE, 723
- Pop-up menu defined, 813
- Port defined, 813
- Port Input command, CodeView, 422, 434–435
- Port Output command, CodeView, 422, 448
- port: option, CodeView, 396
- Postfix operator, CodeView precedence, 406

- Pound sign (#)
 - custom builds, 59
 - HELPMMAKE syntax, 712–713
 - inference rules, NMAKE, 681
 - makefile comments, NMAKE, 654
 - Tab Set command, CodeView, 423, 470
 - TOOLS.INI file syntax, 652
 - user-defined macros, NMAKE, 669
- Power, regular expression syntax, 780, 787
- /PP option, PWB, 141
- Ppage function, PWB, 152, 194
- Ppara function, PWB, 152, 194
- PQ register, CodeView, 419, 450
- Pragmas
 - alloc_text, 599, 626
 - check_pointer, 556–557
 - check_stack, 518–520
 - native_caller, 520–521
 - optimize
 - subexpression optimization, 538
 - unsafe optimizations, 532
 - pack, 554–555
- Precedence
 - defined, 813
 - inference rules, NMAKE, 686–687
 - macros, NMAKE, 680
- .PRECIOUS dot directive, NMAKE, 687
- Predefined expressions syntax
 - non-UNIX, 780
 - UNIX, 778, 785
- Predefined inference rules, NMAKE, 684–685
- Predefined macros, PWB, 142–150, 222–224, 227–263
- Predefined names, removing, CL option, 542–544
- Prefixes
 - context, HELPMMAKE, 729
 - program segments, 323
- PRELOAD keyword, module-definition files, 621
- Preprocessing
 - copying output, CL, 493–494
 - preserving comments, CL, 491
- Preprocessing directives, NMAKE, 688–690–692
- .previous command, HELPMMAKE, 723
- Previous command, PWB
 - described, 76
 - function, 145
- Previous Error command, PWB
 - described, 74
 - predefined macros, 144
- Previous match command, PWB
 - described, 73
 - predefined macros, 144
 - searching, 89
- Print command
 - CodeView, 358–359
 - PWB, 72
- Print dialog box, CodeView, 359
- Print function, PWB, 152, 194–195
- Print Results command, PWB, 77
- Printcmd switch, PWB, 264–266, 291
- Printfile entry, TOOLS.INI file, CodeView, 330, 334
- Printing
 - canceling, _pwbcancelprint macro, 230
 - files
 - CodeView, 359
 - PWB, 194–195
 - specifying program, PWB, 291
- PRIVATELIB keyword, LIBRARY statement, module-definition files, 612–613
- Privileged mode defined, 813
- PRN, CL options, appending to, 497–498
- Procedure call defined, 813
- Procedure defined, 813
- Process defined, 813
- Process Descriptor Block, command field, CodeView, 382
- Process identification number defined, 813
- Program Arguments command, PWB, 74
- Program Item, adding, PWB, 66
- Program segment prefixes, debugging considerations, 323
- Program Step command, CodeView
 - controlling execution, 386
 - described, 422, 449, 452
- Program step defined, 814
- Programmer's Workbench. *See* PWB
- Programs
 - building, 56, 58
 - debugging, preparing for, 321–325
 - overlaid
 - creating, 619–620
 - LINK, 570, 579, 597–601
 - module-definition files, 607–608
 - MOVE, 597–605
- PWB
 - building, 45–46
 - debugging, 29
 - editing, 47–49, 55

Programs (*continued*)

- PWB (*continued*)
 - multimodule, 41–42
 - non-PWB makefiles, 61–63
 - project dependencies, 45, 48
 - running, 46–47
- Project dependencies, PWB, 45
- Project function, PWB, 152, 195–196
- Project menu, PWB
 - described, 74
 - predefined macros, 144
- Project Templates command, PWB, 75
- Projects
 - opening automatically, 285–286
 - PWB
 - adding files, 44, 48, 50
 - closing, 234
 - contents, 43–44
 - creating, 42
 - defined, 41
 - deleting files, 48
 - dependencies, 45, 48
 - editing, 47–49
 - extending, 58–61
 - makefiles, 56, 58
 - menu commands, 74
 - moving files, 49
 - opening, 195–196
 - status files, 138–139
 - using, 49
- Prompt function, PWB, 116–117, 152, 196–197
- Prompts
 - Askexit switch, PWB, 267
 - Asktrn switch, PWB, 267
 - LIB, 698–699
 - LINK, 573–576
- Protected mode
 - defined, 803, 814
 - module-definition files, specifying, 617
 - optimizing entry/exit codes, 515
- PROTMODE statement, module-definition files, 609, 617
- Psearch function, PWB, 87, 107, 153, 197
- Pseudofiles
 - creating, in PWB, 187–188, 245
 - Saveall function, PWB, 209
- Pseudotargets, NMAKE, 658
- PTR operator, debugging assembly language, 414–415

Public names

- overlaid DOS programs, 604
 - restricting length, CL option, 525
- Public symbols, searching, CodeView, 406–407
- PWB
- Browse menu
 - described, 76
 - functions, 145, 200
 - browser database. *See* Browser database
 - command line, 141–142
 - commands
 - choosing, 78–79
 - cursor movement, 154–155
 - executing, 78–82, 142, 170, 219
 - configuration
 - autoloading, 131
 - environment variables, 137
 - overview, 130–131
 - customizing colors, 124–125
 - Edit menu
 - described, 73
 - predefined macros, 143
 - File menu
 - described, 72
 - predefined macros, 142
 - files
 - adding, 48, 50
 - deleting, 48
 - estimating size, 103–104
 - moving, 49
 - functions
 - Assign, 121–124
 - Backtab, 127–128
 - described, 150–221
 - executing, 106–108
 - modifying, 181
 - Prompt, 116–117
 - Set Switch, 123
 - Tab, 127–128
 - Help menu
 - described, 78, 757
 - predefined macros, 146
 - Help
 - copying and pasting, 761, 771
 - getting, 756–765
 - global searches, 766–767
 - keywords, 762
 - managing files, 772–773
 - opening files, 765–766
 - structure, 755
 - HELPMAKE restrictions, 709

PWB (*continued*)

