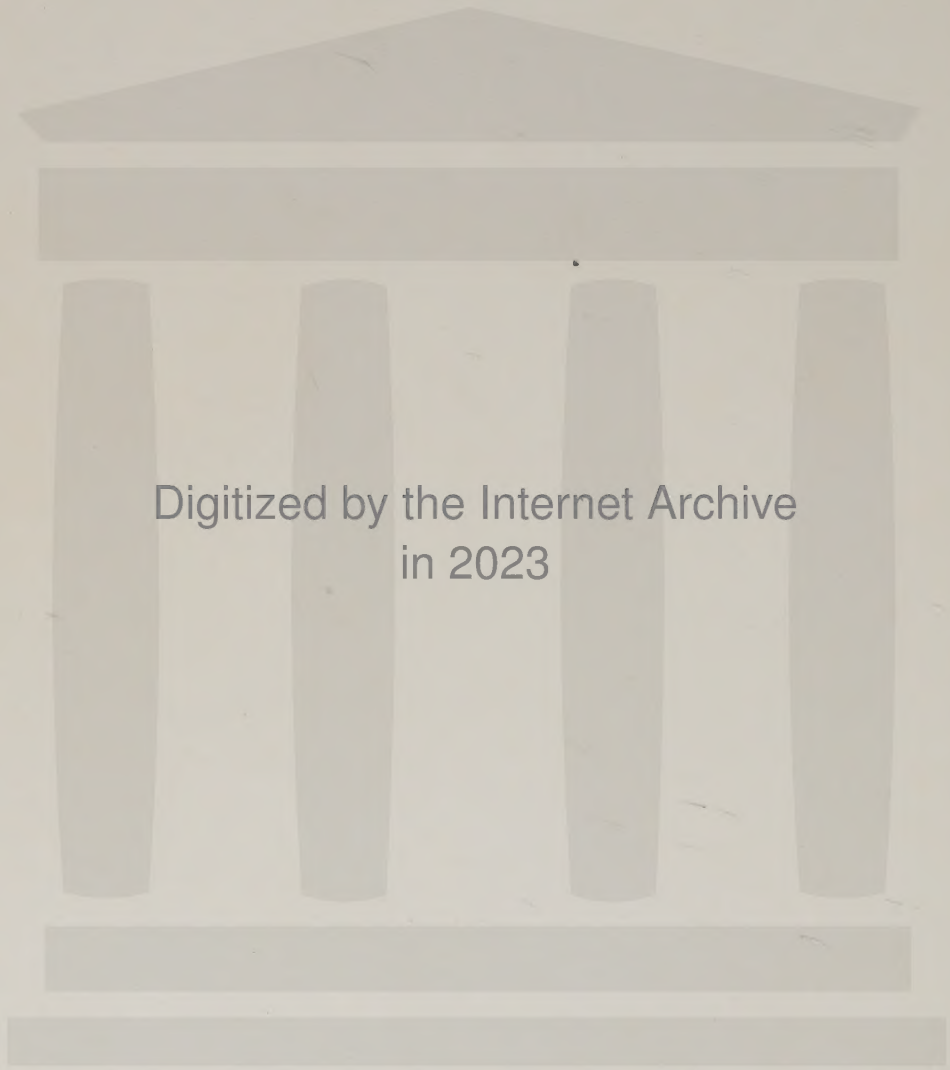
Microsoft® QuickBASIC 4.0

Learning and Using Microsoft QuickBASIC



Microsoft®



Digitized by the Internet Archive
in 2023

https://archive.org/details/microsoftquickba0000auth

TABLE OF CONTENTS

Introduction

System Requirements xxiii
QuickBASIC Documentation xxiii
Using This Manual xxiv
Typographic Conventions xxv
Resources for Learning xxvii
Books about BASIC xxvii
Books about Assembly Language xxviii
Books about DOS xxviii
Requesting Assistance xxviii

Part 1 ◇ Getting Started

1 Installing QuickBASIC 5
1.1 Backing Up Your Disks 7
1.2 Disk Contents 7
1.3 Installing QuickBASIC: Hard-Disk Setup 9
1.4 Installing QuickBASIC: Floppy-Disk Setup 10
1.5 If You Have a Mouse 12
1.6 If You Have a Math Coprocessor 12

1.7 Setting DOS Environment Variables....13
1.7.1 The PATH Environment Variable....14
1.7.2 The LIB Environment Variable....15
1.7.3 The NO87 Environment Variable....15

2 Your First QuickBASIC Session....17

2.1 If You Are a BASICA Programmer....19
2.1.1 Starting QuickBASIC....21
2.1.2 Loading a Program....21
2.1.3 Running Your Program....24
2.1.4 A Comparison of Interpreting, Compiling, and Running in QuickBASIC....26
2.1.5 Creating an Executable Program on Disk....28
2.1.6 Exiting QuickBASIC....29
2.1.7 Running a Program from DOS....29
2.1.8 How QuickBASIC Differs from BASICA....29
2.1.8.1 Environment Enhancements....30
2.1.8.2 New Language Features....31
2.1.9 What's Next....32

2.2 If You Have Used QuickBASIC Version 2.0 or 3.0....33
2.2.1 Starting QuickBASIC....34
2.2.2 Loading a QuickBASIC Program....34
2.2.3 Where Are Your Subprograms?....36
2.2.4 Running a QuickBASIC Program....38
2.2.5 Creating an Executable Program....38
2.2.6 Running a Program from DOS....39
2.2.7 Improvements in This Version of QuickBASIC....40
2.2.7.1 Environment Enhancements....40
2.2.7.2 New Language Features....41
2.2.8 What's Next....42

2.3 If You Have Used a BASIC Compiler....42
2.3.1 Compiling and Linking from the Command Line....43
2.3.2 What's Next....44

2.4 Vital Information for All QuickBASIC Users 45
2.4.1 Why QuickBASIC Is Better Than Other BASIC Products 45
2.4.1.1 The Development Environment You've Always Wanted 45
2.4.1.2 New Language Features 47
2.4.2 The qb Command 49
2.4.3 Customizing the QuickBASIC Screen 51
2.4.4 The QB.INI File 52
2.4.5 Getting Help from QuickBASIC 52
2.4.6 QuickBASIC File-Naming Conventions 55
2.4.6.1 Path Names 55
2.4.6.2 Uppercase and Lowercase Letters 55
2.4.6.3 File-Name Extensions 56
2.4.6.4 Specifying File Names 57

Part 2 ◇ The QuickBASIC Program Development Environment

3 Getting Around in the QuickBASIC Environment ... 63
3.1 The QuickBASIC Screen 66
3.2 Opening Menus and Choosing Commands 68
3.2.1 Keyboard Technique 69
3.2.2 Mouse Technique 70
3.2.3 Using Shortcut Keys 71
3.3 Using Dialog Boxes 72
3.4 Using Windows 76
3.4.1 Window Types 76
3.4.2 Splitting the View Window 77
3.4.3 Changing the Active Window 78

3.4.4 Changing Window Size 78
3.4.5 Scrolling in the Active Window 79
3.4.6 The Immediate Window 79

3.5 Returning Temporarily to DOS (DOS Shell) 80

3.6 Exiting QuickBASIC (Exit) 81

4 Managing Source Files 83

4.1 Classifying Files in QuickBASIC 86

4.2 Programs 88
4.2.1 Loading and Listing Programs (Open Program) 88
4.2.1.1 Specifying a File 90
4.2.1.2 Listing Directory Contents 90
4.2.2 Saving a Program (Save, Save All, Save As) 92

4.3 Modules 94
4.3.1 Creating and Loading Modules (Load File) 95
4.3.2 Displaying Multiple Modules 98
4.3.3 Saving Multiple Modules (Save All) 99

4.4 Changing the Main Module (Set Main Module) 99
4.4.1 The .MAK File 100
4.4.2 Removing Modules from a Program (Unload File) 101

4.5 Include Files 101
4.5.1 Creating and Loading Include Files (Create File) 102
4.5.2 Viewing and Editing Include Files (Include File, Included Lines) 103
4.5.3 Nesting Include Files 105

4.6 Documents 105

4.7 Merging the Contents of Two Files (Merge) 107

4.8 Printing Files (Print) 109

5 Editing 111
5.1 Entering Text 113
5.2 Smart-Editor Features 113
5.2.1 When Is the Smart Editor On? 114
5.2.2 Automatic Syntax Checking 114
5.2.3 Turning Off Syntax Checking 115
5.2.4 Automatic Formatting 116
5.3 Inserting and Overtyping 116
5.4 Selecting Text 117
5.5 Deleting and Inserting Text 117
5.5.1 Using the Clipboard 117
5.5.2 The Clear, Cut, Copy, and Paste Commands 118
5.6 Undoing Your Last Edit 119
5.7 Moving and Copying Text 120
5.8 Searching and Replacing 121
5.8.1 Finding Text 122
5.8.1.1 Find 123
5.8.1.2 Selected Text 125
5.8.1.3 Repeat Last Find 125
5.8.1.4 Label 125
5.8.2 Replacing Selected Text (Change) 126
5.9 Using Place Markers in Text 129
5.10 Entering Special Characters 129
5.10.1 Entering High-Order ASCII Characters 129
5.10.2 Entering Control-Sequence Characters 130
5.11 Indenting 131
5.12 Joining Lines 132
5.13 Copying Text from Other Files 132
5.13.1 Copying an Entire File 132
5.13.2 Copying Part of a File 133
5.14 Summary of Editing Commands 134

6 Creating and Running QuickBASIC Programs 137

6.1 Writing a Program 139
- 6.1.1 Creating a Main Module 140
- 6.1.2 Entering Your Program's Statements 140
- 6.1.3 Translating and Running in Memory 143
- 6.1.4 Returning to the Output Screen (Output Screen) 143
- 6.1.5 Testing Ideas in the Immediate Window 143
  - 6.1.5.1 Characteristics of the Immediate Window 144
  - 6.1.5.2 Statements Allowed in the Immediate Window 144
  - 6.1.5.3 Doing Calculations 145
  - 6.1.5.4 Testing Screen Output 145
  - 6.1.5.5 Invoking Procedures 146
  - 6.1.5.6 Changing Variable Values in a Running Program 146
  - 6.1.5.7 Simulating Run-Time Errors 147
- 6.1.6 Programs with Command-Line Arguments (Modify COMMAND$) 148

6.2 Creating Executable Files from within QuickBASIC (Make EXE File) 148
- 6.2.1 Quick Libraries and Executable Files 151
- 6.2.2 Run-Time-Module Considerations 152
  - 6.2.2.1 Using the Run-Time Module 152
  - 6.2.2.2 Creating Stand-Alone Programs 152
- 6.2.3 Run-Time Error Checking in Executable Files 153
- 6.2.4 Floating-Point Arithmetic in Executable Files 153

6.3 Using SUB and FUNCTION Procedures in Your Programs 155
- 6.3.1 Creating a SUB or FUNCTION Procedure (New SUB, New FUNCTION) 155
- 6.3.2 Saving and Naming Procedures 157

6.3.3 Viewing and Editing Modules and Procedures (SUBs) 158
6.3.4 Automatic Procedure Declarations 159
6.4 Combining Modules into a Program 160

7 Debugging While You Program 163
7.1 Debugging with QuickBASIC 165
7.2 Preventing Bugs 166
7.3 QuickBASIC's Debugging Features 168
7.3.1 Debugging Terms and Concepts 168
7.3.2 The Debug Menu 170
7.3.3 Tracing Program Execution: Beyond TRON and TROFF 171
7.3.3.1 Single Step (F8) 171
7.3.3.2 Procedure Step (F10) 171
7.3.3.3 Animated Trace (Trace On) 171
7.3.3.4 History On 172
7.3.3.5 History Back (SHIFT+F8) 172
7.3.3.6 History Forward (SHIFT+F10) 172
7.3.4 Breakpoints, Watchpoints, and Watch Expressions 172
7.3.4.1 The Watch Window 173
7.3.4.2 Breakpoints (F9) 175
7.3.4.3 Watchpoints 175
7.3.4.4 Watch Expressions: Beyond PRINT 176
7.3.5 Controlling Execution 178
7.3.5.1 Continue (F5) 178
7.3.5.2 Executing to the Cursor (F7) 179
7.3.5.3 Set Next Statement 179
7.3.5.4 Next Statement 180
7.3.5.5 Restart 180
7.4 Advanced Debugging 180
7.4.1 The Calls Menu 180
7.4.2 Compatibility with the CodeView® Debugger 182

Part 3 ◇ Libraries and Tools

8 Quick Libraries 187

8.1 Types of Libraries 189

8.2 Advantages of Quick Libraries 190

8.3 Creating a Quick Library 191
- 8.3.1 Files Needed to Create a Quick Library 192
- 8.3.2 Making a Quick Library 192
- 8.3.3 Making a Quick Library from within the Environment (Make Library) 193

8.4 Using Quick Libraries 195
- 8.4.1 Loading a Quick Library 196
- 8.4.2 Floating-Point Arithmetic in Quick Libraries 197
- 8.4.3 Viewing the Contents of a Quick Library 197

8.5 The Supplied Library (QB.QLB) 197

8.6 The .QLB File-Name Extension 198

8.7 Making a Library from the Command Line 198

8.8 Using Routines from Other Languages in a Quick Library 199

8.9 Memory Considerations with Quick Libraries 200

8.10 Making Compact Executable Files 201

9 Compiling and Linking from DOS 203

9.1 BC, LINK, and LIB 205

9.2 Advantages of Using BC 206

9.3 The Compiling and Linking Process 206

9.4 Compiling with the bc Command 207
9.4.1 Specifying File Names 208
9.4.1.1 Uppercase and Lowercase Letters 208
9.4.1.2 File-Name Extensions 209
9.4.1.3 Path Names 209
9.4.2 Using bc Command Options 210
9.5 Linking 212
9.5.1 Defaults for the Linker 214
9.5.2 Specifying Files to the Linker 215
9.5.3 Specifying Libraries to the Linker 216
9.5.4 Memory Requirements for the Linker 217
9.5.5 Using Linker Options 218
9.5.5.1 Viewing the Options List (/HE) 219
9.5.5.2 Pausing during Linking (/PAU) 219
9.5.5.3 Displaying Linker-Process Information (/I) 220
9.5.5.4 Preventing Linker Prompting (/B) 220
9.5.5.5 Creating Quick Libraries (/Q) 221
9.5.5.6 Packing Executable Files (/E) 221
9.5.5.7 Disabling Segment Packing (/NOP) 222
9.5.5.8 Ignoring the Usual BASIC Libraries (/NOD) 222
9.5.5.9 Setting Maximum Number of Segments (/SE) 222
9.5.5.10 Creating a Map File (/M) 223
9.5.5.11 Including Line Numbers in a Map File (/LI) 225
9.5.5.12 Packing Contiguous Segments (/PAC) 225
9.5.5.13 Using the CodeView Debugger (/CO) 226
9.5.5.14 Distinguishing Case (/NOI) 226
9.5.6 Other link Command-Line Options 226
9.6 Managing Stand-Alone Libraries: LIB 227
9.6.1 Running LIB 228
9.6.2 Usual Responses for LIB 230
9.6.3 Command Symbols 230
9.6.4 Cross-Reference-Listing Files 233
9.6.5 Setting the Library-Page Size 233

Appendixes

A Converting BASICA Programs to QuickBASIC 237

A.1 Source-File Format 239

A.2 Cassette BASIC 240

A.3 Statements and Functions Prohibited by QuickBASIC 240

A.4 Statements Requiring Modification 240

B Differences from Previous Versions of QuickBASIC 243

B.1 New Features 245

B.1.1 User-Defined Types 247

B.1.2 IEEE Format and Math-Coprocessor Support 247

B.1.2.1 Ranges of IEEE-Format Numbers 248

B.1.2.2 PRINT USING and IEEE-Format Numbers 249

B.1.2.3 Recompiling Old Programs with /mbf 249

B.1.2.4 Converting Files and Programs 250

B.1.3 On-Line Help 252

B.1.4 Long (32-Bit) Integers 252

B.1.5 Fixed-Length Strings 253

B.1.6 Syntax Checking on Entry 253

B.1.7 Binary File I/O 253

B.1.8 FUNCTION Procedures 253

B.1.9 Support for the CodeView Debugger 254

B.1.10 Other-Language Compatibility....254
B.1.11 Multiple Modules in Memory....254
B.1.12 ProKey™, SideKick®, and SuperKey® Compatibility....254
B.1.13 Insert and Overtype Modes....255
B.1.14 WordStar®-Style Keyboard Commands....255
B.1.15 Recursion....255
B.1.16 Error Listings during Separate Compilation....256
B.1.17 Assembly-Language Listings during Separate Compilation....256
B.2 Differences in the Environment....256
B.2.1 Choosing Commands and Options....257
B.2.2 Windows....257
B.2.3 New Menus....257
B.2.4 Menu Commands....258
B.2.5 Editing-Key Changes....261
B.3 Differences in Compiling and Debugging....262
B.3.1 Command-Line Differences....262
B.3.2 Separate Compilation Differences....263
B.3.3 User Libraries and BUILDLIB....263
B.3.4 Restrictions on Include Files....264
B.3.5 Debugging....264
B.4 Changes to the BASIC Language....266
B.5 File Compatibility....272

C Calling C and Assembly-Language Routines....273
C.1 Organizing Mixed-Language Programs....275
C.2 Mixed-Language Programming Elements....276
C.2.1 Making Mixed-Language Calls....277
C.2.2 Naming-Convention Requirement....280
C.2.3 Calling-Convention Requirement....282
C.2.4 Parameter-Passing Requirement....283

C.2.5 Compiling and Linking 284
C.2.5.1 Compiling with Proper Memory Models 284
C.2.5.2 Linking with Language Libraries 285

C.3 BASIC Calls to C 286
C.3.1 The BASIC Interface to Other Languages 286
C.3.1.1 The DECLARE Statement 286
C.3.1.2 Using ALIAS 287
C.3.1.3 Using the Parameter List 288
C.3.2 Alternative BASIC Interfaces 289
C.3.3 BASIC Calls to C 290
C.3.4 Restrictions on Calls from BASIC 293
C.3.4.1 Memory Allocation 293
C.3.4.2 Incompatible Functions 294
C.3.4.3 Allocating String Space 295
C.3.4.4 Performing I/O on BASIC Files 296
C.3.4.5 Events and Errors 296