- key assignments, 146–150
- macros
 - changing key assignments, 119–121, 135
 - executing, 106–108, 170
 - flow control statements, 112–114
 - overview, 109
 - recording, 109–112
 - user input statements, 114–117
- makefiles
 - loading, 142
 - opening, 141
- multimodule programs, 45, 49
- Options menu, 75
- options, 141–142
- predefined macros, 142–146, 222–224, 227–263
- programs
 - adding files, 44, 50
 - build process, 56–58
 - creating projects, 42
 - described, 45–46, 53–56
 - editing, 47–49, 55
 - extending projects, 58–61
 - non-PWB makefiles, 61–63
 - overview, 41–42
 - project contents, 43–44
 - project dependencies, 48
 - running, 46–47
- project file list, 43
- Project menu, 74, 144
- prompt
 - Askexit switch, 267
 - Askrtm switch, 267
- quitting, 47, 171, 251
- regular expressions syntax, 303–304
- Run menu
 - adding commands, 125–127
 - described, 74
 - predefined macros, 144
- screen display, 67–68
- Search menu
 - described, 73
 - predefined macros, 144
- searching
 - find command, 87–90
 - mark function, 86
 - overview, 85–86
 - regular expressions, 90–93
- single-module programs, debugging, 29

PWB (*continued*)

- source browser
 - browser database, 61
 - building database, 101
 - call tree, showing, 99–101
 - CL options, 507–508
 - combined database, 106
 - creating database, 97–98
 - described, 200
 - estimating file size, 103–104
 - finding symbols, 98, 101, 103
 - non-PWB project database, 104–106
- starting, 65–67
- status files, 138–139
 - changing, 122, 124
 - Filename-Parts syntax, 265–266
 - Help, 313–315
 - library, 310–312
 - source browser, 309–310
- syntax, 141–142, 265–266
- tabs, 127–130, 132–133
- text, replacing, 93–96
- TOOLS.INI file, 113, 115, 132–136
- undefined macros, 224, 226
- View menu, 770
- Window menu
 - described, 77
 - predefined macros, 145
- PWB Windows command, PWB, 77
- _pwbarrange predefined macro, 222, 227–228
- _pwbboxmode predefined macro, 222, 228
- _pwbbuild predefined macro, 222, 229
- PWBC library switches, 310–312
- _pwbcancelbuild predefined macro, 222, 229–230
- _pwbcancelprint predefined macro, 222, 230
- _pwbcancelsearch predefined macro, 222, 230–231
- _pwbcascade predefined macro, 222, 231–232
- _pwbclear predefined macro, 222, 232
- _pwbclose predefined macro, 222
- _pwbcloseall predefined macro, 222, 232–233
- _pwbclosefile predefined macro, 222, 233
- _pwbcloseproject predefined macro, 222, 233–234
- _pwbcompile predefined macro, 222, 234
- _pwbfile predefined macro, 222
- _pwbgotomatch predefined macro, 222, 235
- Pwbhelp function, 153, 198
- _pwbhelp_again predefined macro, 222, 236
- _pwbhelp_back predefined macro, 222, 237
- _pwbhelp_contents predefined macro, 222, 237–238

_pwbhelp_context predefined macro, 222, 238
 _pwbhelp_general predefined macro, 222, 239
 _pwbhelp_index predefined macro, 222, 239–240
 _pwbhelpn1 predefined macro, 222
 Pwbhelpnext function, 153, 198–199
 _pwbhelpnl predefined macro, 235
 Pwbhelpsearch function, 153, 199
 _pwbhelp_searchres predefined macro, 222, 240
 _pwblinemode predefined macro, 222, 241
 _pwblogsearch predefined macro, 222, 241–242
 _pwbmaximize predefined macro, 223, 242
 _pwbminimize predefined macro, 223, 243–244
 _pwbmove predefined macro, 223, 244
 _pwbnewfile predefined macro, 223, 245
 _pwbnewwindow predefined macro, 223, 245–246
 _pwbnextfile predefined macro, 223, 246
 _pwbnextlogmatch predefined macro, 223, 247
 _pwbnextmatch predefined macro, 223, 247–248
 _pwbnextmsg predefined macro, 223, 248
 _pwbpreviouslogmatch predefined macro, 223,
 248–249
 _pwbpreviousmatch predefined macro, 223,
 249–250
 _pwbprevmsg predefined macro, 223, 250
 _pwbprevwindow predefined macro, 223, 250
 _pwbquit predefined macro, 223, 251
 _pwbrebuild predefined macro, 223, 252
 _pwbrecord predefined macro, 223, 252–253
 _pwbredo predefined macro, 223, 253
 _pwbrepeat predefined macro, 223, 253–254
 _pwbresize predefined macro, 223, 254–255
 _pwbrestore predefined macro, 223, 255
 PWBMAKE.EXE, 731–734
 Pwbrowse1stdef function, 153, 200
 Pwbrowse1stref function, 153, 200
 Pwbrowsealltree function, 153, 200
 Pwbrowseclhier function, 153, 200
 Pwbrowsecltree function, 153, 200
 Pwbrowsefuhier function, 153, 200
 Pwbrowsegotodef function, 153, 200
 Pwbrowsegotoref function, 153, 200
 Pwbrowselistref function, 153, 200
 Pwbrowsenext function, 153, 200
 Pwbrowseoutline function, 153, 200
 Pwbrowsepop function, 153, 200
 Pwbrowseprev function, 153, 200
 Pwbrowseviewrel function, 153, 200
 Pwbrowsewhref function, 153, 200
 _pwbsaveall predefined macro, 223, 255–256
 _pwbsavefile predefined macro, 223, 256
 _pwbsetmsg predefined macro, 223, 257

_pwbshell predefined macro, 223, 257–258
 _pwbstreammode predefined macro, 223, 258
 _pwbtile predefined macro, 223, 258–259
 _pwbundo predefined macro, 223, 259
 _pwbusern predefined macro, 223, 260
 PWBUTILS, PWB Options menu, 75
 _pwbviewbuildresults predefined macro, 223, 261
 _pwbviewsearchresults predefined macro, 223,
 261–262
 Pwbwindow function, 153, 201
 _pwbwindow predefined macro, 223, 262–263
 Pword function, PWB, 153, 201

Q

Q command, CodeView, 423, 449
 /Q option
 EXP, 750
 LINK, 590–591
 NMAKE, 649–650
 /qc option, CL, 540
 QH command, MS-DOS, 768–769
 .QLB files defined, 814
 Qreplace function, PWB, 153, 202
 Question mark (?)
 call tree, PWB, 100
 decorated names, C++, 409
 Display Expression command, CodeView, 424,
 477–478
 filename macros, NMAKE, 672–673
 Quick Watch command, CodeView, 424, 478–479
 SBRPACK syntax, 740
 wildcard character
 HELPMMAKE syntax, 711
 NMAKE, 653–654
 regular expression syntax, 780, 786
 UNDEL, 749
 Quick Compile option, CL, 540
 Quick Watch command, CodeView, 364, 367–368,
 424, 478–479
 Quick Watch dialog box, CodeView
 described, 367–368
 displaying, 478–479
 exploring watch expressions, 327
 QuickHelp
 BSCMAKE option, 736
 CL option, 525
 commands, 770
 copying text, 771
 CVPACK option, 745
 EXEHDR option, 630