C.4 C Calls to BASIC 297
C.4.1 The C Interface to Other Languages 297
C.4.2 Calls from C to BASIC 299

C.5 Passing Data by Reference or Value 302
C.5.1 BASIC Arguments 302
C.5.2 C Arguments 303

C.6 Numeric and String Data in Mixed-Language Programming 304
C.6.1 Integer and Real Numbers 305
C.6.2 Strings 306
C.6.2.1 String Formats 306
C.6.2.2 Passing BASIC Strings 308
C.6.2.3 Passing BASIC Strings to C 309
C.6.2.4 Passing C Strings 310

C.7 Special Data Types 311
C.7.1 Arrays 311
C.7.1.1 Passing Arrays from BASIC 311
C.7.1.2 Array Indexing and Declaration 313
C.7.2 Storage of Structures and User-Defined Types 315

C.8 Pointers, Address Variables, and Common Blocks 316
C.8.1 Common Blocks 316
C.8.2 Using a Varying Number of Parameters 317
C.9 The B_OnExit Routine 319
C.10 Assembly-Language-to-BASIC Interface 321
C.10.1 Writing the Assembly-Language Procedure 321
C.10.1.1 Setting Up the Procedure 322
C.10.1.2 Entering the Procedure 323
C.10.1.3 Allocating Local Data (Optional) 323
C.10.1.4 Preserving Register Values 324
C.10.1.5 Accessing Parameters 325
C.10.1.6 Returning a Value (Optional) 327
C.10.1.7 Numeric Return Values Other Than Two- and Four-Byte Integers 327
C.10.1.8 Exiting the Procedure 329
C.10.2 Calls from BASIC 330
C.10.3 Using CDECL in Calls from BASIC 332
C.10.4 The Microsoft® Segment Model 333

D Command Summary 335
D.1 Invocation Options 337
D.2 Menu Commands and Shortcut Keys 339
D.3 Debugging Commands 344
D.4 Editing Commands 345
D.5 Window Commands 348

Glossary 351

Index 367

Figures

Figure 2.1 The File Menu 22
Figure 2.2 Open Program Dialog Box 23
Figure 2.3 The Run Menu 25
Figure 2.4 Make EXE File Dialog Box 28
Figure 2.5 QuickBASIC Screen with DEMO2.BAS Loaded 35
Figure 2.6 SUBs Dialog Box 37
Figure 2.7 Make EXE File Dialog Box 39
Figure 2.8 Options Dialog Box 51
Figure 2.9 Help for Menu Commands 53
Figure 2.10 General Help 54
Figure 3.1 QuickBASIC Screen 66
Figure 3.2 The File Menu 68
Figure 3.3 Shortcut Keys 72
Figure 3.4 Load File Dialog Box 73
Figure 3.5 Options Dialog Box 74
Figure 3.6 Screen with Four Windows Open 77
Figure 4.1 The File Menu 86
Figure 4.2 Open Program Dialog Box 89
Figure 4.3 Save As Dialog Box 93
Figure 4.4 Load File Dialog Box 95
Figure 4.5 Create File Dialog Box 96
Figure 4.6 SUBs Dialog Box 98
Figure 4.7 Set Main Module Dialog Box 100
Figure 4.8 Include File Commands on View Menu 103
Figure 4.9 Merge Dialog Box 107
Figure 4.10 Print Dialog Box 109

Figure 5.1 Syntax-Checking Toggle on Edit Menu.....115
Figure 5.2 The Search Menu.....121
Figure 5.3 Find Dialog Box.....123
Figure 5.4 Change Dialog Box.....126
Figure 5.5 Change, Skip, and Cancel Command Buttons.....128
Figure 6.1 Error Message.....141
Figure 6.2 Context-Sensitive Help for CALL Statement.....142
Figure 6.3 Make EXE File Dialog Box.....150
Figure 6.4 Edit Menu.....156
Figure 6.5 SUBs Dialog Box.....158
Figure 7.1 The Debug Menu.....170
Figure 7.2 Debugging Features.....173
Figure 7.3 Sequence of Procedures on Calls Menu.....181
Figure 8.1 Make Library Dialog Box.....194
Figure C.1 Mixed-Language Call.....278
Figure C.2 Naming Convention.....281
Figure C.3 C Call to BASIC.....300
Figure C.4 BASIC String-Descriptor Format.....307
Figure C.5 C String Format.....307
Figure C.6 Structure and Record Storage.....315
Figure C.7 The Stack Frame.....325
Figure C.8 BASIC Return Values.....328
Figure C.9 BASIC Stack Frame.....331

Tables

Table 1.1 Files You Need When Using QuickBASIC 8
Table 1.2 Disk Setup for Two-Floppy-Drive System.....10
Table 3.1 Mouse Commands.....71
Table 3.2 Scrolling.....79
Table 5.1 Editing Commands 134
Table 9.1 Input to the bc Command.....208
Table 9.2 Input to the link Command 213
Table A.1 Statements Requiring Modification 241
Table B.1 New Features of Microsoft QuickBASIC Version 4.0 246
Table B.2 New Menus in QuickBASIC Version 4.0.....257
Table B.3 New Menu Commands in QuickBASIC Version 4.0 258
Table B.4 Editing-Key Changes 261
Table B.5 qb and bc Options Not Used in QuickBASIC Version 4.0 262
Table B.6 New Options for the qb and bc Commands 263
Table B.7 Debugging-Key Changes 265
Table B.8 Changes to the BASIC Language 266
Table C.1 Language Equivalents for Routine Calls.....279
Table C.2 Parameter-Passing Defaults 284
Table C.3 Equivalent Numeric Data Types 305
Table C.4 Equivalent Array Declarations 314
Table C.5 Default Segments and Types for Medium Memory Model.....333
Table D.1 Invocation Options for the qb Command 337
Table D.2 Invocation Options for the bc Command 338

Table D.3 Menu Commands and Shortcut Keys 339
Table D.4 QuickBASIC Shortcut Keys 342
Table D.5 Keyboard Debugging Commands 344
Table D.6 Editing Commands 345
Table D.7 Window Commands 348

INTRODUCTION

Welcome to Microsoft® QuickBASIC Version 4.0. QuickBASIC combines a total programming environment with the most powerful BASIC language available—everything you need to develop lightning-fast, compact programs quickly. In the QuickBASIC environment, you can mix the editing, running, and debugging of a program in any way you please, because all your programming tools are up and running all the time. Here's why you'll save time and write better programs with QuickBASIC:

- **Instant Execution of Your Code As You Write It**

  QuickBASIC immediately translates each new line of code you write into instructions the computer can execute. There is no separate "compiling" step to slow you down and interrupt your train of thought. So you can run your code any time, no matter at what stage of completion, just by pressing a key. When you are satisfied with the way your program runs, it takes just four keystrokes for QuickBASIC to turn it into a free-standing executable program that you can run from DOS.

- **Instant Help and Syntax Checking**

  Context-sensitive on-line Help provides syntax for every QuickBASIC statement, as well as Help for menu commands, shortcut keys, and American Standard Code for Information Interchange (ASCII) codes. QuickBASIC's "smart editor" automatically checks the syntax of each line as you type it, so you can correct errors as you write your program.

- **Instant Debugging**

  In QuickBASIC, you can debug your code as you write it—there is no separate debugging step. Just suspend your running code, analyze its behavior, edit it, and then continue running it from the point where you stopped. You can trace through programs three different ways and display the changing values of variables in a special window. Multiple breakpoints and watchpoints make it easy to change and test new code, because you can plug it right into your running program.

- **Multiple Modules**

  Large programs are simpler to write and debug if you separate their parts into logically distinct components. QuickBASIC makes multiple-module programs easy to create, because it keeps all your program's modules in memory simultaneously and provides an instant outline view, not only of all the program's modules, but the procedures contained in each module.

- **Multiple Editing Windows**

  QuickBASIC's multiple editing windows allow you to view different parts of your program at the same time. Using multiple windows when tracing through procedures keeps execution logic clearer while you debug your code.

- **Full Graphics Support**

  QuickBASIC fully supports the new IBM® Personal System/2™ (PS/2™) graphics modes on the Video Graphics Array (VGA) and Multicolor Graphics Array (MCGA), as well as the Color Graphics Adapter (CGA), Enhanced Graphics Adapter (EGA), and monochrome modes. QuickBASIC now supports the Hercules® Graphics Card.

In addition to the programming environment, QuickBASIC extends the BASIC language to include these powerful features:

- QuickBASIC supports binary files, so your programs can create and manipulate files in any format.
- User-defined data types simplify random-access file input/output (I/O) and help you build more complex data structures from numeric variables and strings.
- QuickBASIC subprograms and functions can be recursive (they can call themselves), allowing you to write compact, powerful code.
- Long (32-bit) integers and 8087/80287 coprocessing help you do fast, accurate math.
- Sophisticated control-flow statements such as **SELECT CASE** and block **IF...THEN...ELSE** improve the logical flow of your programs.

System Requirements

Microsoft QuickBASIC requires the following hardware:

- An IBM Personal Computer or "compatible" that runs MS-DOS® or PC-DOS Version 2.1 or later
- At least one floppy-disk drive (two are recommended)
- 320 kilobytes (K) installed memory

QuickBASIC Documentation

Version 4.0 includes expanded documentation to help you get the most out of QuickBASIC. This package has three manuals:

Manual	Content
Learning and Using Microsoft QuickBASIC	Explains how to install and use Microsoft QuickBASIC on your computer. If you are just starting out, you should begin with this manual. In addition to explaining the fundamentals, it tells you how to use the QuickBASIC programming environment's advanced features, such as the smart editor and debugging tools. See the next section, "Using This Manual," for more information.
Programming in BASIC: Selected Topics	Shows how to use QuickBASIC to perform common programming tasks such as creating graphics, managing data-file I/O, and building complex programs with subprograms and functions. Designed around specific, practical topics, this manual includes many programming examples.
BASIC Language Reference	Describes fully every statement in the QuickBASIC programming language. It also discusses the elements of BASIC, such as data types, operators, variables, constants, and expressions. The appendixes include a table of ASCII codes, a list of words reserved in BASIC, and a complete list of all the error messages that you might encounter while programming in QuickBASIC.

Using This Manual

This manual is your central reference for how to use the QuickBASIC program development environment. Here is a summary of the manual's parts:

Part	Content
Part 1, "Getting Started"	Explains how to install and start QuickBASIC Version 4.0, with specific instructions for those who are experienced with BASICA or earlier versions of QuickBASIC.
Part 2, "The QuickBASIC Program Development Environment"	Discusses how to edit, run, and debug programs within the QuickBASIC programming environment. You also learn how to use menus and dialog boxes, and how to manage files.
Part 3, "Libraries and Tools"	Explains how to create, modify, and use Quick libraries, as well as how to compile and link programs from DOS. It also describes how to compile and link programs using the BASIC compiler (BC) and the Microsoft Overlay Linker (LINK), provided with QuickBASIC; how to build Quick libraries using LINK; and how to build stand-alone libraries using the Microsoft Library Manager (LIB).
Appendixes	Contain additional helpful information, such as a description of the differences between QuickBASIC Version 4.0 and older versions of QuickBASIC, instructions for converting BASICA programs to QuickBASIC and calling programs written in other languages from QuickBASIC, and a command summary. A glossary at the end of this part defines terms used in this manual and in the other two manuals in this set, the *BASIC Language Reference* and *Programming in BASIC: Selected Topics.*

Typographic Conventions

This manual uses the following typographic conventions to convey information about QuickBASIC. Note that these conventions are somewhat simpler than those used in the other QuickBASIC manuals in this set.

Example of Convention	Description
`Example: input`	The typeface shown in the left column is used in examples to simulate the appearance of information printed on your screen. The bold version of this typeface indicates input entered in response to a prompt.
Apostrophe (')	The apostrophe marks a comment in sample programs. An apostrophe is entered by using the key that produces the single right quotation mark. This key may be labeled ' or ´ or ’. Do not use a single left quotation mark, produced by the key labeled ‘ or `.
KEYWORDS	Bold capital letters indicate BASIC keywords. Keywords are a required part of the statement syntax, unless they are enclosed in double brackets as explained below. In programs you write, you must enter keywords exactly as shown. However, you may use uppercase (capital) letters or lowercase letters.
placeholders	Items in italics are placeholders for information that you must supply, such as a file name. Italics are also occasionally used in the text for emphasis.
[[**optional items**]]	Items inside double square brackets are optional.
{*choice1*|*choice2*}	Braces and a vertical bar indicate a choice among two or more items. You must choose one of the items unless all of the items are also enclosed in double square brackets.
Repeating elements...	Three dots following an item indicate that more items having the same form may appear.

`Program`
`.`
`.`
`.`
`Fragment`
	A column of three dots tells you that part of a program has been intentionally omitted.

KEY NAMES	Small capital letters are used for the names of keys and key sequences, such as ENTER and CTRL+R. Notice that a plus (+) indicates a combination of keys. For example, CTRL+E tells you to hold down the CTRL key while pressing the E key. The names of the keys referred to in this manual correspond to the names printed on the IBM Personal Computer key tops. If you are using a different machine, these keys may have slightly different names. The cursor movement keys (sometimes called "arrow" keys) located on the numeric keypad to the right of the main keypad are called the DIRECTION keys. Individual DIRECTION keys are referred to either by the direction of the arrow on the key top (LEFT, RIGHT, UP, DOWN) or the name on the key top (PGUP, PGDN). The carriage-return key is referred to as ENTER.
"Defined term"	Quotation marks usually indicate a term defined in the text.

Note

The syntax below (for the **PRINT** statement) illustrates many of the typographic conventions in this manual.

PRINT [[*expressionlist*]] [[{, | ;}]]

Resources for Learning

The manuals in this package provide complete instructions for using the QuickBASIC programming environment. However, they do not attempt to teach you how to use DOS or begin programming in BASIC or assembly language. The following books contain more information on those subjects. They are listed for your convenience only. With the exception of its own publications, Microsoft Corporation does not endorse these books or recommend them over others on the same subject.

Microsoft Press books are available where software and books about microcomputers are sold. To order by phone, call 1-800-638-3030; in Maryland, call collect 824-7300. For a complete catalog of Microsoft Press books, write to Microsoft Press, 16011 NE 36th Way, Box 97919, Redmond, WA 98073-9717.

Books about BASIC

If you are new to BASIC, or if you need help learning how to program in BASIC, the following books may be helpful:

Dwyer, Thomas A., and Margot Critchfield. *BASIC and the Personal Computer.* Reading, Mass.: Addison-Wesley Publishing Co., 1978.

Enders, Bernd, and Bob Petersen. *BASIC Primer for the IBM PC & XT.* New York, N.Y.: New American Library, 1984.

Hergert, Douglas. *Microsoft QuickBASIC.* 2d ed. Redmond, Wash.: Microsoft Press. In press.

The first edition of this book discusses programming techniques appropriate for QuickBASIC Versions 2.0 and 3.0. Use the forthcoming second edition for information on programming in QuickBASIC Version 4.0.

Books about Assembly Language

If you need help learning to program in assembly language, the following books may be useful:

Lafore, Robert. *Assembly Language Primer for the IBM PC & XT.* New York, N.Y.: New American Library, 1984.

Metcalf, Christopher D., and Sugiyama, Marc B. *COMPUTE!'s Beginner's Guide to Machine Language on the IBM PC and PCjr.* Greensboro, N.C.: COMPUTE! Publications Inc., 1985.

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

Books about DOS

If you need more information about DOS, see the following:

Duncan, Ray. *Advanced MS-DOS.* Redmond, Wash.: Microsoft Press, 1986.

Wolverton, Van. *Running MS-DOS.* 2d ed. Redmond, Wash.: Microsoft Press, 1985.

Wolverton, Van. *Supercharging MS-DOS.* Redmond, Wash.: Microsoft Press, 1986.

Requesting Assistance

If you feel you have discovered a problem in the software, please report the problem using the Product Assistance Request form at the back of this manual.

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

PART 1

GETTING STARTED

CHAPTERS

1 Installing QuickBASIC....5

2 Your First QuickBASIC Session...17

PART 1 ◇ GETTING STARTED

Welcome to Microsoft QuickBASIC Version 4.0. QuickBASIC provides a convenient, integrated programming environment with all the tools you need to develop fast, compact programs quickly.

This part of the manual describes how to install QuickBASIC on your computer and start using it, with specific instructions for BASICA programmers and those who are already familiar with BASIC compilers.

CHAPTER 1

INSTALLING QUICKBASIC

1.1 Backing Up Your Disks....7

1.2 Disk Contents....7

1.3 Installing QuickBASIC: Hard-Disk Setup....9

1.4 Installing QuickBASIC: Floppy-Disk Setup....10

1.5 If You Have a Mouse....12

1.6 If You Have a Math Coprocessor....12

1.7 Setting DOS Environment Variables....13

1.7.1 The PATH Environment Variable....14

1.7.2 The LIB Environment Variable....15

1.7.3 The NO87 Environment Variable....15

This chapter describes how to set up QuickBASIC. When you have read this chapter you will know how to do the following:

- Copy (or "back up") your product disks.
- Install QuickBASIC software on your hard- or floppy-disk system.
- Set environment variables so QuickBASIC can locate all the files it needs.

If you are unfamiliar with any DOS procedures mentioned in this chapter, consult your DOS documentation for instructions.

1.1 Backing Up Your Disks

The first thing to do after unwrapping your package is to make working copies of the QuickBASIC distribution disks using the DISKCOPY command from DOS. Put the distribution disks in a safe place and use the copies from now on. If the copies are ever damaged or destroyed, you can make new copies from the distribution disks.

1.2 Disk Contents

Table 1.1 describes the files provided with this product. Your distribution disks contain additional files, including BASIC programs (files that have the .BAS extension) and other files added after this manual was printed. For a complete list of all files and their exact locations on your distribution disks, see the file named PACKING.LST on distribution disk 1.