QuickHelp (*continued*)

- EXP option, 750
- format
 - defining topics, 716–717
 - described, HELPMMAKE, 716
 - dot commands, 722–724
 - formatting attributes, 718–719, 721
 - global contexts, 719–720
 - linking topics, 717–719
 - local contexts, 720
- Help files, opening and closing, 769
- /HELP option, 768
- HELMMAKE option, 715
- IMPLIB option, 747
- LIB option, 701
- LINK option, 581
- NMAKE option, 648
- pasting text, 771
- QH command, MS-DOS, 768–769
- SBRPACK option, 740
- specifying format, HELPMMAKE, 713
- topics
 - displaying, 769
 - navigation, 770
 - selecting, 763
- UNDEL option, 749
- /QUICKLIBRARY option, LINK, 590–591
- QuickWin CL option, 528
- Quit command, CodeView, 423, 449
- Quitting
 - CodeView, 360
 - PWB, 47, 171, 251
- Quotation marks ("")
 - character strings, 805
 - CodeView syntax, 340
 - LINK syntax, 565
 - long filenames, NMAKE, 654
 - module statement syntax, 610–611
 - Pause command, CodeView, 423, 470
- Quote function, PWB, 153, 203
- Quoted string, predefined expression syntax, 778, 780, 786

R

- R command, CodeView, 423
- :r command, HELPMMAKE, 724
- /R option
 - CodeView, 396
 - EXEHDR, 631
 - EXP, 750

/R option (*continued*)

- NMAKE, 650
- PWB, 142
- RM, 748
- /r option
 - BSCMAKE, 737
 - CL, 562
- Radix
 - changing in CodeView, 444–445
 - CodeView expression evaluators, 407–408
 - defined, 814
- Radix command, CodeView, 422, 444–445
- RAM defined, 814
- Random access memory defined, 814
- rate option, CodeView, 396
- .raw command, HELPMMAKE, 724
- .RC files defined, 814
- RC.HLP file, 771
- RCVCOM option, CodeView, 396
- RCVCOM.EXE file, remote debugging, 395
- RCVWCOM option, CodeView, 396
- Read Only command, PWB, 73
- READONLY keyword, module-definition files, 621
- Readonly switch, PWB, 122, 264–266, 291–292
- READWRITE keyword, module-definition files, 621
- Real mode
 - defined, 814
 - specifying, module-definition files, 617
- REALMODE statement, module-definition files, 609, 617
- Realtabs switch, PWB, 127–128, 264, 292
- Rebuild All command, PWB, 144
- Rebuilding, _pwbrebuild macro, 252
- Record function, PWB, 153, 203–204
- Record On command, PWB
 - described, 73
 - predefined macros, 143
- Record Results, PWB, 77
- Recording macros, PWB, 109–112, 203–204, 252–253
- Recursion macros, NMAKE, 674–675
- Red, color value, 273
- Redirect Input command, CodeView, 340, 475, 477
- Redirect Output command, CodeView, 340, 424, 476–477
- Redirection defined, 814
- Redo command, PWB, 73, 143
- Redraw command, CodeView, 424, 479
- .ref command, HELPMMAKE, 724
- Refresh function, PWB, 153, 204

- Register command, CodeView, 373–374, 391–392, 423
- Register indirection, debugging assembly language, 414
- Register names, CodeView recognition, 400–401, 419
- Register window
 - CodeView
 - described, 327
 - function, 354–355
 - opening, 374
 - overview, 348
 - debugging p-code, 391–392
 - defined, 814
- Registers
 - allocating, CL options, 533
 - calling conventions, CL options, 520
 - changing values, CodeView, 450–452
 - CodeView expressions, 400–401, 419
 - defined, 814
 - display radix, 444–445
 - displaying value, CodeView, 354–355
 - flags, defined, 809
 - math coprocessors, dumping registers, 473–474
- Regular expressions
 - defined, 814
 - finding, CodeView, 361
 - global searches, in Microsoft Advisor, 766
 - matching
 - non-UNIX, 788
 - PWB, 307–308
 - predefined. *See* Predefined expressions
 - replacing text, PWB, 93–96
 - searching for, CodeView, 472–473
 - searching, PWB, 85–86, 90–93
 - syntax
 - CodeView, 779
 - non-UNIX, 780, 786
 - PWB, 92
 - UNIX, 777–781, 785
 - tagged. *See* Tagged expressions
- Relocatable defined, 814
- Relocatable files, LINK, 563
- Relocatable Object-Module Format, LINK, 563
- Relocation table, optimizing, LINK, 580
- Relocations, EXEHDR, 639
- Remote debugging
 - bit rate, 396
 - options, 396
 - overview, 393
 - requirements, 393–395
- Remote debugging (*continued*)
 - starting a session, 397–398
 - syntax, 396
- Remove Custom Project Templates command, PWB, 75
- Removing
 - breakpoints, CodeView, 367
 - invariant code, CL option, 535–536
 - library name, CL option, 553–554
 - p-code entry points, 520–521
 - predefined names, CL option, 542–544
 - status bar, CodeView, 347
- Renaming files
 - executable, CL, 499
 - object, CL, 498
- Repeat command, PWB, 73, 143
- Repeat function, PWB, 153, 205
- Repeat Last Find command, CodeView, 361–362
- Repeat, regular expression syntax, 778–787
- Repeating function actions, in PWB, 281–282
- Replace command
 - LIB, 704–705
 - PWB, 73
- Replace function, PWB, 153, 205–207
- Replacing
 - object modules, 704–705
 - text
 - Mreplace function, PWB, 185
 - Mreplaceall function, PWB, 185–186
 - Qreplace function, PWB, 202
 - Replace function, PWB, 205–207
- .RES files defined, 814
- /RESERERROR option, EXEHDR, 631
- Reserved words, module statements, 611
- Resetting
 - CodeView command, 436–437
 - PWB, 141
- Resident option, CodeView, 396
- Resize function, PWB, 153
- Resizing windows, PWB, 254–255
- Resource Compiler, NMAKE macros, 676
- Resource-compiler source file, PWB, 43
- Response files
 - BSCMAKE, 738
 - defined, 814
 - LIB, 699
 - LINK, 573–575
- Restart command, CodeView, 362, 422, 436–437
- Restart macro, PWB, 224, 227
- Restcur function, PWB, 153, 208

- Restore command
 - CodeView, 373–374
 - PWB
 - described, 77
 - predefined macros, 145
 - Restorelayout switch, PWB, 264, 293
 - Restoring
 - files, UNDEL, 749
 - status bar, CodeView, 347
 - windows, PWB, 255–256
 - Return codes
 - CVPACK, 745
 - defined, 807
 - LINK, 596
 - NMAKE, 696
 - from commands, 662–663
 - ignoring, 649, 687
 - SBRPACK, 741
 - Windows applications, optimizing, 515
 - Windows functions
 - customizing, 515
 - generating, 522–523
 - Return instructions, overlaid DOS programs, 603–604
 - RFLAGS options macro, NMAKE, 676
 - Rich text format, HELPMMAKE
 - described, 716, 725–726
 - encoding, 727
 - formatting codes, 726
 - specifying, 713
 - Right function, PWB, 153, 208
 - RM
 - command line, 748–749
 - options, 748
 - overview, 743, 747–748
 - syntax, 748–749
 - Rmargin switch, PWB, 264, 293–294
 - Root defined, 815
 - Routines
 - defined, 815
 - listing in CodeView, 372–373
 - .RSP files defined, 815
 - RTF. *See* Rich text format
 - Rules, inference, NMAKE
 - commands, 660
 - inferred dependents, 685–686
 - precedence, 686–687
 - search paths, 682
 - syntax, 680–682
 - user-defined, 682–685
 - Run DOS Command, PWB, 74
 - Run menu
 - CodeView, 362–363
 - PWB
 - adding menu items, 125, 127, 304–306
 - custom items, 260
 - described, 74
 - predefined macros, 144
 - Run OS/2 Command command, PWB, 74
 - Running
 - LINK, 572
 - NMAKE, 647
 - programs, PWB, 46–47
 - Run-time error defined, 815
 - Run-time startup code, CL linking options, 528
- ## S
- /S option
 - BSCMAKE, 737
 - CodeView, 338, 341, 343
 - EXEHDR, 631
 - NMAKE, 650
 - Sample programs
 - ANNUITY1.C, 29
 - COUNT, 41–63, 97–103
 - Save All command, PWB, 72, 142–143
 - Save As command, PWB, 72
 - Save command, PWB, 72, 142–143
 - Save Custom Project Template command, PWB, 75
 - Saveall function, PWB, 153, 209
 - Savecur function, PWB, 153, 209
 - Savescreen switch, PWB, 264, 294
 - Saving
 - CodeView environment, 360
 - files
 - Autosave switch, PWB, 269
 - PWB, 72, 209, 255–256, 300
 - macros, PWB, 112
 - marks, PWB, 287
 - .SBR files
 - building browser database, 97–98, 105
 - defined, 815
 - estimating size, 103–104
 - sbrfiles field
 - BSCMAKE, 735
 - SBRPACK, 740
 - SBRPACK
 - command line, 740–741
 - exit codes, 741
 - options, 740
 - overview, 731, 739–740

- SBRPACK (*continued*)
 - suppressing, CL options, 507–508
 - syntax, 740
 - turning off, CL option, 555–556
- Scope
 - defined, 815
 - specifying, searching for symbols, 467–468
- Scope operator (::), CodeView precedence, 406
- Screen exchange
 - CodeView options, 341–343, 371, 445–447
 - defined, 815
- Screen Exchange command, CodeView, 424, 479
- Screen Swap command, CodeView, 368, 371
- Screen, PWB display, 67–68
- Scroll bars
 - CodeView
 - options, 445–447
 - toggling options, 370
 - PWB
 - screen display, 68
 - window styles, 221, 275
- Scrolling
 - defined, 815
 - Mlines function, PWB, 183
 - Plines function, 193
 - switches, PWB, 284
 - Vscroll switch, PWB, 306
- Sdelete function, PWB, 153, 209–210
- Search command
 - CodeView, 423, 472–473
 - QuickHelp, 769
- Search logging, PWB, 178, 241–242
- Search Memory command, CodeView, 443–444
- Search menu
 - CodeView, 361–362
 - PWB
 - described, 73
 - predefined macros, 144
- Search paths
 - inference rules, NMAKE, 682
 - specifying, NMAKE, 660
- Search Results command, PWB
 - described, 77–78
 - predefined macros, 146
- Search Results dialog box, PWB, 89
- Search Results window, PWB
 - clearing, 162
 - described, 261–262
 - Mgrep function, 181–182
 - Nextsearch function, 189–190
- Searchall function, PWB, 153, 210
- Searchdialog switch, PWB, 264, 294
- Searchflush switch, PWB, 264, 295
- Searching
 - backwards, PWB, 186
 - canceling, `_pwbcascade` macro, 161, 230–231
 - CodeView, in, 361–362
 - directories, CL option, 525–526
 - Find command, PWB, 87–90
 - global, Microsoft Advisor, 766–767
 - Help system, in PWB, 199
 - highlighting search strings, PWB, 210
 - library files, LINK, 570, 583–584
 - logging searches in PWB, 178, 241–242, 278
 - mark function, PWB, 86
 - memory, CodeView, 443–444
 - Mgrep function, PWB, 181–182
 - Mgreplist macro, PWB, 226
 - module-definition files, LINK, 571
 - object files, LINK, 565
 - overview, PWB, 85–86
 - regular expressions
 - CodeView, 472–473
 - PWB, 90–93
 - symbol definitions, PWB, 98–103
 - symbols, CodeView, 406–407, 421–422, 467–468
 - text, PWB, 197
- Searchwrap switch, PWB, 264, 295–296
- Section tags, TOOLS.INI file, PWB, 132
- /SEG option, LINK, 591–592
- Segment tables, EXEHDR output, 635–638
- Segmented executable files
 - defined, 815
 - EXEHDR output, 634–636
 - format, 631
 - heap allocation, setting in EXEHDR, 630
 - information, providing in EXEHDR, 631
- Segmented Executable Linker. *See* LINK
- Segmented files
 - adding to, module-definition files, 614
 - creating, LINK, 564
 - packing code segments, LINK, 587–588
- Segments
 - defined, 815
 - defining attributes, module-definition files, 618–620
 - memory models, CL options, 488–490
 - naming, CL option, 528–530
 - ordering
 - debugging considerations, 322
 - LINK, 578, 585
 - overlaid DOS programs, 598, 603–604