Table 1.1
Files You Need When Using QuickBASIC

File Name	Description	Tasks Supported
QB.EXE	The Microsoft QuickBASIC program development environment	Creating, debugging, and running BASIC programs in memory; creating stand-alone executable programs; creating Quick libraries
BC.EXE	The BASIC command-line compiler invoked by the Run menu's Make EXE File command	Creating executable programs from Quick BASIC and from DOS; creating Quick libraries
SETUP.BAT	QuickBASIC installation program	Installing QuickBASIC
QB.QLB	Quick library containing support routines for DOS interrupts	---
QB.LIB	Stand-alone library containing support routines for DOS interrupts	Creating Quick libraries
QB.BI	Include file for use with QB.QLB and QB.LIB	See Chapter 8, "Quick Libraries," for information
README.DOC	Important information that became available after this manual was printed	---
PACKING.LST	List of all files included on the distribution disks	---
BRUN40.EXE	QuickBASIC run-time module	Running executable BASIC programs created with BRUN40.LIB
BQLB40.LIB	Quick-library run-time support	Creating Quick libraries
BRUN40.LIB	QuickBASIC run-time-module library	Creating executable programs from QuickBASIC and from DOS
BCOM40.LIB	QuickBASIC alternate run-time library	Creating executable programs from within QuickBASIC and from DOS. Executable programs created with this library do not require BRUN40.EXE in order to run.
LINK.EXE	The Microsoft Overlay Linker	Creating executable programs from within QuickBASIC and from DOS; creating Quick libraries

Table 1.1 *(continued)*

File Name	Description	Tasks Supported
LIB.EXE	The Microsoft Library Manager	Creating stand-alone libraries
NOCOM.OBJ	Object file for linking with programs that require no communications support	Minimizing the size of executable programs that require no communications support
MOUSE.COM	Mouse driver for use with QuickBASIC	Using the mouse
DEMO1.BAS, DEMO2.BAS, DEMO3.BAS	Example programs	Using the QuickBASIC practice session in Chapter 2, "Your First QuickBASIC Session"
QB.HLP	Text for QuickBASIC's on-line Help	Using on-line Help

1.3 Installing QuickBASIC: Hard-Disk Setup

The SETUP.BAT file, found on QuickBASIC distribution disk 1, copies QuickBASIC to your hard disk.

You can also install your QuickBASIC files without using SETUP.BAT. Depending on how you want to use QuickBASIC, install the appropriate files from Table 1.1.

Once you determine the files you need, do one of the following:

- Copy all the files to one directory on your hard disk.
- Copy different types of files to different directories, and set DOS "environment variables" that tell both DOS and the compiler where to find the files.

The second method keeps the organization of your hard disk tidier. See Section 1.7, "Setting DOS Environment Variables," for more information.

1.4 Installing QuickBASIC: Floppy-Disk Setup

To install QuickBASIC on a floppy-disk system, use the COPY command from DOS to copy the files from the distribution disks onto your own disks. To determine the files you need, consult Table 1.1.

Table 1.2 describes the recommended disk configuration to use for a two-floppy-drive system. You need four formatted floppy disks.

Table 1.2
Disk Setup for Two-Floppy-Drive System

Disk Number	Files
1	COMMAND.COM AUTOEXEC.BAT (optional) CONFIG.SYS (optional) QB.EXE QB.HLP MOUSE.COM
2	BRUN40.EXE BRUN40.LIB BQLB40.LIB LINK.EXE LIB.EXE BC.EXE
3	BCOM40.LIB LINK.EXE QB.LIB QB.BI*
4	BASIC source files QB.QLB (optional)† Quick libraries

* This file is required only if your program uses QB.LIB for DOS interrupt support. See Chapter 8, "Quick Libraries," for more information.

† This Quick library is required only if your program uses DOS interrupts. See Chapter 8, "Quick Libraries," for more information.

Using the above disk configuration, the key thing to remember is to invoke QuickBASIC from drive B while QB.EXE is in drive A. This ensures that your program will be able to locate any libraries it needs, and executable files will be written to the disk in drive B. To invoke QuickBASIC from drive B, put disk 1 in drive A and disk 4 in drive B.

1. Type `b:` and press ENTER to log onto drive B
2. Type `a:qb` to invoke QuickBASIC

Most of the time you will probably need only disk 1 and disk 4. You will need disks 2 and 3 in some of the following circumstances:

Processing Task	Drive Configuration
Writing and running programs in memory without a Quick library	Place disk 1 in drive A, and disk 4 in drive B.
Writing and running programs in memory with a Quick library	Place disk 1 in drive A and disk 4 in drive B. Disk 4 must contain the BASIC source file and the Quick library the program requires.
Creating a Quick library or an executable program using EXE Requiring BRUN40.EXE option from the Run menu's Make EXE File command	Place disk 1 in drive A, and place disk 4 in drive B. When you see the prompt "Cannot find file *filename*," insert disk 2 in drive A, type "a:", and press ENTER. If your program requires a Quick library, make sure the library is on the disk in drive B.
Creating an executable program using the Stand-Alone EXE File option from the Run menu's Make EXE File command	Place disk 1 in drive A and place disk 4 in drive B. When you see the prompt "Cannot find file *filename*," type "a:", insert the appropriate disk in drive A, and press ENTER. (At the first prompt, the appropriate disk will be disk 2, containing BC.EXE and LINK.EXE; at the next prompt, the appropriate disk will be disk 3, containing BCOM40.LIB). Finally, you will be prompted for the disk containing QB.EXE.

You may not be able to make a stand-alone executable file from a program that uses a Quick library. If you encounter such a problem, create the stand-alone executable file by loading the original modules in QuickBASIC.

Note

If you are using a single-drive system or 3.5-inch disks, see the README.DOC file on distribution disk 1 for a suggested disk setup.

1.5 If You Have a Mouse

You can use QuickBASIC with or without a mouse. You can also use a combination of mouse techniques with keyboard commands.

Microsoft QuickBASIC is designed for use with the Microsoft Mouse. Although many manufacturers advertise their pointing devices as being compatible with the Microsoft Mouse, QuickBASIC is guaranteed to work with them only if they emulate Microsoft Mouse function calls exactly. Check with your mouse manufacturer for details.

You can use QuickBASIC with MOUSE.COM or with MOUSE.SYS Version 6.00 or later. However, if your program uses mouse function calls, then you must run MOUSE.COM Version 6.11, provided on QuickBASIC distribution disk 3. The SETUP program automatically copies this file to your hard disk. If you do not use SETUP and you have a mouse, you should copy the version of MOUSE.COM provided with this product.

1.6 If You Have a Math Coprocessor

The 8087 and 80287 math-coprocessor chips provide fast and accurate "floating-point calculations." Floating-point calculations involve arithmetic done with numbers that are not whole numbers: 4.567/(23.4 ^ 2) is an example of a floating-point calculation.

Microsoft QuickBASIC automatically uses either of these math coprocessors if one is present. Otherwise, QuickBASIC uses software routines to emulate the coprocessor. Emulating the coprocessor results in slower execution than can be achieved with a coprocessor but provides a similar degree of accuracy.

Note

You can force QuickBASIC to ignore the presence of an 8087 or 80287 coprocessor by setting the NO87 environment variable. See Section 1.7.3, "The NO87 Environment Variable," for information.

Using the coprocessor—or emulating it with software—makes floating-point arithmetic generally much more accurate than it is when done with BASICA or previous versions of QuickBASIC.

1.7 Setting DOS Environment Variables

If you don't want to keep all your QuickBASIC files in one directory, you can put them in different directories, then use the PATH and SET commands from DOS to define environment variables that tell QuickBASIC where to find the files it needs.

You don't have to set any environment variables to run QuickBASIC, but they are helpful. If you do not set environment variables, QuickBASIC searches only the current working directory for needed files, and displays the message "Cannot find *file*. Input path:" if it cannot find them.

Here are the files QuickBASIC needs:

File Type and Extension	Description
Executable files (.EXE)	Files that are executed as your program is processed. They include QB.EXE (when you are working in the programming environment) and—if you are compiling and linking stand-alone programs or creating libraries—the BC.EXE compiler, the LINK.EXE linker, the BRUN40.EXE run-time module, and the LIB.EXE library manager.
Library files (.LIB)	Library files BRUN40.LIB, BCOM40.LIB, and BQLB40.LIB required by the linker, LINK.EXE.

Once you have set an environment variable, it remains effective until you reset it to a different value, or reboot or turn off your computer.

Sections 1.7.1 and 1.7.2 briefly describe two of the most commonly used environment variables—PATH and LIB—and explain how to assign values to them. If you have an 8087 or 80287 math coprocessor, you may also want to read Section 1.7.3 on setting the NO87 environment variable so your programs ignore a math coprocessor.

1.7.1 The PATH Environment Variable

The PATH environment variable tells the compiler where to look for executable files. QB.EXE and BC.EXE both look for the PATH environment variable.

Use the PATH command from DOS to define the PATH variable. Enter the command, a space or an equal sign, and one or more path specifications separated by semicolons (;). For example, the following setting for PATH tells the compiler to search for executable files on drive C:, first in the \BIN directory, then in the \QB directory:

```
PATH=C:\BIN;C:\QB
```

By setting the PATH as shown in this example, you could put QB.EXE in the \BIN directory, then start QuickBASIC from any directory.

Note

If you have DOS Version 3.0 or later, you can define the PATH variable with the SET command, as in the next line:

```
SET PATH=C:\BIN;C:\QB
```

If you are using a version of DOS earlier than Version 3.0, you cannot define the PATH variable with the SET command. With these earlier versions, you *must* use the PATH command.

1.7.2 The LIB Environment Variable

The LIB environment variable tells QuickBASIC where to locate a Quick library specified on the qb command line, and tells LINK.EXE where to find any library files it needs. Define the LIB environment variable with the SET command. For example, the following command tells the linker to look for libraries in the \LIB directory on drive C:

```
SET LIB=C:\LIB
```

1.7.3 The NO87 Environment Variable

As mentioned in Section 1.6, your programs will use an 8087/80287 coprocessor at run time if one is installed. However, you can override the use of the coprocessor and force a program to emulate its function by setting the NO87 environment variable. The following lines show two ways to set the NO87 variable:

```
SET NO87=Use of coprocessor suppressed
SET NO87=one or more spaces
```

The first setting causes the message "Use of coprocessor suppressed" to appear on the screen when a program that uses an 8087 or 80287 is executed, and an 8087 or 80287 is present. The second setting does not display a message.

To turn off forced emulation of a coprocessor, set NO87 equal to a null—or empty—value, as in the next example:

```
SET NO87=
```

Note that no spaces follow the equal sign (=) in this example.

CHAPTER 2

YOUR FIRST QUICKBASIC SESSION

2.1 If You Are a BASICA Programmer 19
- 2.1.1 Starting QuickBASIC 21
- 2.1.2 Loading a Program 21
- 2.1.3 Running Your Program 24
- 2.1.4 A Comparison of Interpreting, Compiling, and Running in QuickBASIC 26
- 2.1.5 Creating an Executable Program on Disk 28
- 2.1.6 Exiting QuickBASIC 29
- 2.1.7 Running a Program from DOS 29
- 2.1.8 How QuickBASIC Differs from BASICA 29
  - 2.1.8.1 Environment Enhancements 30
  - 2.1.8.2 New Language Features 31
- 2.1.9 What's Next 32

2.2 If You Have Used QuickBASIC Version 2.0 or 3.0 33
- 2.2.1 Starting QuickBASIC 34
- 2.2.2 Loading a QuickBASIC Program 34
- 2.2.3 Where Are Your Subprograms? 36
- 2.2.4 Running a QuickBASIC Program 38
- 2.2.5 Creating an Executable Program 38
- 2.2.6 Running a Program from DOS 39
- 2.2.7 Improvements in This Version of QuickBASIC 40
  - 2.2.7.1 Environment Enhancements 40
  - 2.2.7.2 New Language Features 41
- 2.2.8 What's Next 42

2 CHAPTER

2.3 If You Have Used a BASIC Compiler 42
2.3.1 Compiling and Linking from the Command Line 43
2.3.2 What's Next 44
2.4 Vital Information for All QuickBASIC Users 45
2.4.1 Why QuickBASIC Is Better Than Other BASIC Products 45
2.4.1.1 The Development Environment You've Always Wanted 45
2.4.1.2 New Language Features 47
2.4.2 The qb Command 49
2.4.3 Customizing the QuickBASIC Screen 51
2.4.4 The QB.INI File 52
2.4.5 Getting Help from QuickBASIC 52
2.4.6 QuickBASIC File-Naming Conventions 55
2.4.6.1 Path Names 55
2.4.6.2 Uppercase and Lowercase Letters 55
2.4.6.3 File-Name Extensions 56
2.4.6.4 Specifying File Names 57

This chapter offers you three choices for getting started using QuickBASIC, depending on your prior programming experience with BASIC:

Your Experience	Sections You Should Use
No BASIC programming experience or programming with BASICA or other interpreted BASIC, but never used QuickBASIC before	Section 2.1 "If You Are a BASICA Programmer," and Section 2.4, "Vital Information for All QuickBASIC Users"
Programming with other versions of QuickBASIC	Section 2.2 "If You Have Used QuickBASIC Version 2.0 or 3.0," and Section 2.4 "Vital Information for All QuickBASIC Users"
Programming with a Microsoft BASIC command-line compiler	Section 2.3 "If You Have Used a BASIC Compiler," and 2.4 "Vital Information for All QuickBASIC Users"

When you finish this chapter you will know how to do the following tasks with QuickBASIC:

- Start QuickBASIC.
- Load a QuickBASIC program.
- Run a QuickBASIC program in memory.
- Create an "executable" file (a file with the extension .EXE) on disk.
- Exit QuickBASIC.

2.1 If You Are a BASICA Programmer

Begin here if you have never programmed in BASIC, or if you are an interpreted-BASIC or BASICA programmer who has never used QuickBASIC before. If you have already had experience with other versions of QuickBASIC, go on to Section 2.2. If your experience is mostly with a Microsoft BASIC command-line compiler, you can read this section and the next as an introduction to the QuickBASIC program development environment or turn directly to Section 2.3.

Before starting the exercises in this section, you should have installed QuickBASIC according to the instructions in Chapter 1. You also need a BASIC program to work with. Begin with one of your own BASIC programs or the program DEMO1.BAS that is provided on QuickBASIC distribution disk 3.

Try a very simple program first. Some statements and functions behave a little differently in QuickBASIC than they do in BASICA, so if you're using your own program, you may want to consult Appendix A, "Converting BASICA Programs to QuickBASIC," before proceeding further. The program you choose should produce some output so you can tell if it runs correctly. Until you are completely familiar with the QuickBASIC editor, use a backup copy of any old programs you load into QuickBASIC.

Important

If you use your own BASICA program, be sure it is saved in ASCII format. QuickBASIC cannot load binary-format BASICA programs.

If you are not sure which format your BASICA program is in, look at it with the TYPE command from DOS. If the output from the TYPE command looks the same as what you see when you list the program in BASICA, then the program is saved in ASCII format. If however, a lot of nonalphabetic characters appear on the screen, then the program is saved in binary format.

If the program is in binary format, complete these steps before running it with QuickBASIC:

1. Start BASICA by typing:

   ```
   basica
   ```

2. Load your program (SAMPLE.BAS in this example) by typing:

   ```
   load "sample"
   ```

3. Resave the program in ASCII format by typing:

   ```
   save "sample",a
   ```

4. Quit BASICA by typing:

   ```
   system
   ```

2.1.1 Starting QuickBASIC

If you are using a color monitor or a monochrome monitor with a monochrome adapter, to start QuickBASIC, type

```
qb
```

and press ENTER.

QuickBASIC can detect what type of display-adapter card you have, but not what type of monitor. If you are using a black and white (composite) monitor with a CGA, to start QuickBASIC, type

```
qb /b
```

and press ENTER.

If QuickBASIC doesn't start, make sure the file QB.EXE is in your current directory or in a directory specified by your PATH variable, and try again.

When QuickBASIC starts you are ready to load your first program.

2.1.2 Loading a Program

You must load a program into QuickBASIC before you can run it. Follow these steps to load a program:

1. Press the ALT key. The word "File" at the top left of the screen is now "selected" (highlighted).

   Note that you don't have to hold down the ALT key. When working with menus, you can press and release the ALT key before going on to step 2.

2. Press F to display the complete list of commands in the File menu (shown in Figure 2.1). See Chapter 3, "Getting Around in the QuickBASIC Environment," for a complete explanation of menus.

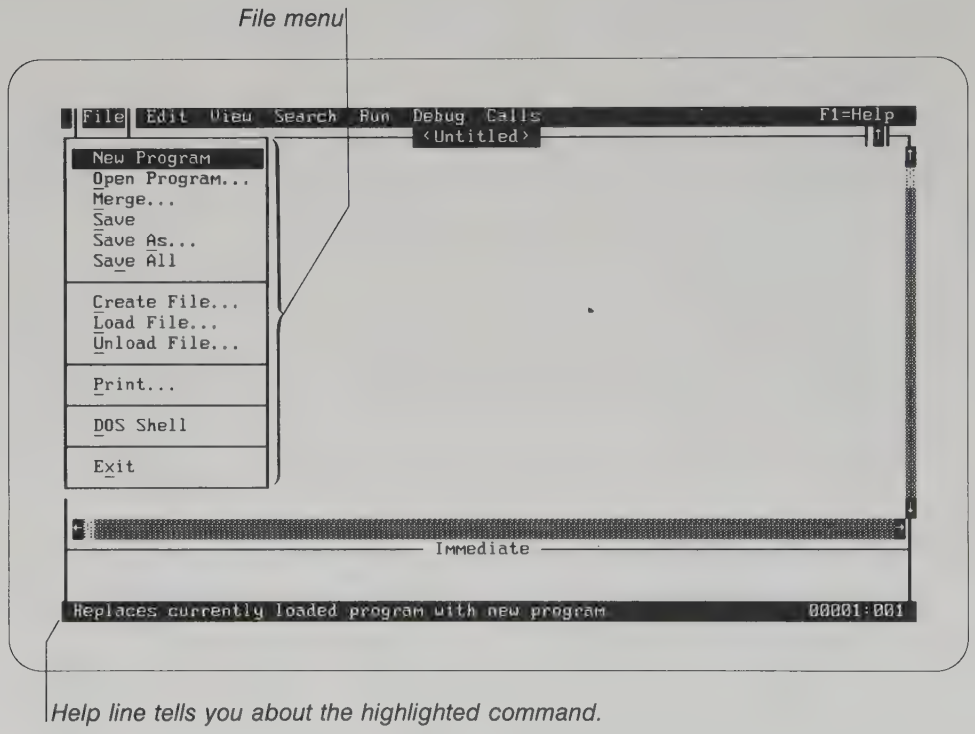


Figure 2.1 The File Menu

3. Press O to choose the Open Program command from the File menu.

   This brings up the Open Program "dialog box," shown in Figure 2.2. A dialog box is where you enter information a command needs. In this case, the Open Program command needs the name of the file you want to load.

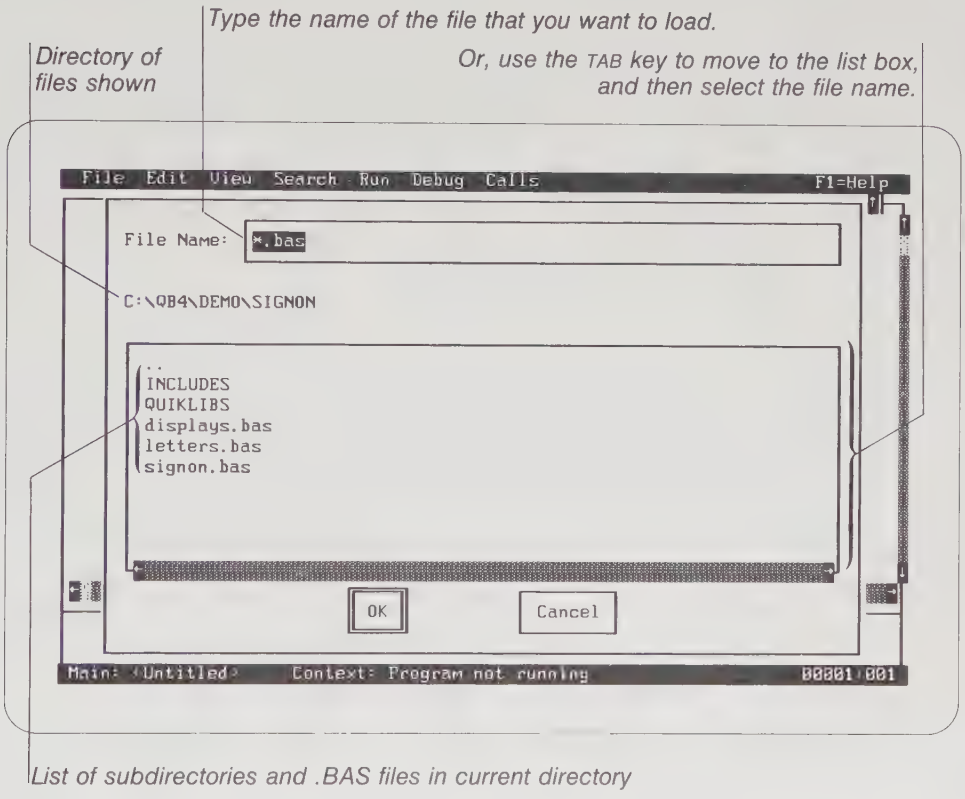


Figure 2.2 Open Program Dialog Box

Your screen may look different from the one in Figure 2.2, since you may have different files and directories on your disk. See Chapter 4, "Managing Source Files" for more information about the Open Program dialog box.

4. Type the name of the file you want to load (and an optional directory path to the file if it's in another directory). You'll see the name you type appear in the box next to the words File Name. Press ENTER.

5. If you are loading one of your own BASIC programs, you may see an error message. The most likely cause is that the program was saved in binary format (what you get when you just type

   ```
   save programname
   ```

   in BASICA; see the "Important" note at the beginning of this section on how to correct this problem). If you see an error message, press ESC to clear the error message from your screen, then repeat steps 1–4, but this time load the program DEMO1.BAS, provided on the QuickBASIC distribution disk 3.

2.1.3 Running Your Program

Now that you have successfully loaded your program, you are ready to run it. This section shows you how to run your program from inside the QuickBASIC environment. The process QuickBASIC uses—known as "translating to executable code"—is very similar to the way you run programs from BASICA. Follow these steps to run the program from QuickBASIC:

1. Press ALT and then press R to open the Run menu, shown in Figure 2.3.

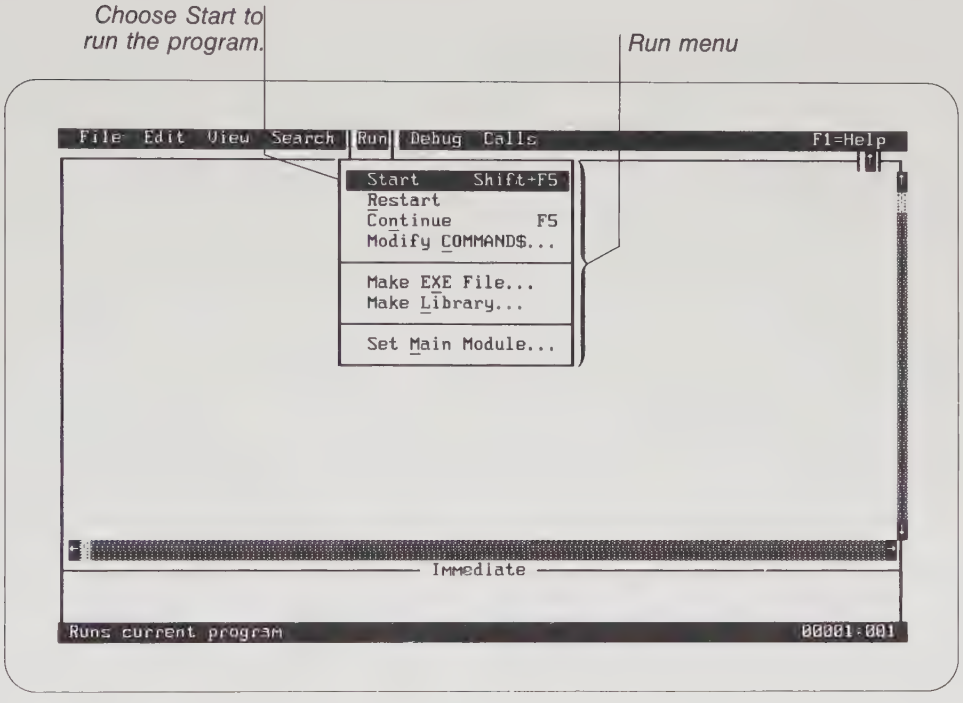


Figure 2.3 The Run Menu

2. Press S to choose the Start command. This command runs the program.

If you run one of your own BASICA programs, you may get an error. Some statements require modification to run properly with QuickBASIC. They are described in Appendix A, "Converting BASICA Programs

to QuickBASIC." If you see an error message, follow these steps to replace the current program with the example program, DEMO1.BAS, provided on distribution disk 3:

1. Press ESC to erase the error message.
2. Press ALT, then press F to open the File menu.
3. Press O to display the Open Program dialog box.
4. Type

   ```
   DEMO1.BAS
   ```

   and press ENTER to load the program.
5. Repeat the steps in the preceding list to run DEMO1.BAS.

Note

You can also turn your program into a "stand-alone executable file," using a process known as "compiling to disk." A stand-alone program, as its name implies, does not require you to load QuickBASIC before running the program. This ability to create stand-alone programs is one of many advantages that QuickBASIC has over BASICA. See Section 2.1.5, "Creating an Executable Program on Disk," for further information.

2.1.4 A Comparison of Interpreting, Compiling, and Running in QuickBASIC

Interpreters and compilers are both programs that translate the statements you write into instructions that can be understood by a computer. A traditional interpreter both translates and executes each statement before moving on to the next, whereas traditional compilers translate all statements in a program before any execution begins. Many compilers also require other steps to precede execution of a program.

BASICA is an example of an "interpreter." When you load a program into BASICA and run it, each statement is translated, then executed, before BASICA moves on to translate and execute the next statement. The process is analogous to having a human interpreter help you understand a

book written in an unfamiliar foreign language. Each time you need to understand a sentence in the book, the interpreter is there to translate it. One drawback to this process is obvious: you always need the interpreter to decipher things for you.

The QuickBASIC package includes two different tools for translating your programs. The one you have been using, as described in the previous sections, is QB.EXE, the program development environment. The other, called BC.EXE, is a command-line compiler. It is strictly a translator—it does not run programs. In this sense, using BC is like having the human interpreter mentioned above write a translation of the *entire* book into your language. Once that is done, you don't need the services of the translator any more; instead, you can just look at the translation.

A program compiled with BC.EXE can be linked and saved as a disk file known as a stand-alone executable program. (Files of stand-alone executable programs have the base name of the original BASIC source file plus the extension .EXE, and are referred to as "executable" files.) Also, since a compiler translates your program only once (while an interpreter must translate each statement anew each time it runs a program) compiled programs run much faster than interpreted programs.

QB.EXE, the QuickBASIC program development environment, is neither a traditional interpreter nor compiler, though it combines the best of both worlds: the convenience of an interpreter and the execution speed of a compiler. It lets you choose between working with your program in memory (so you can run it at any time) or compiling the program and saving it as a disk file that you can execute outside the environment. When you work with a program in memory, QuickBASIC translates each line as you enter it (unlike traditional compilers that make translation a completely separate step). Thus, in QuickBASIC your program is always ready to run, regardless of its state of completion. See Section 2.4.1.1, "The Development Environment You've Always Wanted," for more information.

Working with programs in memory is useful during program development, since you can easily alter the program or fix bugs, then rerun the program immediately to test your changes. Once you have completely debugged the program, you can create an executable file and run the program directly from DOS without first having to start QuickBASIC.

The next section describes the procedure for creating executable files with QuickBASIC.

2.1.5 Creating an Executable Program on Disk

After you have successfully loaded and run a program from within QuickBASIC, you are ready to produce an executable file on disk.

Follow these steps to create an executable file in the current directory:

1. Press ALT, then press R to open the Run menu.
2. Press X to choose the Make EXE File command. The dialog box shown in Figure 2.4 appears. The executable-file name appears in the box next to the words EXE File Name.

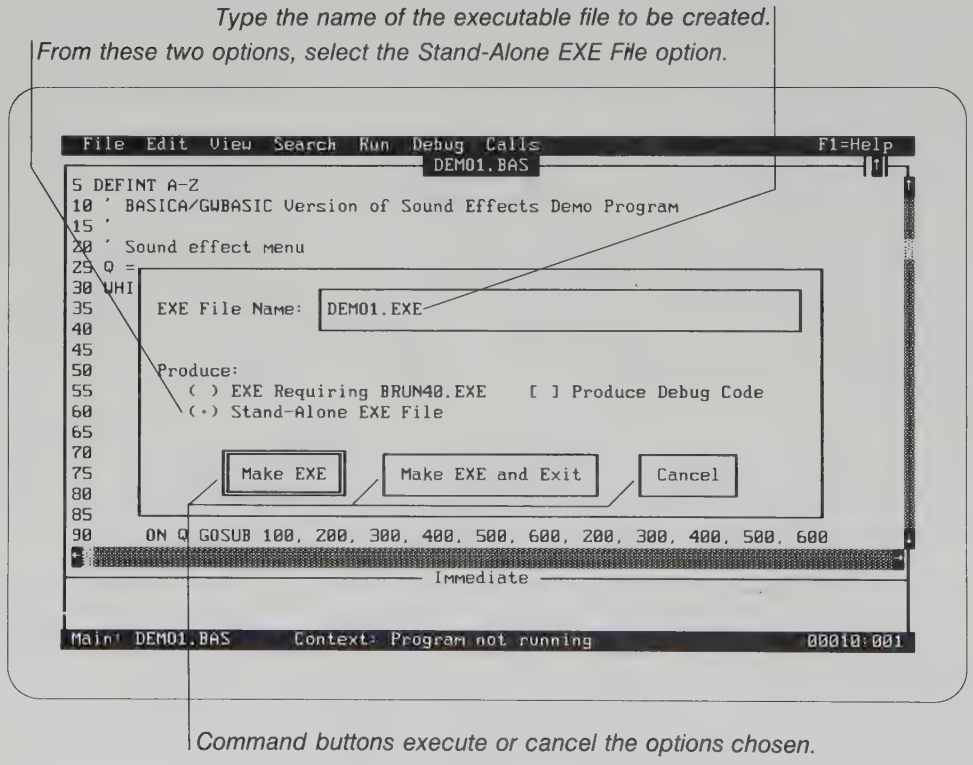


Figure 2.4 Make EXE File Dialog Box

3. Press TAB, then press A to choose the Stand-Alone EXE File option, since you don't want to change the file name.
4. Press M to choose the Make EXE command button.

As the program compiles, you will see messages from the BC compiler and the linker. When the process is complete, you will be back in the QuickBASIC environment.

2.1.6 Exiting QuickBASIC

To test the stand-alone executable file, you must leave QuickBASIC and return to DOS. To exit QuickBASIC

1. Press ALT, then press F to display the File menu.
2. Press X to choose the Exit command.

   The DOS prompt appears. (If you have made changes to the program, a dialog box appears first asking if you want to save the loaded file.)
3. Press Y to save the changes, or N to exit without saving the changes.

2.1.7 Running a Program from DOS

Use the DIR command from DOS to see the name of the executable file you just created. You should find a file with the base name that appeared in the EXE File Name text box, plus the .EXE extension.

To run the stand-alone version of your program

- Type the base name of the file and press ENTER.

2.1.8 How QuickBASIC Differs from BASICA

QuickBASIC differs from BASICA in both its "programming environment"—the part of QuickBASIC where you edit and debug your program—and in its extensions to the BASIC language itself. Both of these differences are described briefly in the next two sections and more fully in Section 2.4.1, "Why QuickBASIC is Better Than Other BASIC Products."

2.1.8.1 Environment Enhancements

QuickBASIC environment enhancements include changes in methods for editing and debugging files.

- **Editing**

  The centerpiece of the QuickBASIC environment is its "smart" editor. While this editor is similar to the one you have used with BASICA, it offers the following features not found in BASICA:

  — Syntax checking as you enter each line of a program. This catches most syntax errors and missing punctuation (such as commas or quotes) before you run your program.

  — Automatic capitalizing of keywords such as **PRINT**. This helps you distinguish keywords from your own labels and names, and serves as a further syntax check as you type in your program.

  See Chapter 5, "Editing," for a complete description of the QuickBASIC editor.

- **Debugging**

  QuickBASIC's debugging commands let you check variable values and stop a program at any statement or when a variable reaches a predetermined value. Moreover, you can stop a program in QuickBASIC, make changes, then resume from the point where you stopped with all your data intact. If you have made a change that prevents the program from continuing, QuickBASIC lets you choose between continuing without the change, or starting at the beginning of the program with the change incorporated.

  In contrast, BASICA debugging is limited to **TRON/TROFF**, and judicious use of the **PRINT**, **STOP**, and **CONT** statements. While BASICA allows you to stop a program and check its variable values, you cannot change the program and resume at the point where you stopped.

  Like BASICA, QuickBASIC supports direct-mode statements. In BASICA, these are statements entered without line numbers. In QuickBASIC, a direct-mode statement is entered in the Immediate window.

2.1.8.2 New Language Features

QuickBASIC extends the BASIC language to provide many of the features of more structured languages such as Pascal and C. These extensions include the following:

- **SUB** and **FUNCTION** procedures facilitate your structured programming.
- **TYPE...END TYPE** statements let you group different types of variables into one composite user-defined type. This greatly simplifies input from and output to random-access files.
- New control-flow constructions, including **SELECT CASE**, block **IF...THEN...ELSE**, and **DO...LOOP**, help keep your program's logic clear and easy to modify.
- Long (32-bit) integers let you do financial calculations using cents, so the results are exact (no round-off errors). These calculations can be done more quickly than is the case when single-precision numbers are used to do the calculations with dollars and cents.
- Recursion allows your **SUB** and **FUNCTION** procedures to call themselves.
- IEEE-format numbers provide additional range and accuracy in floating-point calculations.
- Binary access of disk files lets you read and modify non-ASCII disk files.

2.1.9 What's Next

Now you know the fundamentals of using QuickBASIC: loading and running programs in memory, as well as compiling programs to stand-alone executable files. Next you'll probably want to learn more about menus, commands, and dialog boxes; the new features that QuickBASIC adds to the BASIC language; and how to use the editor. The following list tells you where to locate this information:

Information	Location
More information on new features	Section 2.4.1, "Why QuickBASIC Is Better Than Other BASIC Products"
Options to use when starting QuickBASIC	Section 2.4.2, "The qb Command"
Using menus, windows, and dialog boxes	Chapter 3, "Getting Around in the QuickBASIC Environment"
Modifying your old BASICA programs to run with QuickBASIC	Appendix A, "Converting BASICA Programs to QuickBASIC"
Programming in BASIC using new language features	*Programming in BASIC: Selected Topics*
Customizing the QuickBASIC screen	Section 2.4.3
Using QuickBASIC's Help system	Section 2.4.5

2.2 If You Have Used QuickBASIC Version 2.0 or 3.0

Begin here if you have used previous versions of QuickBASIC. If your experience is limited to using a BASIC interpreter like BASICA (or if you have not used BASIC at all), begin with Section 2.1. If you have used a Microsoft BASIC command-line compiler and do not wish to use the QuickBASIC environment features, turn to Section 2.3.

Before reading the rest of this section, you should have installed QuickBASIC according to the instructions in Chapter 1.

Note

While most programs written with previous versions of QuickBASIC should load and run immediately, certain differences between this and previous versions of QuickBASIC may require slight modification of old programs. The following sections alert you to important language and environment differences and refer you to the information you'll need to get your programs running. QuickBASIC handles some formatting tasks, particularly tabbing and indentation levels, differently than other editors.

Until you are completely familiar with the QuickBASIC editor, you should work with backup copies of your old source files.

Sections 2.2.1–2.2.4 show you how to start QuickBASIC, then guide you through loading, inspecting, and running a program. Sections 2.2.5–2.2.6 explain how to create executable files from within the QuickBASIC environment, then run them from DOS. Sections 2.2.7–2.2.8 describe differences between this version of QuickBASIC and others and point you to other parts of the manual that you should find useful.

2.2.1 Starting QuickBASIC

Starting this version of QuickBASIC is similar to starting previous versions. If you are using a color monitor or a monochrome monitor with a monochrome graphics adapter, type

```
qb
```

and press ENTER.