- Segments (*continued*)
 - packing code, LINK, 587–588
 - setting number, LINK, 591–592
- /SEGMENTS option, LINK, 591–592
- SEGMENTS statement
 - module-definition files, 609, 619–620
 - overlaid DOS programs, 600–601
 - overlay number, LINK, 564, 598
- Selcur function, PWB, 153, 211
- Select function, PWB, 154, 211
- Select To Anchor command, PWB, 73, 143
- Selected text command, CodeView, 361
- Selecting in PWB
 - selection mode, 211–212, 241, 258, 278
 - text, 211
 - windows, 212
- Selection modes, PWB
 - changing, 211–212, 241, 258, 278
 - setting, 73
- Selmode function, PWB, 154, 211–212
- Selwindow function, PWB, 154, 212
- Semaphores defined, 815
- Semicolon (;)
 - command separator, CodeView, 340, 352–353
 - comments, PWB, 136
 - LINK syntax, 565, 571
 - prompt defaults, LIB, 699
 - terminating commands, LIB, 698
 - TOOLS.INI file syntax, 329–330
- Semicolon (;), TOOLS.INI file syntax, 652
- Separator, custom builds in PWB, 59
- Sequence, NMAKE operations, 692–694
- Sessions
 - defined, 806
 - remote debugging, starting, 397–398
- Set Anchor command, PWB
 - described, 73
 - predefined macros, 143
- Set Breakpoint command, CodeView
 - described, 364–366
 - line numbers, 400
- Set Breakpoint dialog box, CodeView, 365–366
- SET command, environment variables in NMAKE, 678
- Set Line-Display Mode option, CodeView, 339
- Set Mark File command, PWB, 73
- Set Project Templates command, PWB, 75
- Set Record command, PWB, 73
- Set Runtime Arguments command, CodeView, 362–363
- Set Screen Swapping option, CodeView, 343
- Set Screen-Exchange Method option, CodeView, 341, 445–447
- Set Switch function, PWB, changing settings, 123
- SETARGV.OBJ file, adding, PWB, 50
- Setfile function, PWB, 154, 212–213, 289–290
- Sethelp function, PWB, 154, 213–214, 765–766
- setjmp function, MOVE programs, 604
- SETUP program
 - CodeView, installing, 327–329
 - Help files, installing, 771
- Setwindow function, PWB, 154, 214
- SHARED keyword, module-definition files, 622
- Shell Escape command, CodeView, 423, 468–469
- Shell function, PWB, 154, 214–215
- Shells
 - defined, 815
 - DOS Shell command, 359
- Shortcut keys
 - CodeView, 346–347
 - PWB, 79
- Shortnames switch, PWB, 264, 296
- Showing call tree, PWB, 99–101
- SI register, CodeView syntax, 419, 450
- .SILENT dot directive, NMAKE, 687–688
- SINGLE keyword, module-definition files, 621
- Single precision defined, 815
- Sinsert function, PWB, 154, 215–216
- Size
 - heaps, specifying, 617–618
 - stacks, specifying, 617
- Size command
 - CodeView, 373–374
 - PWB
 - described, 77
 - predefined macros, 145
- /Sl option, CL, 541
- Slash (/)
 - CL syntax, 488
 - command line, NMAKE, 647
 - HELPMMAKE options, 711
 - LINK syntax, 575
 - Search command, CodeView, 361, 423, 472–473
- Slow motion, CodeView execution, 363, 369
- Small memory model defined, 815
- SMARTDRV.SYS defined, 815
- /Sn option, HELPMMAKE, 713
- Snow, suppressing, CodeView option, 341
- Softer switch, PWB, 264, 296–297
- Sorting frames, CL option, 539
- Source 1 command, CodeView, 373–374
- Source 2 command, CodeView, 373–374

- Source browser
 - browser database, PWB
 - building, 101–102, 106
 - case sensitivity, 309
 - combined, 106
 - creating, 97–98
 - described, 731
 - estimating file size, 103–104
 - finding symbols, 103
 - makefiles, 61
 - non-PWB projects, 104–105
 - specifying, 310
 - CL options, 507–508
 - makefiles, PWB, 61
 - menu commands, PWB, 76
 - Pwbrowse functions, 200
 - searching, PWB, 86
 - switches, 309–310
- Source code, displaying, CodeView, 350, 457–460
- .source command, HELPMMAKE, 724
- Source files
 - creating listing, CL, 501
 - decoding, HELPMMAKE, 713–714
 - defined, 815
 - format options, CL, 541
 - HELMMAKE formats
 - minimally formatted ASCII, 728
 - QuickHelp, 716–724
 - rich text format, 725–726
 - loading, CodeView, 359
 - opening, CodeView, 358
 - PWB project file list, 43
 - specifying type, HELPMMAKE, 713
- Source mode defined, 815
- Source window, CodeView
 - arranging display, 327
 - displaying, 457–460
 - function, 350
 - getting Help, 756
 - opening, 374
 - overview, 347–348
 - setting mode, 453–454
- Source1 Window command, CodeView, 368
- Source2 Window command, CodeView, 368
- /Sp option, CL, 541
- SP register, CodeView syntax, 419, 450
- Space
 - allocation, LINK, 577–578
 - inserting, PWB, 215–216
- Space (*continued*)
 - optimizing
 - PWB, 52–54
 - SBRPACK, 739–740
- Spaces
 - CodeView expression evaluators, 405
 - trailing, display mode, 301–302
- Special characters, NMAKE
 - makefiles, 653
 - user-defined macros, 668
- Special macros, NMAKE, 671–672
- Specifiers
 - CodeView Options command, 445–447
 - displaying source code, 457–460
 - memory format
 - dumping memory, 438–439
 - entering data, 440–441
 - viewing memory, 455–457
 - scope, searching for symbols, 467–468
- Specify Interrupt Trapping option, CodeView, 341–342
- Specifying
 - color, PWB display, 271–273
 - decorated names, 790–791
 - entry tables, CL option, 521
 - environment variables, CL, 557–559
 - execution model, CodeView, 333
 - expression evaluators, CodeView, 331
 - file type, HELPMMAKE, 713
 - filename
 - HELMMAKE, 712, 714
 - LINK, 566–567
 - floating-point math package, CL options, 508
 - inline files, NMAKE, 664
 - interrupt trapping, CodeView, 341–342
 - LIB fields, 699–705
 - libraries, LINK, 566–570
 - module-definition files, LINK, 570
 - object files, LINK, 565
 - options, LINK, 575–576
 - page size, with LIB, 701–702
 - paths, CL, 496
 - response files, LINK, 573–575
 - search path, NMAKE, 660
 - symbol handlers, CodeView, 334–335
- Speed
 - execution
 - CodeView, 453
 - PWB switches, 280, 282