QuickBASIC can detect what type of display adapter you have, but not what type of monitor. If you are using a black-and-white (composite) monitor with a CGA, type

```
qb /b
```

and press ENTER.

If you get an error message, make sure that QB.EXE is in your current directory or in a directory specified by your PATH variable, and try again.

2.2.2 Loading a QuickBASIC Program

Programs created with QuickBASIC Version 2.0 or 3.0 should load without difficulty with one exception: if you have programs that use include files containing **SUB...END SUB** statements, read Section 4.7, "Merging the Contents of Two Files," before trying to load them. For now, choose a program that does not use include files containing **SUB...END SUB** statements. For your convenience, distribution disk 3 contains a sample program, DEMO2.BAS, that you can use to complete the following exercise.

The following steps show how to load a program into QuickBASIC:

1. Press the ALT key.

   This "selects" (highlights) the first item (File) on the menu bar. Note that you don't have to hold down the ALT key. When working with menus you can press it and release it before going on to step 2.

2. Press F to open the File menu and then press O to display the Open Program command's dialog box. The list box in the center shows all BASIC programs available in the current directory.

3. Type the program's name in the text box as a full path name, if the program you want to load is not listed; otherwise just type its name and press ENTER.

   If your program is more than a few lines, you can watch QuickBASIC tick off each line in the counter in the lower-right corner of the screen as it is loaded. When your program is loaded, its name appears in the window's "title bar," centered between the menu bar and the beginning of your code (see Figure 2.5).

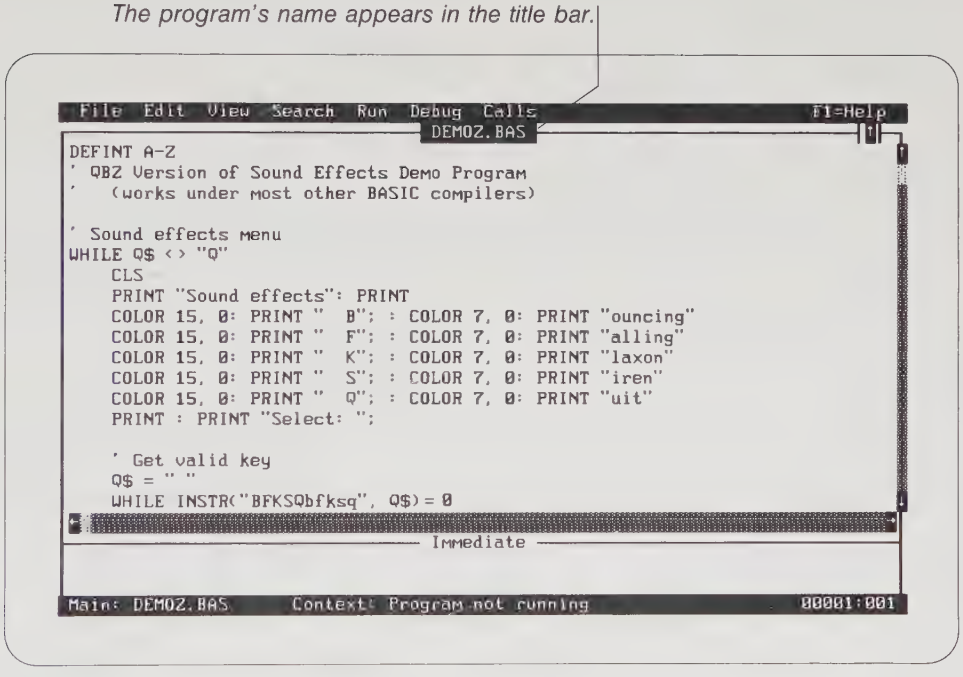


Figure 2.5 QuickBASIC Screen with DEMO2.BAS Loaded

2.2.3 Where Are Your Subprograms?

This version of QuickBASIC includes a special "smart" editor. The smart editor recognizes BASIC syntax and gives special treatment to subordinate parts of your program, such as **FUNCTION...END FUNCTION** or **SUB...END SUB** procedures.

QuickBASIC extracts these procedures and keeps them in a separate part of memory. However, they are still part of your program and are ready to run along with the "module-level" code (that is, the part of your program you see when you first load it).

To view your **SUB** procedures, do the following:

1. Press ALT, then press V to open the View menu and S to choose the SUBs command (or press F2, the shortcut key).

   The SUBs dialog box, shown in Figure 2.6, appears. It lists the parts of the program currently in memory. The name of the program appears at the top of the list. Subordinate procedures are indented beneath the name of the program.

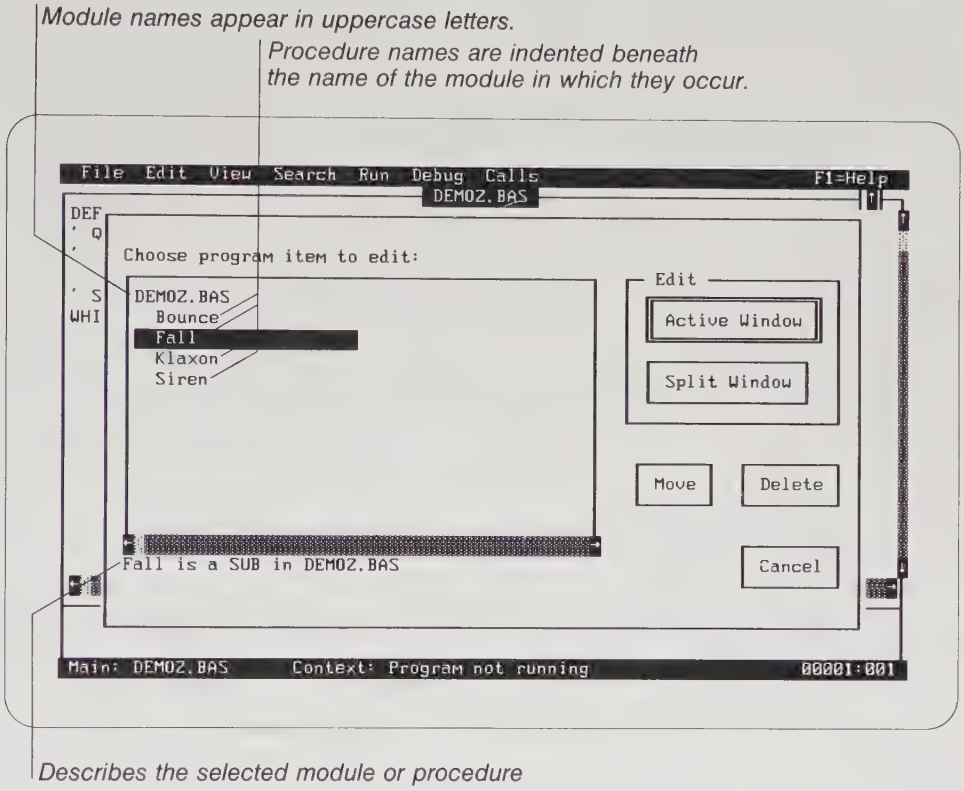


Figure 2.6 SUBs Dialog Box

2. Use the DOWN key to move the highlight down to select a **SUB**; press ENTER to view or edit that **SUB**.

3. Press ALT, then V, and S (or just press F2) to display the SUBs dialog box again.

4. The program name is selected, so press ENTER to return to the module-level code.

See Section 4.3.2, "Displaying Multiple Modules," for more information on the features of the SUBs dialog box.

2.2.4 Running a QuickBASIC Program

If you have successfully loaded one of your own QuickBASIC programs, you should be able to run it now. If your programs have not loaded properly, use one of the demonstration programs from the distribution disk.

To run the program with the Start command from the Run menu

- Press ALT, then press R and S (or just press SHIFT+F5).

If QuickBASIC generates an error message when you run your program, try loading and running a different program so you can complete the next two sections. As soon as you get a program to run without errors, you are ready to move to the next section and turn your program into an executable file that can be run directly from the DOS prompt. (See Appendix A, "Converting BASICA Programs to QuickBASIC," and Appendix B, "Differences from Previous Versions of QuickBASIC," for information about modifying old programs, and the differences between this and previous versions of QuickBASIC, respectively.)

2.2.5 Creating an Executable Program

You can compile your program and produce a file that can be executed from the DOS prompt. Use the following procedure to create an executable program on disk from within QuickBASIC:

1. Press ALT and then press R to display the Run Menu.

2. Press X. (If QuickBASIC asks if you want to save your file, press Y.) The Make EXE File dialog box (shown in Figure 2.7) appears. The executable-file name appears in the box next to the words EXE File Name.

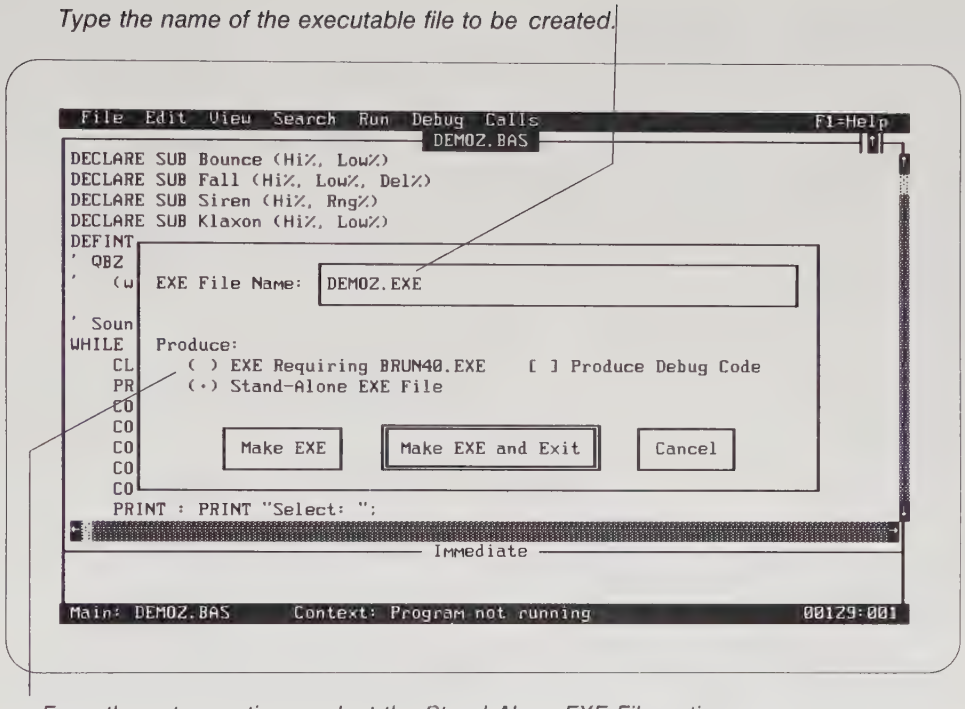


Figure 2.7 Make EXE File Dialog Box

3. Hold down the ALT key while pressing A to select the Stand-Alone EXE File option, since you don't want to change the file name.

4. Press E to choose the Make EXE and Exit command button.

QuickBASIC now creates an executable file and returns you to the DOS prompt.

2.2.6 Running a Program from DOS

Use the DIR command from DOS to confirm the presence of your new executable file. You should find a file having the name that appeared in the EXE File Name text box, plus the .EXE extension.

To run the stand-alone version of your program

1. Type its base name at the DOS prompt.
2. Press ENTER.

2.2.7 Improvements in This Version of QuickBASIC

While similar to previous versions—especially in appearance and ease of use—this version of QuickBASIC also has significant improvements, especially in the following two categories:

1. A more sophisticated "program development environment" that integrates editing, debugging, running, compiling, and making libraries.
2. Significant extensions to the BASIC language itself

Sections 2.2.7.1 and 2.2.7.2 describe these briefly. See Section 2.4.1, "Why QuickBASIC Is Better Than Other BASIC Products," for detailed information on these improvements.

2.2.7.1 Environment Enhancements

Previous versions of QuickBASIC separated the processes of editing, debugging, and running programs. With this version, all these processes can take place simultaneously. As you edit, you are also translating each statement into executable code and debugging your program. This eliminates the tedious edit-compile-run-debug-edit cycle and results in faster and easier program development.

- The smart editor checks the syntax of each line as soon as you type it and shows you the logical organization of your program, displaying subordinate procedures such as **SUB...END SUB** in their own windows for quick access and easy editing.
- Instant translation means that each source line is translated to executable code in memory on entry, letting you catch and correct most syntax errors immediately. Your program is always ready to run.
- Interactive debugging lets you stop a running program at any point, check variable values, edit the program, then resume at the point where you stopped. If you make changes that would prevent the program from continuing, you can choose between continuing without your changes, or restarting the program at the beginning, with the changes incorporated.
- A direct-execution window called the Immediate window lets you test program statements. In the Immediate window, whatever you type is executed as soon as you press ENTER.

- QuickBASIC now supports multiple modules in memory so that you can do all your programming from within the environment. When you are ready to create an executable file, QuickBASIC compiles and links all modules in a single operation.
- Automatic library generation and management with Quick libraries lets you turn the procedures and routines you write into virtual extensions of the BASIC language, just by choosing a single command from the Run menu. Furthermore, every time you create or modify a Quick library, QuickBASIC automatically creates or updates a corresponding stand-alone (.LIB) library, which you can link with programs from the DOS command line.

2.2.7.2 New Language Features

QuickBASIC's new language features provide greater flexibility and power for your programs:

- **FUNCTION** procedures give you another convenient way to break a program into logically distinct units, thereby facilitating structured programming.
- The **TYPE...END TYPE** statement lets you group different types of variables into one composite user-defined type.
- New control-flow constructions help you keep your program's logic clear and easy to modify.
- Long (32-bit) integers let you do financial calculations using cents, so the results are exact (no round-off errors). These calculations can be done more quickly than is the case when using single-precision numbers to do the calculations with dollars and cents.
- Recursion means your **SUB** and **FUNCTION** procedures can call themselves.
- IEEE-format numbers provide additional range and accuracy in floating-point calculations.
- Binary access of disk files lets you read and modify non-ASCII disk files.

2.2.8 What's Next

Sections 2.2.1–2.2.7 have given you a brief overview of how this version of QuickBASIC differs from QuickBASIC Versions 2.0 and 3.0. The next step is to learn more about menus, commands, and dialog boxes, and how to use the editor. The list below shows where to find this information:

Information	Location
More information on new features	Section 2.4.1, "Why QuickBASIC Is Better Than Other BASIC Products"
Options to use when starting QuickBASIC	Section 2.4.2, "The qb Command"
Using menus, windows, and dialog boxes	Chapter 3, "Getting Around in the QuickBASIC Environment"
Features that are new or improved in this version of QuickBASIC	Appendix B, "Differences from Previous Versions of QuickBASIC"
Programming in BASIC with the new language features	*Programming in BASIC: Selected Topics*
Customizing the QuickBASIC screen	Section 2.4.3
Using QuickBASIC's Help system	Sections 2.4.5

2.3 If You Have Used a BASIC Compiler

Begin here if you are experienced with other BASIC compilers, or if you use the separate compilation method with QuickBASIC Versions 2.0 or 3.0 to process your programs.

In this section you will learn how to compile and link a program with BC.EXE, QuickBASIC's stand-alone command-line compiler, and where to look for information on QuickBASIC's programming environment and language extensions to write programs that are easier to maintain and debug.

2.3.1 Compiling and Linking from the Command Line

Most of the time you will probably want to develop your programs within QuickBASIC, the advantages of which are discussed in Section 2.4.1, "Why QuickBASIC Is Better Than Other BASIC Products." However, you can also compile and link a QuickBASIC program from the DOS command line, as explained in this section.

Before reading the rest of this section, you should have installed your QuickBASIC software as described in Chapter 1, "Installing QuickBASIC."

You can use one of your own BASIC programs in the following exercise or the DEMO2.BAS program supplied with QuickBASIC. If you use your own program for this exercise, you should choose a very simple program.

If you get an error message while using any of the following steps to compile your program from the command line, start again from step 1 with the DEMO2.BAS program just for the purpose of completing this exercise:

1. Enter the following command line after the DOS prompt to compile the program:

   `bc` *`filename;`*

   This command creates an object file with the base name of the source file and the extension .OBJ, and then the DOS prompt reappears.

Note

There are many options you can use with the bc command (and some you *must* use; for instance, if the program you chose uses error or event trapping, you must compile it with special options). See Section 9.4.2 for more information on bc options.

---

2. Enter the following command line to link your program:

   `link /e` *`filename`*`.obj ;`

   This step creates an executable file with the base name of the source file and the extension .EXE. The /e option "packs" the executable file so it requires less disk space and loads faster than an unpacked file. See Sections 9.5.5–9.5.6 for a full description of all the options available with the link command.

Now try running the program. If it ran correctly with previous Microsoft versions of BASIC, it should run without errors now.

2.3.2 What's Next

This section has given you an overview of the BC compiler and LINK. Now you are ready to study the QuickBASIC environment in detail and learn how to use it in developing your programs. The following list tells you how to locate the information you need:

Information	Location
Summary of new environment and language features	Section 2.4.1, "Why QuickBASIC Is Better than Other BASIC Products"
Options to use when starting QuickBASIC	Section 2.4.2, "The qb Command"
Using menus and dialog boxes	Chapter 3, "Getting Around in the QuickBASIC Environment"
Programming in BASIC with the new language features	*Programming in BASIC: Selected Topics*
Customizing the QuickBASIC screen	Section 2.4.3
Using QuickBASIC's Help system	Section 2.4.5

2.4 Vital Information for All QuickBASIC Users

The rest of this chapter contains information that you should find useful, regardless of your background in BASIC. The following sections discuss improvements to QuickBASIC and show you how to start QuickBASIC, customize the appearance of the QuickBASIC screen to suit your own tastes, and get on-line Help from QuickBASIC.

2.4.1 Why QuickBASIC Is Better Than Other BASIC Products

QuickBASIC's new programming environment enhancements and its extensions to the BASIC language let you rapidly create and debug programs that run with maximum speed and reliability.

2.4.1.1 The Development Environment You've Always Wanted

Until now, BASIC programmers have had to choose between BASIC interpreters and BASIC compilers. The advantage of a BASIC interpreter such as BASICA is that there is no separate step required to translate your code into something comprehensible to your computer. Thus, the transition from writing and debugging to running is virtually instantaneous. However, interpreted programs have their disadvantages, chief among them being their slow execution speed.