- Speed (*continued*)
 - optimizing
 - LINK, 580–581
 - PWB, 54
- Splitting help files, 773
- /Ss option, CL, 541
- SS register, CodeView syntax, 419, 450
- /ST option, LINK, 592
- /St option, CL, 541
- Stack allocation, setting in EXEHDR, 631
- Stack frame defined, 815
- Stack machine, debugging p-code, 389
- /STACK option
 - EXEHDR, 631
 - LINK, 592
- Stack probes defined, 518–520
- Stack trace defined, 816
- Stack Trace command, CodeView, 422, 435–436
- Stacks
 - defined, 815
 - size, specifying, 494, 617
 - stack-probe routines, CL options, 518–520
- STACKSIZE statement, module-definition files, 609, 617
- Standard error defined, 816
- Standard input defined, 816
- Standard library defined, 816
- Standard mode defined, 816
- Standard output defined, 816
- Starting PWB
 - from command line, 65–66
 - from Windows Program Manager, 66
- Startup code
 - CL linking options, 528
 - defined, 816
- Startup files, PWB configuration, 137
- STARTUP.CMD, PWB configuration, 137
- State file, CodeView
 - overview, 344
 - toggling status, 343
- Statefile read entry, TOOLS.INI file, CodeView, 330, 334, 343
- Statements
 - flow control, PWB, 112–114
 - module-definition files, 608–627
 - multiple, debugging, 322
 - overlaid DOS programs, 600–601
 - specifying file type, LINK, 564
 - TOOLS.INI syntax, PWB, 134–135
 - user input, PWB, 114–117
- Static library defined, 816
- Static linking defined, 697–698, 816
- Static Overlay Manager, 586, 600, 604–605
- Status bar
 - defined, 816
 - overview, CodeView, 347
 - showing, CodeView option, 445–447
- Status Bar command, CodeView, 368, 370
- Status files, PWB, 138–139
- __stdcall keyword, enabling, CL options, 550
- Stream Mode command, PWB, 73, 143
- Stream selection mode, setting in PWB, 258
- String literals, CodeView expression evaluators, 408
- Strings
 - CodeView expression evaluators, 408
 - debugging assembly language, 415
 - defined, 816
 - embedding, CL option, 544
 - searching, PWB, 85–90
 - user-defined macros, NMAKE, 668
- Structure member defined, 816
- Structures
 - debugging assembly language, 415
 - defined, 816
 - expanding and contracting, CodeView, 367–368, 478–479
 - nested, expanding and contracting, 479
 - storage allocation, CL options, 554–555
- Stub file defined, 816
- STUB statement, module-definition files, 609, 614
- Subdirectories, copying files to, PWB, 95
- Subexpressions, optimizing, CL option, 533
- Subroutine defined, 816
- .SUFFIXES dot directive, NMAKE
 - clearing, 650
 - described, 688
 - inference rules, 680–682, 685
- Suppress Snow option, CodeView, 341
- Swapping
 - defined, 816
 - screen exchange, CodeView, 341–343, 371, 445–447
- Switches, PWB
 - Boolean switch syntax, 266
 - changing, 122, 124
 - described, 263–308
 - Filename-Parts syntax, 265–266
 - Help, 313–315
 - library, 310–312
 - source browser, 309–310
 - syntax, TOOLS.INI file, 135

Switches, PWB (*continued*)

- tabs, 127–129
- unixre, 91

Switching Window function, PWB, 220

Symbol handler, specifying, CodeView, 334–335

Symbolhandler entry, TOOLS.INI file

- CodeView, 328, 330, 335
- remote debugging, 393–395

Symbolic Debugging Information

- compressing, 324–325
- defined, 323, 806
- LINK, 577
- loading, 342
- memory requirements, 324
- preserving, with CVPACK, 745
- searching, 362

Symbols

- defined, 805–806, 811
- format, CodeView, 408–409
- local, building a database, 732, 736
- PWB, defined, 98–99, 101–103
- .SBR files, PWB, 104
- searching for, CodeView, 421–422, 467–468
- unreferenced, packing files, 739

Syntax. *See specific command or utility*

.SYS files defined, 817

__syscall keyword, enabling, CL options, 550

SYSTEM environment variable defined, 817

System include files, finding symbols, PWB, 101

T

T command, CodeView, 423, 433, 452–453

/T option

- HELPMAKE, 713–714, 722
- LINK, 592–593
- NMAKE, 650
- PWB, 142

/Ta option, CL, 541–542

\tab formatting code

- HELPMAKE, 727

Tab function, PWB, 127–128, 154, 216

Tab Set command, CodeView, 423, 470

Tabalign switch, PWB, 127–129, 264, 297

Tabdisp switch, PWB, 127–128, 264, 297–298

Tables, EXEHDR output, 635–636, 638

Tabs

- HELPMAKE syntax, 711
- hyperlinks, navigating with, 759–761
- module statement syntax, 610

Tabs (*continued*)

PWB

- aligning switches, 297
- handling, 127–129
- line continuation, 136
- preserving, 292
- previous, 159
- setting, 298–299
- width, 282

regular expressions, PWB, 93

setting, CodeView, 470

Tabstops switch, PWB, 122, 127, 264, 298–299

Tagged expressions

- Build:message switch, 784
- justifying, 785
- overview, 782–784
- regular expression syntax, 778–781, 787
- replacing text, PWB, 93–96

Tags, TOOLS.INI file, 132–134, 329–330

Target files, NMAKE

- defined, 646
- dependency lines, 655–658

Targets

- building, NMAKE, 648
- compiling, PWB, 163–164
- function, PWB, 58–61
- makefiles, PWB, 62–63

targets field, NMAKE, 647

/Tc option, CL, 541–542

Tell dialog box, PWB, 216–218

Tell function, PWB

- changing key assignment, 122
- described, 154, 216–218
- executing, 108

TEMP environment variable defined, 817

Temporary files

- defined, 817
- LINK, 595

Terminate-and-stay-resident programs

- defined, 817
- DOS Shell command, 359
- Shell function, PWB, 214–215

Terminating, CodeView execution, 387–388

Ternary operator defined, 817

Text

- Arg function, PWB, 106–108
- copying
 - CodeView commands, 353
 - Microsoft Advisor, 761
 - QuickHelp, 771
- deleting, PWB, 166, 232

- Text (*continued*)
 - editing, menu commands, PWB, 73
 - finding, PWB, 91–93
 - formatting, HELPMMAKE topics, 721
 - indenting, PWB, 296
 - pasting
 - Microsoft Advisor, 761
 - QuickHelp, 771
 - replacing
 - PWB, 93–96
 - Qreplace function, PWB, 202
 - Replace function, PWB, 205–207
 - searching, PWB, 85–90, 197
 - selecting, PWB, 211
- Text argument, Arg function, PWB, 106–108
- Text Argument dialog box, PWB
 - default key assignments, 150
 - Lasttext function, 175–176
 - Prompt function, 196–197
- Text box, PWB, 81
- Text files
 - See also* Files
 - defined, 817
- Text strings
 - embedding, CL option, 544
 - searching, PWB, 85–90
- Text switches, PWB, 122
- TH register, CodeView syntax, 419, 450
- Thread defined, 817
- Thread ID defined, 817
- Thread of execution defined, 817
- Threshold data, setting with CL option, 522
- Thunk defined, 817
- Tilde (~), menu command, PWB, 126
- Tile command
 - CodeView, 373–374
 - PWB
 - described, 77
 - predefined macros, 145
- Tilemode switch, PWB, 264, 299–300
- Tiling windows, PWB, 258–259, 299–300
- Time
 - current, PWB, 166
 - optimizing, PWB, 54–55
- Time stamps
 - changing, NMAKE, 650
 - defined, 646, 817
 - displaying, NMAKE, 648
- Timersave switch, PWB, 265, 300
- Tiny memory model defined, 817
- /TINY option, LINK, 592–593
- TL register, CodeView syntax, 419, 450
- TMP environment variable
 - defined, 817
 - starting PWB, 67
- .TMP files defined, 817
- Tmpsav switch, PWB, 265, 300–301
- Toggle State-File Reading option, CodeView, 343
- Toggling defined, 817
- TOOLS.INI file
 - CodeView
 - configuring, 329–330
 - entries, 330–336
 - installing, 328–329
 - remote debugging, 393–395
 - setting options, 344
 - defined, 817
 - NMAKE, 652
 - PWB
 - autoloading extensions, 131
 - comments, 136
 - extension switches, 265
 - filename-extension tags, 132–133
 - Initialize function, 172–173
 - line continuation, 136
 - macros, 113, 115
 - named tags, 133–134
 - operating-system tags, 132
 - sections tags, 132
 - switch syntax, 135
- TOOLS.PRE file, CodeView, installing, 328
- Topic command
 - CodeView, 374–375
 - HELMMAKE, 724
 - PWB, 78, 146
- .topic command, HELPMMAKE, 724
- Topic lists, Microsoft Advisor, 765
- Topic: command, PWB, 757
- Topics, help files, linking, 716–719
- Trace command, CodeView, 386, 423, 452
- Trace speed command, CodeView, 368–369, 423, 433, 453
- Tracepoint defined, 817
- Tracing
 - defined, 817
 - functions, CodeView, 452–453
- Traildisp switch, PWB, 265, 301
- Trailing lines, display mode, in PWB, 301–302
- Trailing spaces, display mode, in PWB, 301–302
- Traillines switch, PWB, 265, 301–302
- Traillinesdisp switch, PWB, 265, 302
- Trailspace switch, PWB, 265, 302

Translating

- source code
 - to assembly code, CL, 501–503, 505
 - to machine-code, CL, 502–503, 505
- white space, PWB, 128

Transport entry, TOOLS.INI file

- CodeView, 328, 330, 335–336
- remote debugging, 393–395

Transport layer, specifying, CodeView, 335–336

Trapping, interrupting, CodeView, 341–342

Truncated files, building a database, 733, 735

TSF option, CodeView, 338, 343

TSR

- See also* Terminate-and-stay-resident programs defined, 817

Tutorial, PWB, 7

Twips defined, 726

/Tx option, CL, 541–542

.TXT files defined, 817

Type casting defined, 817

U

U command, CodeView, 423, 453–454

:u command, HELPMMAKE, 724

\u formatting attribute, HELPMMAKE, 721

/U option, CL, 542–544

/u option, CL, 542–544

\ul formatting code, HELPMMAKE, 727

Unary operators

- CodeView precedence, 406
- defined, 818
- preprocessing directives, NMAKE, 690–691

Unassemble command, CodeView, 423, 453–454

Unassembling

- defined, 806, 818
- p-code, 392

Unassigned function, PWB, 154, 218

!UNDEF preprocessing directive, NMAKE, 680, 690

Undefined macros

- NMAKE, 670
- PWB, 224

UNDEL

- command line, 749
- options, 749
- overview, 743, 747–748
- syntax, 749

Undelcount switch, PWB, 265, 303

Underlining, HELPMMAKE code, 726

Underscore (_)

- macros, NMAKE, 668
- regular expressions, PWB, 93
- symbol format, CodeView, 409

Undo command

- CodeView, 360
- PWB
 - described, 73
 - predefined macros, 143

Undo function, PWB, 154, 218

Undocount switch, PWB, 122, 265, 303

UNIX

- predefined expression syntax, 778, 785
- regular expression syntax, 303–304, 777–778, 781