Compilers overcome this speed disadvantage, but with a cost to the programmer. Less sophisticated BASIC compilers require a time-consuming cycle of editing, compiling, reediting to fix compile-time errors, recompiling, and so on, until you fix all the compile-time errors. And when that's done, you still have to link the compiled file with libraries that allow it to run by itself, then run the program to see if it behaves the way you expect. If there are run-time errors, you have to start the whole process again.

QuickBASIC frees you from the tedium of the edit/compile/link/debug cycle by integrating these steps into a seamless process, all of which is done in memory, rather than with physical files on disk. This results in much faster and easier program development. QuickBASIC provides the convenience of an interpreter combined with the power and execution speed of a compiler.

The following list discusses some of the features of the QuickBASIC program-development environment that make it so easy and convenient to use:

- **Interactive Editor**

  The centerpiece of the QuickBASIC environment is a smart editor that checks the syntax of each line as soon as you type it. If the syntax is correct, the line is translated to executable code immediately; if not, a description of the error appears. The editor also capitalizes keywords and corrects certain omissions and errors. For example, it will insert a quotation mark at the end of a **PRINT** statement that is missing one.

  QuickBASIC also shows you the logical organization of your program. For example, it displays subordinate procedures such as **SUB...END SUB** in their own windows for easy access and fast editing.

- **Instant Execution**

  Because each line is translated to executable code on entry, you can immediately catch and correct most errors. Also, you don't have to wait for your program to compile after you finish editing it—it is ready to run.

- **Interactive Debugging**

  You don't have to switch back and forth between editing, running, and debugging any more. For example, you can stop a running program at any point, edit the program, then resume at the point where you stopped. If you have made changes that would prevent the program from continuing at that point, you can choose between continuing without your changes, or restarting the program at the beginning, with the changes incorporated.

  The Immediate window at the bottom of the screen lets you test program statements. In the Immediate window, whatever you type is executed as soon as you press ENTER. You can even make temporary changes to data in a running program, so you don't have to restart your program to see the effects of a change.

  Many of QuickBASIC's debugging features look and work like those in the Microsoft CodeView® debugger. In fact, using the BC compiler, you can generate modules compatible with the CodeView debugger. This makes them easily debugged when they are later linked into larger projects.

- **Quick Creation of Executable Files**

  When you are satisfied with the way your program is running within QuickBASIC, it takes only four keystrokes to make a version that runs directly from the DOS command line.

- **Multiple Modules in Memory**

  Before this version of QuickBASIC, you could program with multiple modules only by writing the modules in QuickBASIC, then leaving QuickBASIC to create the program by using a BASIC command-line compiler and linking the modules together with a linker. QuickBASIC now supports more than one module in memory. If you decide to compile all the modules into an executable file, QuickBASIC does this in a single operation.

- **Automatic Library Generation and Management**

  Quick libraries turn the procedures and routines you write into virtual extensions of the BASIC language. In this version of QuickBASIC, you can make a Quick library by just choosing a command from the Run menu. Furthermore, every time you create or modify a Quick library, QuickBASIC automatically creates or updates a corresponding stand-alone (.LIB) library, which you can link with programs from the DOS command line.

2.4.1.2 New Language Features

QuickBASIC's new language features provide greater flexibility and power for your programs:

- **SUB and FUNCTION Procedures**

  Procedures facilitate structured programming by allowing you to break a program into logically distinct units. See Chapter 2, "Procedures: Subprograms and Functions," in *Programming in BASIC: Selected Topics*, for more information.

- **TYPE...END TYPE**

  The **TYPE...END TYPE** statement lets you group different types of variables into one composite user-defined type (this is similar to C-language "structures" and Pascal "records"). The **TYPE...END TYPE** statement greatly simplifies input from and output to random-access files.

  See Chapter 2, "Data Types," in the *BASIC Language Reference*, for more information on user-defined data types; and Chapter 3, "File and Device I/O," in *Programming in BASIC: Selected Topics*, for more information on how user-defined types simplify random-access I/O.

- **New Control-Flow Constructions**

  SELECT CASE, block **IF...THEN...ELSE**, and **DO...LOOP** help you keep your program's logic clear and easy to modify.

  See Chapter 1, "Control-Flow Structures," in *Programming in BASIC: Selected Topics*, and the entries for these statements in the *BASIC Language Reference*, for more information.

- **Long (32-bit) Integers**

  Long integers accommodate numbers between approximately plus or minus two billion. Calculations done with whole numbers that fall within this range are much faster and generally more accurate than floating-point calculations done with single-precision numbers.

  Long integers let you do financial calculations using cents, so the results are exact (no round-off errors). These calculations can be done more quickly than is the case when using single-precision numbers to do the calculations with dollars and cents.

  See Chapter 2, "Data Types," in the *BASIC Language Reference*, for more information on long integers.

- **Recursion**

  Recursion means your **SUB** and **FUNCTION** procedures can call themselves. See Chapter 2, "Procedures: Subprograms and Functions," in *Programming in BASIC: Selected Topics* and Chapter 4, "Programs and Modules," in the *BASIC Language Reference*, to learn more about this programming technique.

- **IEEE-Format Numbers**

  QuickBASIC now uses IEEE-format numbers for additional range and accuracy in floating-point calculations. See Appendix B, "Differences from Previous Versions of QuickBASIC," for further information.

- **Binary Access of Disk Files**

  Binary access lets you read and modify non-ASCII disk files. See Chapter 3, "File and Device I/O," in *Programming in BASIC: Selected Topics*, for more information.

- **Testing Command-Line Arguments**

  Previous versions of QuickBASIC supported the **COMMAND$** function, which allows you to pass command-line arguments to a compiled program. While this function is useful to have for compiled programs, the only way you could test a program using **COMMAND$** was either to compile the program, or repeatedly

invoke QuickBASIC with different arguments after the /cmd option. (See Section 2.4.2, "The qb Command," for more information on /cmd and other QuickBASIC options.) However, by using the Modify COMMAND$ command from the Run menu, you can now modify **COMMAND$** from within QuickBASIC itself. This feature lets you test different command-line arguments quickly and conveniently.

2.4.2 The qb Command

The qb command starts QuickBASIC. This command also allows you to set a number of options, as shown by the following syntax:

qb [[**/run**] [*sourcefile*] [**/b**] [**/g**] [**/h**] [**/c:***buffersize*]
[**/l** [*libraryname*]] [**/mbf**] [**/ah**] [**/cmd** *string*]]

The following list describes the arguments to the qb command:

Argument	Description
/run *sourcefile*	Causes QuickBASIC to load and run *sourcefile* before displaying it.
sourcefile	Names the file to be loaded when QuickBASIC starts. See Section 2.4.6, "QuickBASIC File-Naming Conventions," for more information.
/b	Allows use of a composite (black-and-white) monitor with a color graphics card. This option also displays QuickBASIC in black and white if you have a color monitor. Note that you don't need to use the /b option if you have a monochrome adapter card; QuickBASIC can detect what type of card you have, but not what type of monitor.
/g	Modifies the way QuickBASIC updates your screen. If you have a CGA, QuickBASIC automatically adjusts the way it updates the screen to accommodate your hardware. However, on some machines it is possible to update the screen faster than the way QuickBASIC does it. The /g option causes QuickBASIC to update the screen as fast as possible. If you see "snow" (dots flickering on the screen) when you update large areas of the screen (for example, with PGUP), then your hardware cannot handle the fast speed. If you prefer a clear screen to the fast response, you should restart QuickBASIC without the /g option. This option affects only machines with a CGA.

/h	Displays the highest resolution possible on your hardware. For example, if you have an EGA, the /h option causes QuickBASIC to display 43 lines and 80 columns of text.
/c:*buffersize*	Sets the size of the buffer receiving remote data. (The transmission buffer is allocated 128 bytes and cannot be changed from the qb command line.) This option has an effect only if an asynchronous communications card is present. The default buffer size is 512 bytes; the maximum size is 32,767 bytes. (See Chapter 3, "File and Device I/O," in *Programming in BASIC: Selected Topics*, and the entry for **OPEN COM** in the *BASIC Language Reference*, for more information about asynchronous communication.)
/l *libraryname*	Loads the Quick library that is specified by *libraryname*. If *libraryname* is not specified, the default Quick library, QB.QLB, is loaded. See Chapter 8, "Quick Libraries," for more information.
/mbf	Causes the QuickBASIC conversion functions to treat IEEE-format numbers as Microsoft Binary format numbers. The intrinsic functions **MKS$**, **MKD$**, **CVS**, and **CVD** behave the same as **MKSMBF$**, **MKDMBF$**, **CVSMBF**, and **CVDMBF**, respectively. See the entries for these functions in the *BASIC Language Reference* for more information.
/ah	Allows dynamic arrays of records, fixed-length strings, and numeric data to be larger than 64K each. If this option is not specified, the maximum size for each array is 64K. This option has no effect on the way data items are passed to procedures.
/cmd *string*	Passes *string* to the **COMMAND$** function. This option must be the last option on the line. See the entry for the **COMMAND$** function in the *BASIC Language Reference* for further information.

2.4.3 Customizing the QuickBASIC Screen

You can customize the QuickBASIC screen to suit your individual taste with the Options command from the View menu. Choosing the Options command brings up a dialog box that lets you set things like the background color, foreground color, whether or not you want scroll bars to appear at the bottom and side of the screen, and so on. You can even change the number of spaces the TAB key advances the cursor.

For example, the following steps show you how to change the background color for QuickBASIC:

1. Press ALT and then press V to open the View menu.
2. Press O to display the Options dialog box, shown in Figure 2.8.

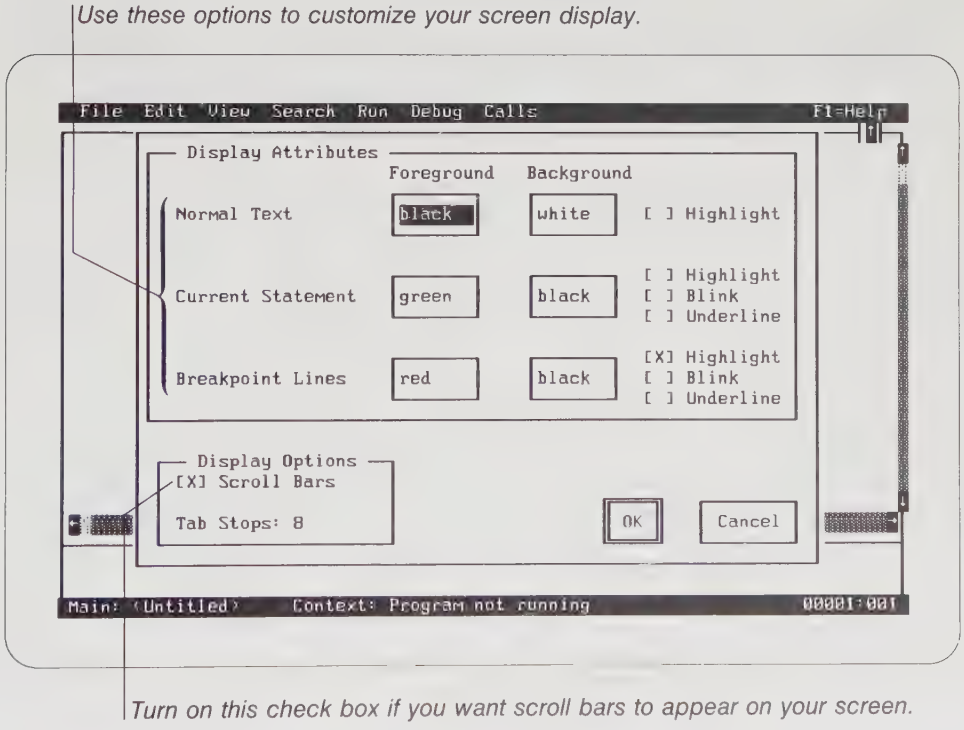


Figure 2.8 Options Dialog Box

3. Press the TAB key to move among options and boxes under the "Background" heading.
4. Use the UP and DOWN direction keys to select the background color from the colors in the boxes.
5. Press ENTER when you have set all desired options.

Experiment to get the screen configuration you want; you can set more than one option at a time. (See Section 3.3 for more information on how to select items in dialog boxes.)

2.4.4 The QB.INI File

When you exit QuickBASIC, the settings you have selected in the Options command from the View menu are automatically saved in a file called QB.INI. As long as you are in the directory where QB.INI is located, the next time you start QuickBASIC, these settings are automatically invoked. (Alternatively, you could move the QB.INI file to a directory listed in your PATH environment variable. This way, you get your customized settings regardless of the directory you're in when you start QuickBASIC. See Section 1.7.1 to learn how to set the PATH environment variable.)

2.4.5 Getting Help from QuickBASIC

QuickBASIC can help you if you have a question about the right shortcut keys to use for an editing or debugging task, or if you want to check the syntax for a statement or function.

QuickBASIC has three kinds of on-line Help:

1. Help for QuickBASIC's menu commands.

   This consists of a brief description of an open menu's currently highlighted choice (see Figure 2.9). You don't have to press any keys or make any menu choices to get this description; it appears on the status bar at the bottom of your screen whenever a menu is open.

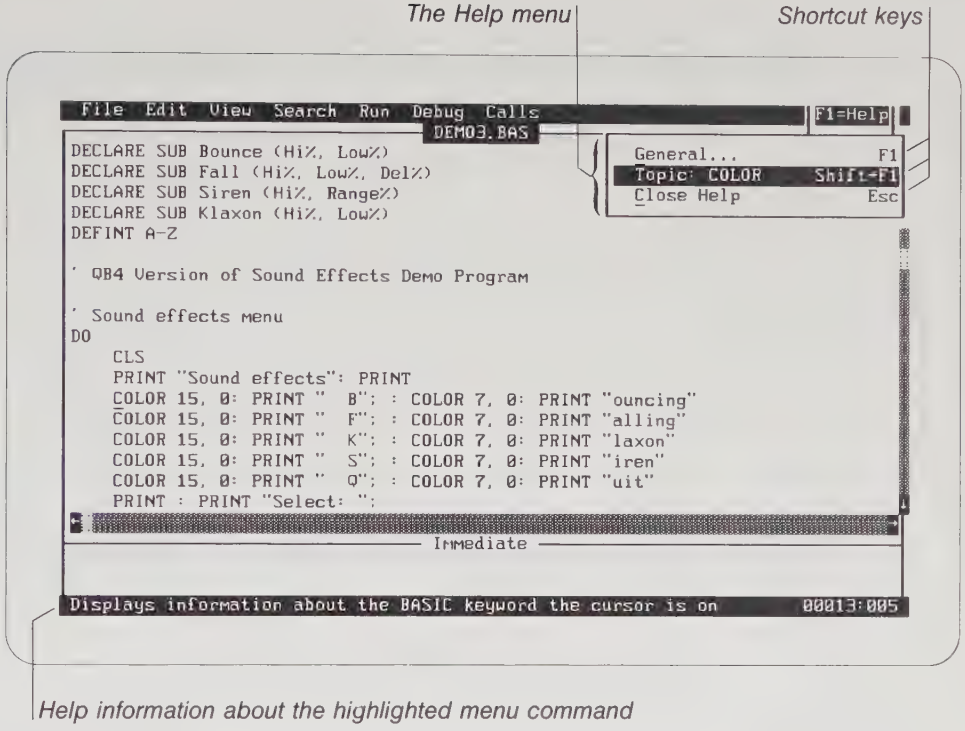


Figure 2.9 Help for Menu Commands

2. General Help.

   General Help is available through the Help menu (press ALT+H) on the extreme right side of the menu bar (see Figure 2.9 above). It describes such things as editing procedures and shortcut key sequences (see Figure 2.10). (You can also press the F1 key to get general Help.)

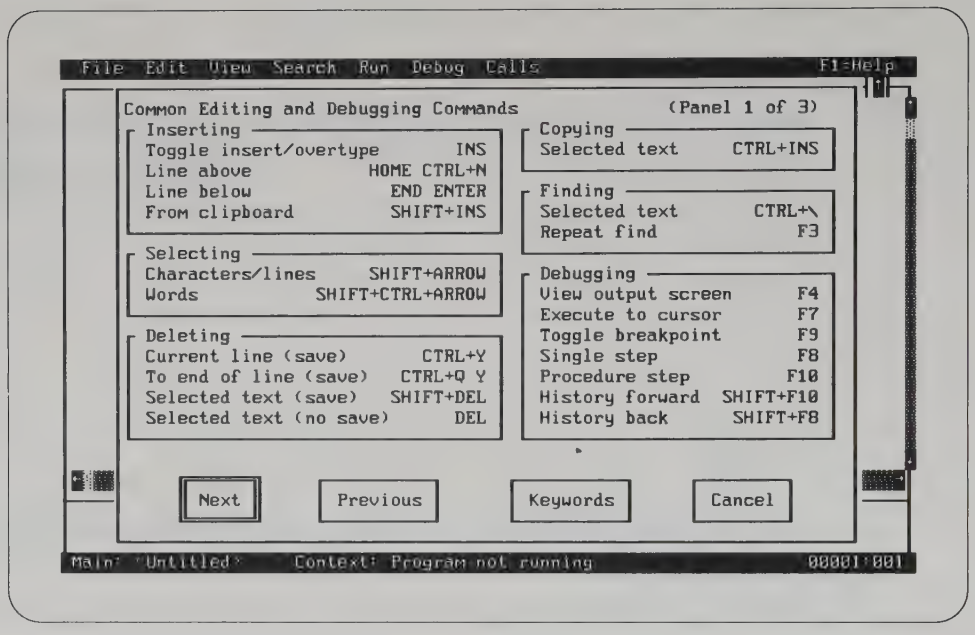


Figure 2.10 General Help

3. Context-sensitive Help.

   Context-sensitive help answers your questions about BASIC syntax. To use context-sensitive Help, place the cursor on the keyword you want to know about (or on the space immediately after the keyword), then press SHIFT+F1 to see a synopsis of the statement's syntax.

   Context-sensitive Help is also available from the Topics command on the Help menu.

2.4.6 QuickBASIC File-Naming Conventions

QuickBASIC makes certain assumptions about the files you specify. This section describes these assumptions and other rules you should follow when naming and specifying files.

2.4.6.1 Path Names

Any file name can contain a full or partial path name. A full path name starts with the drive name; a partial path name has one or more directory names before the file name, but does not include a drive name. A full path name allows you to specify files on different drives and in different directories.

■ **Examples**

```
a:progs\graphic\circle.bas      ' a full path name
\graphic\circle.bas             ' a partial path name
..\graphic\circle.bas           ' a partial path name
```

2.4.6.2 Uppercase and Lowercase Letters

Within QuickBASIC, you can use any combination of uppercase and lowercase letters for filenames, as in the following examples, all of which refer to the same file:

■ **Examples**

```
abcde.bas
ABCDE.BAS
aBcDe.Bas
```

2.4.6.3 File-Name Extensions

A DOS file name has two parts: the "base name," which is everything up to but not including the period (.), and the "extension" which includes the period and up to three characters following the period. In general, the extension identifies the type of file, such as a BASIC source file or an executable file.

The default file extensions are as follows:

Extension	Explanation
.BAS	A BASIC source file
.EXE	An executable file (see Chapter 6, "Creating and Running QuickBASIC Programs," and Chapter 9, "Compiling and Linking from DOS," for more information)
.QLB	A Quick-library file (see Chapter 8, "Quick Libraries," for more information)
.LIB	A stand-alone library file (see Chapter 8, "Quick Libraries," and Chapter 9, "Compiling and Linking from DOS," for more information)

■ **Examples**

```
circle.bas      ' a BASIC source file
circle.exe      ' an executable program file
circle.qlb      ' a Quick library
circle.lib      ' a stand-alone library
```

2.4.6.4 Specifying File Names

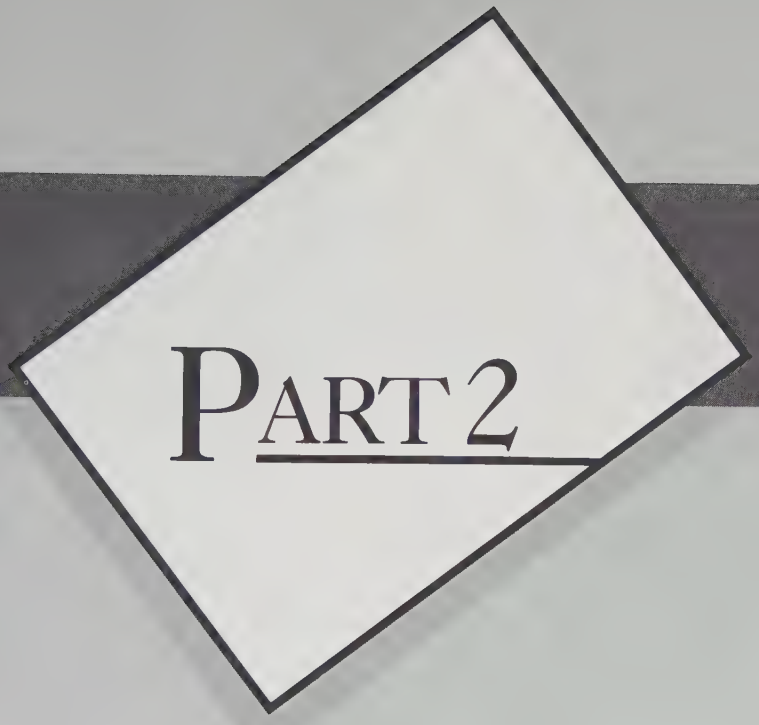
QuickBASIC makes the following assumptions about file names:

- If no extension is specified, QuickBASIC assumes the extension .BAS when you use the qb command with the *sourcefile* argument, and when you start new programs and open files.
- When you are creating an executable file, QuickBASIC assumes it will have the extension .EXE.
- When you use the /l (load Quick library) option, QuickBASIC assumes the .QLB extension.

Note

If you want to specify a file that has no extension, place a period at the end of the file name. For example, if you are starting QuickBASIC and you want to load a file called MYPROG, enter the following to prevent QuickBASIC from looking for MYPROG.BAS:

```
MYPROG.
```



THE QUICKBASIC PROGRAM DEVELOPMENT ENVIRONMENT

CHAPTERS

3 Getting Around in the QuickBASIC Environment....63

4 Managing Source Files....83

5 Editing....111

6 Creating and Running QuickBASIC Programs....137

7 Debugging While You Program....163

PART 2 ◇ THE QUICKBASIC PROGRAM DEVELOPMENT ENVIRONMENT

This part of the manual contains detailed instructions on using the QuickBASIC program development environment, including its screens, menus, and other features. It explains how to load, save, and manage QuickBASIC program files and modules; how to use QuickBASIC's advanced editing features; and how to create executable files and multiple-module programs. Also, it describes how to analyze and debug your programs, even while they're running.

CHAPTER 3

GETTING AROUND IN THE QUICKBASIC ENVIRONMENT

3.1 The QuickBASIC Screen 66

3.2 Opening Menus and Choosing Commands 68
- 3.2.1 Keyboard Technique 69
- 3.2.2 Mouse Technique 70
- 3.2.3 Using Shortcut Keys 71

3.3 Using Dialog Boxes 72

3.4 Using Windows 76
- 3.4.1 Window Types 76
- 3.4.2 Splitting the View Window 77
- 3.4.3 Changing the Active Window 78
- 3.4.4 Changing Window Size 78
- 3.4.5 Scrolling in the Active Window 79
- 3.4.6 The Immediate Window 79

3.5 Returning Temporarily to DOS (DOS Shell) 80

3.6 Exiting QuickBASIC (Exit) 81

This chapter introduces the QuickBASIC programming environment and illustrates many of its features. You don't have to understand all of QuickBASIC's features before you start writing programs with it, so this chapter gives you the necessary basics to get you going.

In this chapter, you will learn how to

- Choose commands from menus.
- Choose items from dialog boxes.
- Scroll listings.
- Select text in a dialog box or window.
- Change window size.
- Enter and directly execute one or more BASIC statements.

3.1 The QuickBASIC Screen

When you start QuickBASIC without a file-name argument, your screen appears as shown in Figure 3.1.

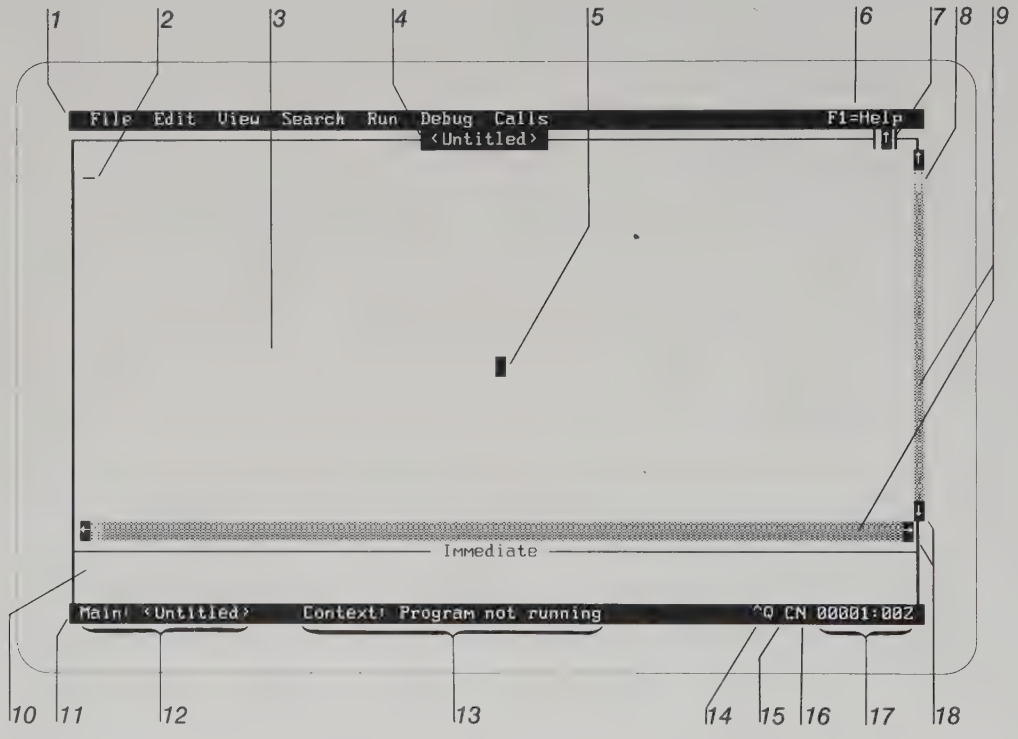


Figure 3.1 QuickBASIC Screen

The following list explains the parts of the screen and refers to Figure 3.1:

1. Menu bar. Names each menu. Highlighted letter indicates shortcut key.
2. Cursor. Flashing underscore that shows where the text you type will appear.
3. View window. A window into your program text. (Can be split into two windows.)
4. Title bar. Names file (and procedure, if applicable) currently shown in window below.
5. Mouse pointer. Shows current on-screen location of mouse (mouse only).
6. F1=Help. Indicates key to press to receive on-line Help.
7. Maximize box. Expands the active window to fill the screen (for use with the mouse only).
8. Scroll box. Shows cursor's relative position within the file or procedure.
9. Scroll bars. Scroll text in the currently active window (mouse only).
10. Immediate window. Area where BASIC statements can be directly executed.
11. Status bar. Contains text position, keyboard status, and program information.
12. Main. Indicates the name of the main module of your program.
13. Context. Names the module (and procedure, if appropriate) containing the statement that will be executed next.
14. ^Q. Appears when this WordStar®-style command sequence is entered. ^K appears here when you are setting a place marker, and ^P appears here when you are entering a literal control character (see Chapter 5, "Editing," for more information).
15. C. Appears when the CAPSLOCK key is toggled on.
16. N. Appears when the NUMLOCK key is toggled on.
17. Line and column counters. Give current position of cursor within the text in the active window.
18. Scroll arrows. Scroll the text one character or one line at a time (mouse only).

At this point, you can begin to write your program, or you can ask QuickBASIC for help. To use the QuickBASIC on-line Help system, press F1 to invoke the first of a series of Help screens explaining key assignments and BASIC syntax.

To clear a Help screen, press the ESC key. Pressing ESC also clears menus, dialog boxes, and error messages from the QuickBASIC screen. (See Section 2.4.5 for more information about QuickBASIC's on-line Help.)

3.2 Opening Menus and Choosing Commands

QuickBASIC commands are organized in menus on the menu bar. Figure 3.2 shows one of these, the File menu.

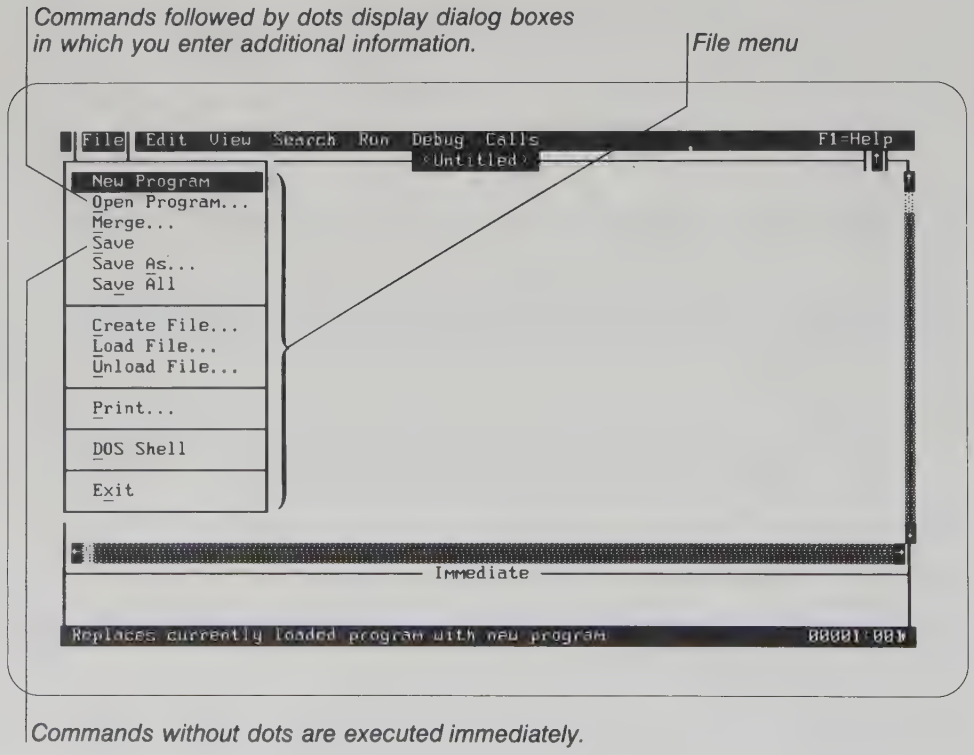


Figure 3.2 The File Menu

The QuickBASIC environment is designed for fast, simple operation. You can do most operations in QuickBASIC with either of the following techniques, so choose the one that is most convenient for you:

- Open menus and choose commands either with the keyboard or with the mouse, if you have one.
- Execute commands directly with "shortcut" keys and key combinations.

3.2.1 Keyboard Technique

Using the keyboard, you can choose any command in a QuickBASIC menu by following these steps:

1. Open the menu.

   Press and release the ALT key, then press the first letter of the menu's name. This "selects" (highlights) the entire name.

Note

When working with menus, you don't have to hold the ALT key down. The ALT key is a "toggle," so if you press it again, the highlight disappears. The "press and release" feature of the ALT key is true only with menus. When using key combinations in other circumstances, you must hold down the first key while pressing the rest of the sequence.

If you open the wrong menu, take one of the following actions:

- Close the menu by pressing ESC, then try step 1 again.
- Move to another menu by pressing the LEFT or RIGHT direction keys.

2. Choose the command.

   Menu names and command names always contain a highlighted letter. On color monitors, this letter is a different color from the other letters in the name; on monochrome monitors, it is underlined as shown in Figure 3.2. Take one of the following actions to move to the desired command:

   - Press the key corresponding to the highlighted (or underlined) letter.
   - Use the UP or DOWN direction keys to move the highlight to the command you want, then press ENTER.

   If no letter in a command is highlighted, you cannot choose that command until you take some other action. For example, if you are editing a program, but you haven't selected any text for copying or deleting, then commands like Cut and Copy in the Edit menu can have no effect, and so appear without any highlighted letter. As soon as you select some text, you can use the Cut and Copy commands. (See Chapter 5, "Editing," to learn how to select text for copying or deleting.)

   Some commands, such as the New Program command on the File menu, take effect immediately and cannot be canceled once chosen. However, others cause QuickBASIC to open a dialog box where you can supply additional information. To cancel such a command, press ESC. See Section 3.3 for a discussion of dialog boxes.

3.2.2 Mouse Technique

To choose commands with the mouse

1. Point to the menu name, and then click the left mouse button.
2. Point to the command, and then click the left mouse button.

To cancel a command with the mouse

- Move the pointer off the menu, and then click the left button.

Note

Use only the left mouse button to choose QuickBASIC menus or commands. Use the right mouse button when debugging your programs; pressing the right button always executes your program up to the line on which the mouse pointer rests.

Table 3.1 explains mouse techniques for accomplishing other on-screen tasks discussed in this chapter.

Table 3.1

Mouse Commands

Task	Mouse Technique
Making a window active (see Section 3.4.3)	Click anywhere in the window.
Expanding or shrinking an active window (see Section 3.4.4)	Move the mouse cursor to the title bar and drag the title bar up or down.
Expanding the active window to occupy the full screen (see Section 3.4.4)	Click the maximize box (|↑|), at the right of the title bar, or double click the mouse anywhere on the title bar.
Scrolling text (see Section 3.4.5)	Point at the scroll box and drag it to a position on the bar that corresponds to the general location in the file you want. To scroll a page at a time, place the mouse pointer in the scroll bar between the scroll box and the top or bottom of the scroll bar and click the left button. To scroll one line or one character at a time, click the scroll arrows at either end of the scroll bars.

See Appendix D, "Command Summary," for more information on mouse commands.

3.2.3 Using Shortcut Keys

Shortcut keys for a menu command are always listed on the menu, next to the command. As you become familiar with QuickBASIC, you'll learn many such shortcut sequences. For example, to run a program, you can choose the Start command from the Run menu, or you can bypass the menu and use the shortcut key combination SHIFT+F5 (see Figure 3.3). See Section D.2, "Menu Commands and Shortcut Keys," in Appendix D, "Command Summary," for a list of QuickBASIC's shortcut keys.

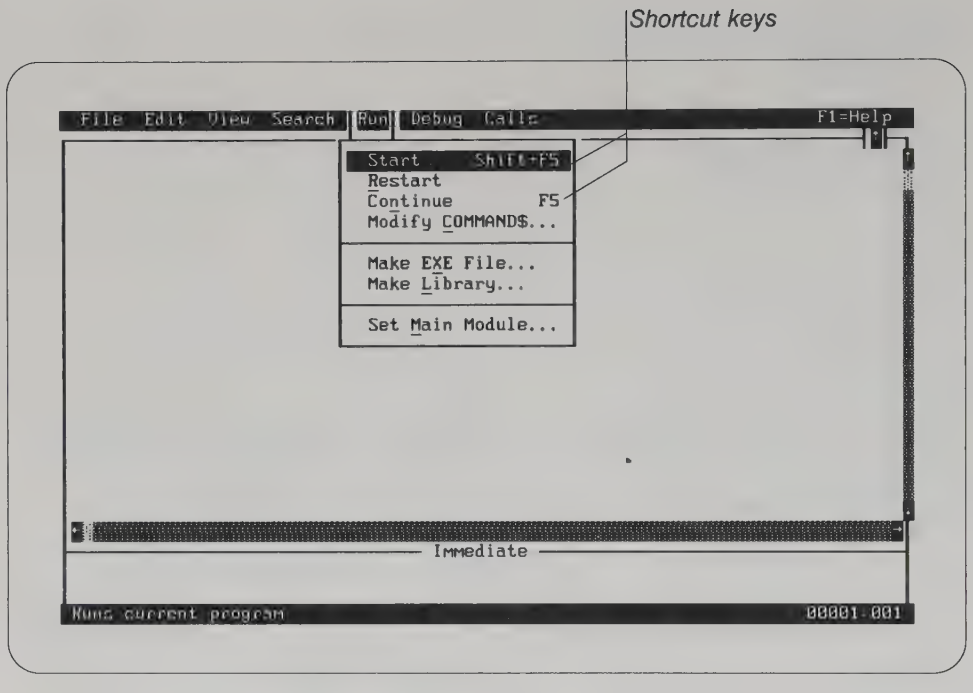


Figure 3.3 Shortcut Keys

3.3 Using Dialog Boxes

QuickBASIC displays a dialog box when it needs additional information from you before it can carry out an action. For example, a dialog box might

- Prompt you for the name of a file
- Display a list of options for you to choose
- Ask you to verify or cancel a command

In this section, the File menu's Load File dialog box and the View menu's Options dialog box, shown in Figures 3.4 and 3.5, illustrate the parts of a dialog box.