Unixre switch

- PWB, 264, 303–304
- regular expression syntax, 91, 777

Unresolved external defined, 818

Unsigned numbers, predefined expression syntax, 778, 780, 786

Up function, PWB, 154, 219

Use 8514 Displays option, CodeView, 339

Use Black-and-White Display option, CodeView, 339

USE command, CodeView, 423, 454–455

Use Language command, CodeView, 423, 454–455

Use Two Displays option, CodeView, 338

Use VGA Displays option, CodeView, 339

User input statements, 114–117

User switch, PWB, 265–266, 304–306

Usercmd function, PWB, 154, 219

User-defined inference rules, writing, NMAKE, 682–684

User-defined macros, NMAKE

- creating, 668
- special characters, 668
- where to define, 669–670

User-defined type defined, 818

UTILERR.HLP file, 771

Utilities extension, PWB Options menu, 75

UTILS.HLP file, 771, 773

V

\v formatting attribute, HELPMMAKE, 718, 721

\v formatting code, HELPMMAKE, 727

/V option

- CL, 544
- EXEHDR, 631, 635–639
- HELMMAKE, 715
- NMAKE, 650, 679

- /v option, BSCMAKE, 737
 - Values, entering, CodeView, 440–441
 - Variables
 - addresses, debugging assembly code, 414
 - defined, 818
 - editing, CodeView, 351
 - environment. *See* Environment variables
 - local, CodeView, 354
 - .SBR files, PWB, 104
 - scope, CodeView, 445–447
 - VCPI server
 - See also* Virtual Control Program Interface server
 - defined, 818
 - /VERBOSE option, EXEHDR, 631, 635–639
 - Verbose output
 - BSCMAKE, 737
 - DOS header information, 637
 - HELPMMAKE option, 715
 - OS/2 header information, 637–638
 - Vertical Scrollbars command, CodeView, 368, 370
 - VGA
 - defined, 818
 - specifying, CodeView, 339
 - Video graphics adapter defined, 818
 - View Back command, QuickHelp, 770
 - View History command, QuickHelp, 770
 - View Last command, QuickHelp, 770
 - View Memory command, CodeView, 423, 436–437, 455–457
 - View menu, PWB, 770
 - View Next command, QuickHelp, 770
 - View Output command, CodeView, 373–374
 - View References command, PWB, searching, 86
 - View Relationship command, PWB, 76, 145
 - View Source Command, CodeView, 400, 423, 457–460
 - Virtual Control Program Interface server, 336
 - Virtual memory
 - browser database, 733–734
 - defined, 818
 - VM command, CodeView, 423, 436–437, 455–457
 - VS command, CodeView, 400, 423, 457–460
 - Vscroll switch, PWB, 265, 306
 - V-tables, naming, CL option, 528–530
- W**
- /W option
 - CL, 544–545
 - HELPMMAKE, 713, 721
 - LINK, 593
 - W? command, CodeView, 423, 460
 - /WARNFIXUP option, LINK, 593
 - Warning level, setting, CL options, 544–545
 - Watch command, CodeView, 373–374
 - Watch expressions
 - adding, 364, 460
 - deleting, 461
 - listing, 365, 465–466
 - saving, 344
 - setting, 326–327
 - Watch window, CodeView
 - exploring watch expressions, 327
 - function, 351
 - opening, 374
 - overview, 348
 - Watchpoint defined, 818
 - WC command, CodeView, 423, 461
 - WDG command, CodeView, 377, 382, 387–388, 423, 462
 - WDL command, CodeView, 377, 383, 423, 462
 - WDM command, CodeView, 383, 423, 464
 - WGH command, CodeView, 377, 386, 423, 463–464
 - Which Reference command, PWB
 - described, 76
 - function, 145
 - White space
 - converting, PWB, 276–277
 - predefined expression syntax, 778, 780, 785
 - searching, PWB, 91, 93
 - tab switches, PWB, 127–128
 - translating, PWB, 292
 - White, color value, 273
 - Width switch, PWB, 265, 306
 - Width, HELPMMAKE text, 713
 - Wildcards
 - defined, 818
 - HELPMMAKE syntax, 711
 - listing files, PWB, 94
 - makefile names, NMAKE, 653–654
 - regular expression syntax, 778, 780–781, 786
 - SBRPACK, 740
 - SETARGV.OJB files, PWB, 50
 - Windlllibs switch, PWB, 310–311
 - Window function, PWB, 154, 220
 - Window menu, PWB, 77, 145
 - Windows
 - CodeView
 - 8087 window, 355–356
 - Command window, 351–353, 417
 - Help window, 357–358

Windows (*continued*)

- CodeView (*continued*)
 - Local window, 354
 - Memory windows, 356–357
 - navigation, 349
 - opening, 373
 - overview, 347–348
 - Register window, 354–355
 - Source windows, 350
 - Watch window, 351
- debugging, 377–382
- File Manager, starting PWB, 67
- functions, entry/exit codes, customizing, 515
- Program Manager
 - Help, getting, 768–769
 - starting in PWB, 66
- PWB
 - activating, 262–263
 - cascade arrangement, 231
 - closing, 232–233
 - default key assignments, 150
 - maximizing, 179–180, 242
 - minimizing, 182–183, 243–244
 - moving, 183–184, 244
 - opening, 201, 220, 245–246
 - resizing, 254–255
 - restoring, 255–256
 - screen display, 67–68
 - selecting, 212
 - styles, in PWB, 221, 275
 - tiling, 258–259, 299–300
- Windows Dereference Global Handle command, CodeView, 377, 386, 423
- Windows Dereference Local Handle command, CodeView, 377, 386, 423
- Windows Display Global Heap command, CodeView, 377, 382, 423, 462–463
- Windows Display Local Heap command, CodeView, 377, 383, 423, 462
- Windows Display Modules command, CodeView, 377, 383, 423, 463
- Windows Kill Application command, CodeView, 385, 387–388, 423, 464–465
- Windows menu, CodeView, 373
- Winlibs switch, PWB, 310–311
- Winstyle function, PWB, 154, 221
- WKA command, CodeView, 377, 385, 464–465
- WL command, CodeView, 423, 465–466
- WLH command, CodeView, 377, 386, 423, 466–467
- WO operator, CodeView, 405, 414–415

Word processor

- formatting HELPMMAKE text, 726
- rich text format, HELPMMAKE, 725–726
- Word switch, PWB, 265, 307–308
- Word wrapping, PWB switches, 265, 293, 308
- Words
 - English, regular expression syntax, 778, 780, 785
 - finding in CodeView, 361
- Wordwrap switch, PWB, 265, 308

X

- X command, CodeView, 423, 467–468
- :x command, HELPMMAKE, 722
- /X option
 - CL, 545
 - NMAKE, 650
- /x option, LINK, 562, 591, 594
- XMS
 - See also* Extended memory manager
 - defined, 818
 - Keepmem switch, PWB, 285
- XMS server defined, 818

Y

- :y command, HELPMMAKE, 722
- \Yc option, CL, 546–548
- \Yd option, CL, 546, 548–549
- \Yu option, CL, 546, 549–550
- Yellow, color value, 273

Z

- :z command, HELPMMAKE, 722
- /Za option, CL, 550–552
- /Zc option, CL, 552
- /Zd option, CL, 324, 409, 553
- /Ze option, CL, 550–552
- /Zf option, CL, 555
- /Zg option, CL, 552–553
- /Zi option, CL, 324, 409, 553
- /Zl option, CL, 553–554
- /Zn option
 - BSCMAKE, 732
 - CL, 555–556
- /Zp option, CL, 554–555
- /Zr option, CL, 556–557
- /Zs option
 - BSCMAKE, 732
 - CL, 557

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399

Microsoft®