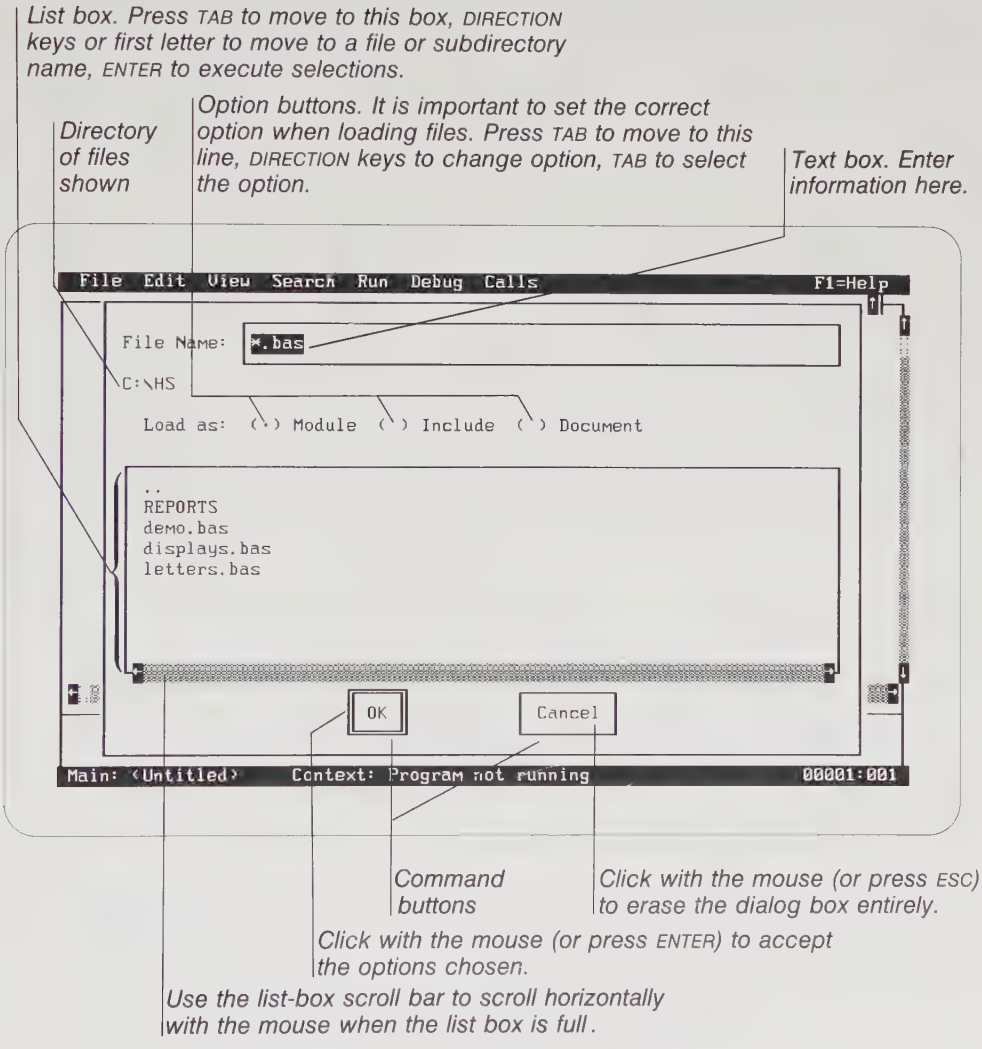


Figure 3.4 Load File Dialog Box

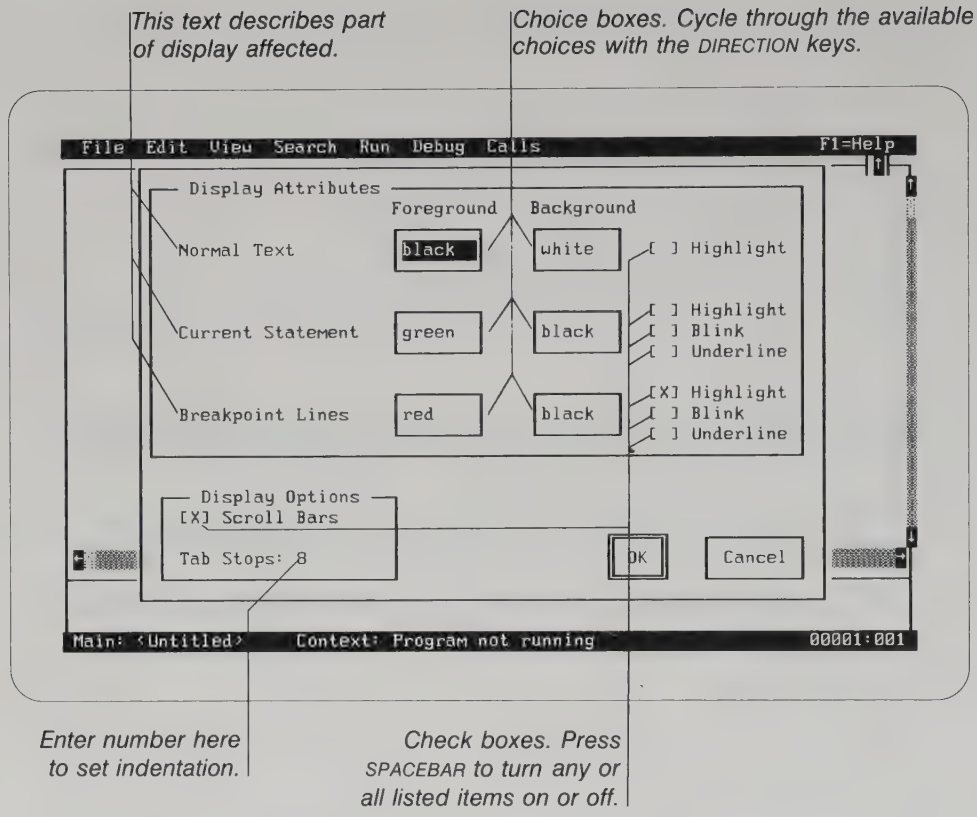


Figure 3.5 Options Dialog Box

When you work with QuickBASIC, you need to select the object to be affected by the next command or action. The selected object is usually highlighted in reverse video.

The area of the dialog box affected by your next action has the "input focus." The blinking cursor indicates where the input focus is. To move the input focus within a dialog box

- Press the TAB key, or hold down the ALT key while pressing the key corresponding to the highlighted letter in the item you want to select.

  For example, in the File menu's Load File dialog box, you can select the option for loading a file (Module, Include, or Document) by holding down the ALT key while pressing either M, I, or D.

Note that with dialog boxes you must keep the ALT key pressed while pressing the highlighted letter. This is slightly different from menu commands, which don't require you to hold the ALT key down while pressing the highlighted letter.

The following list shows what having the input focus means for different components in a dialog box:

Input Focus	Meaning
Text box	Shows where typed text will appear.
Options box	Shows the currently chosen option.
Check box	Toggles an option on or off. Press SPACEBAR, or hold down ALT while pressing the highlighted letter. When an option is on, an X appears in the check box; when it is off, the check box is empty.
Option button	Toggles an option on or off. Use the DIRECTION keys, or hold down ALT while pressing the highlighted letter or number.
Command button	Activates a command. Press either SPACEBAR or ENTER. Note that any time you press ENTER, the command button that is currently surrounded with a double outline takes effect, so there is no need to move to the OK button when you are finished with a dialog box.

Note

Clicking an item with a mouse affects that item, regardless of the location of the input focus.

Double-clicking an item in a list box not only selects the item, it also confirms all previous choices and closes the dialog box.

When a dialog box opens, it often contains information on options chosen earlier. For example, the Options dialog box opens with the options set from the last time you made a change.

3.4 Using Windows

This section introduces QuickBASIC's windows, and tells you how to

- Change the active window.
- Move different parts of programs in and out of an active window.
- Change the size of a window.
- Scroll text in a window.

See Section D.5 in Appendix D, "Command Summary," for a complete list of the window-control commands.

3.4.1 Window Types

The following list describes the types of windows in QuickBASIC:

Window	Description
View	The window that appears at the top of the screen when you start QuickBASIC (see Figure 3.1). When you load a program, the code that is outside **FUNCTION** or **SUB** procedures—known as the "module-level" code—appears in the View window. The View menu contains commands that let you easily move different parts of your program in and out of View windows. For example, to see a **FUNCTION** or **SUB** procedure in the View window, choose the SUBs command from the View menu, then select the procedure from the dialog box.
Immediate	The window that appears at the bottom of the screen when you start QuickBASIC. (See Sections 3.4.6 and 6.1.5 for more information on the Immediate window.)
Watch	The window you can open at the top of the screen when you are debugging. It displays variable values as your program is running. (See Section 7.3.4.1 for more information on using the Watch window for debugging.)

Since the View window can be split in two, you can have a total of four windows open at one time, as shown in Figure 3.6.

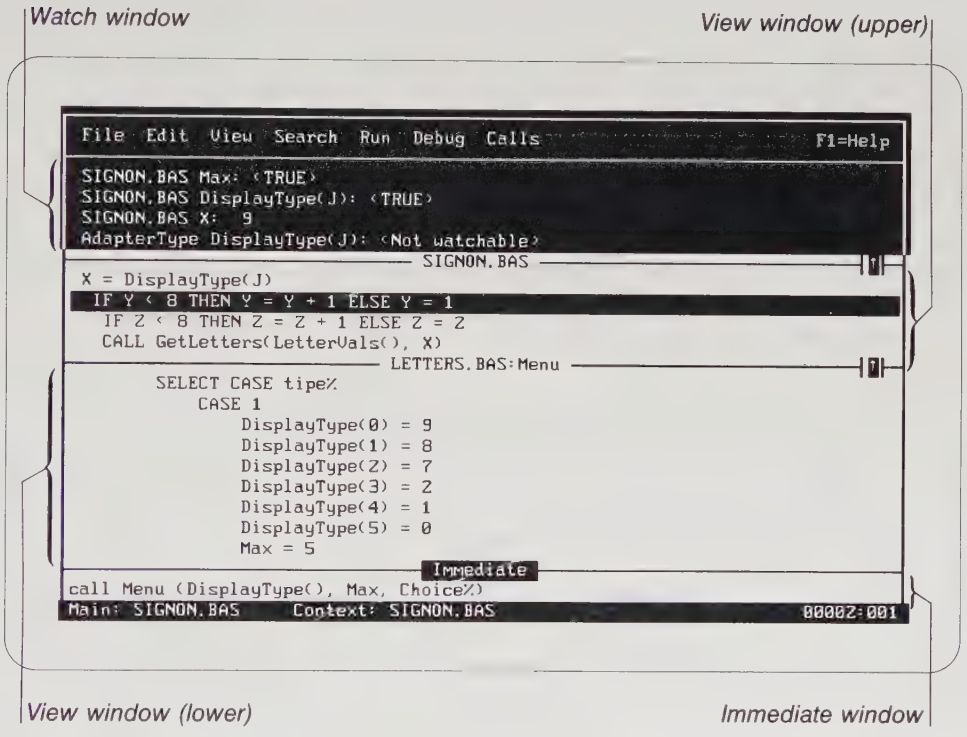


Figure 3.6 Screen with Four Windows Open

3.4.2 Splitting the View Window

You can split the View window horizontally. This allows you to view or edit two parts of a program simultaneously.

To split the View window

1. Press ALT+V to open the View menu.
2. Press P (the highlighted letter in the Split command).

The screen is now divided into two parts. You can manipulate the screen in any the following three ways:

1. Press F6 to move from the top half to the bottom half.
2. Press SHIFT+F6 to move from the bottom to the top.
3. Press ALT+V, then press P to restore the screen. This clears the bottom window.

3.4.3 Changing the Active Window

The window that contains the cursor is referred to as the "active" window; that is, it is the place where text entered from the keyboard appears.

To make another window active

- Press F6 or SHIFT+F6.

  Using F6 lets you cycle downward through the windows on the screen, making each active in turn; SHIFT+F6 lets you cycle upward.

3.4.4 Changing Window Size

You can increase or decrease the size of a window by one line at a time, or expand it to fill the whole screen. To change the size of a window, first make it the active window. Then, use the following key combinations to increase or decrease its size (hold down the first key while pressing the second):

Key Combination	Result
ALT+PLUS	Expands the active window one line.
ALT+MINUS	Shrinks the active window one line.
CTRL+F10	Expands the active window to fill the screen, or, if it already occupied the whole screen, returns it to its former size. If the View window is split in two when you press CTRL+F10, pressing CTRL+F10 again restores both windows to the screen.

3.4.5 Scrolling in the Active Window

Often, the part of a file you want to work on is too wide—or too long—to fit within the boundaries of a View window. To look at the parts that won't fit, you can move the text up, down, right, or left. This process is known as "scrolling."

Once you have reached a window's edge, press the appropriate DIRECTION key to start scrolling. For example, to scroll right one character at a time, go to the rightmost character on the screen and keep pressing the RIGHT key.

Refer to Table 3.2 for information on scrolling more than one character at a time. The rightmost column in this table shows key strokes that you may find more convenient if you prefer WordStar-style commands.

Table 3.2
Scrolling

Scrolling Action	Keystrokes	WordStar-Style Equivalents
First indentation level	HOME	---
Column 1 of current line	---	CTRL+Q S
End of line	END	CTRL+Q D
Page up	PGUP	CTRL+R
Page down	PGDN	CTRL+C
Left one window	CTRL+PGUP	---
Right one window	CTRL+PGDN	---
Beginning of file	CTRL+HOME	CTRL+Q R
End of file	CTRL+END	CTRL+Q C

You can also set place markers anywhere in your program and jump between them while editing. See Section 5.9 for more information.

3.4.6 The Immediate Window

The bottom window on the initial QuickBASIC screen, labeled with the word Immediate in the title bar, is for immediate execution of QuickBASIC statements. For example, if you want to use a **LOCATE** statement to place the cursor at a certain point on the output screen, make the Immediate window active by pressing F6, then type the following line (here row is chosen to be 4, and column is 5):

```
cls : locate 4,5 : print "test"
```

As soon as you press ENTER you will see the output. Press any key to return to the Immediate window, edit the line to adjust the row and column arguments, and repeat the cycle until the word "test" appears exactly where you want it. When you get the word test to appear where you want it, you can copy the statement into your program.

You can enter up to 10 separate lines in the Immediate window, then move among them with the DIRECTION keys. Each line is limited to 256 characters. Multiple statements are permitted on a line, but each complete statement must be separated from the next by a colon (:), as shown in the example above. Whenever you place the cursor on a line and press ENTER, only the statements on that line are executed. See Section 6.1.5 for further discussion of the Immediate window.

3.5 Returning Temporarily to DOS (DOS Shell)

The DOS Shell command on the File menu lets you return temporarily to the DOS command level, where you can execute other programs and DOS commands. QuickBASIC remains in memory so you can return to the same place in your program, without reloading it.

QuickBASIC needs to find the COMMAND.COM file before it can execute the Shell command. QuickBASIC looks for COMMAND.COM first in the directory specified in the COMSPEC environment variable, then in the current directory. See your DOS documentation for more information about COMMAND.COM and COMSPEC.

To return to QuickBASIC from the DOS command level

1. Type

   ```
   exit
   ```

2. Press ENTER.

The QuickBASIC screen reappears just as you left it.

3.6 Exiting QuickBASIC (Exit)

The Exit command on the File menu removes QuickBASIC from memory and returns you to the DOS prompt.

When you exit from a new or modified program that has not been saved, QuickBASIC displays a dialog box asking if you want to save the program. Use one of these three procedures:

1. To save the program in its current state, press ENTER.

   If this is a new program that you haven't named yet, QuickBASIC displays the Save As dialog box. See Section 4.2.2 for more information on saving files.

2. To return to the system without saving the file in its current state, press N or move the input focus to the No button, then press ENTER or SPACEBAR.

3. To return to the QuickBASIC environment (that is, not exit after all) press ESC.

CHAPTER 4

MANAGING SOURCE FILES

4.1 Classifying Files in QuickBASIC 86

4.2 Programs 88
- 4.2.1 Loading and Listing Programs (Open Program) 88
  - 4.2.1.1 Specifying a File 90
  - 4.2.1.2 Listing Directory Contents 90
- 4.2.2 Saving a Program (Save, Save All, Save As) 92

4.3 Modules 94
- 4.3.1 Creating and Loading Modules (Load File) ... 95
- 4.3.2 Displaying Multiple Modules 98
- 4.3.3 Saving Multiple Modules (Save All) 99

4.4 Changing the Main Module (Set Main Module) 99
- 4.4.1 The .MAK File 100
- 4.4.2 Removing Modules from a Program (Unload File) 101

4.5 Include Files 101
- 4.5.1 Creating and Loading Include Files (Create File) 102
- 4.5.2 Viewing and Editing Include Files (Include File, Included Lines) 103
- 4.5.3 Nesting Include Files 105

4.6 Documents 105

4.7 Merging the Contents of Two Files (Merge) 107

4.8 Printing Files (Print) 109

This chapter explains how to manipulate files in the QuickBASIC environment. When you finish this chapter you will understand the difference between programs, modules, include files, and documents. You will also know how to accomplish the following:

- Load, list, and save programs.
- Create modules.
- Understand the difference between a program's main module and its other modules.
- Load and unload modules from a program.
- View and edit include files.
- Load ordinary documents into QuickBASIC for viewing and editing.
- Merge the contents of two files.
- Print files.

4.1 Classifying Files in QuickBASIC

The File menu, shown in Figure 4.1, is your center for working with files in QuickBASIC. From the File menu, you can create new files, load existing files into QuickBASIC, modify files, or remove files from your program.

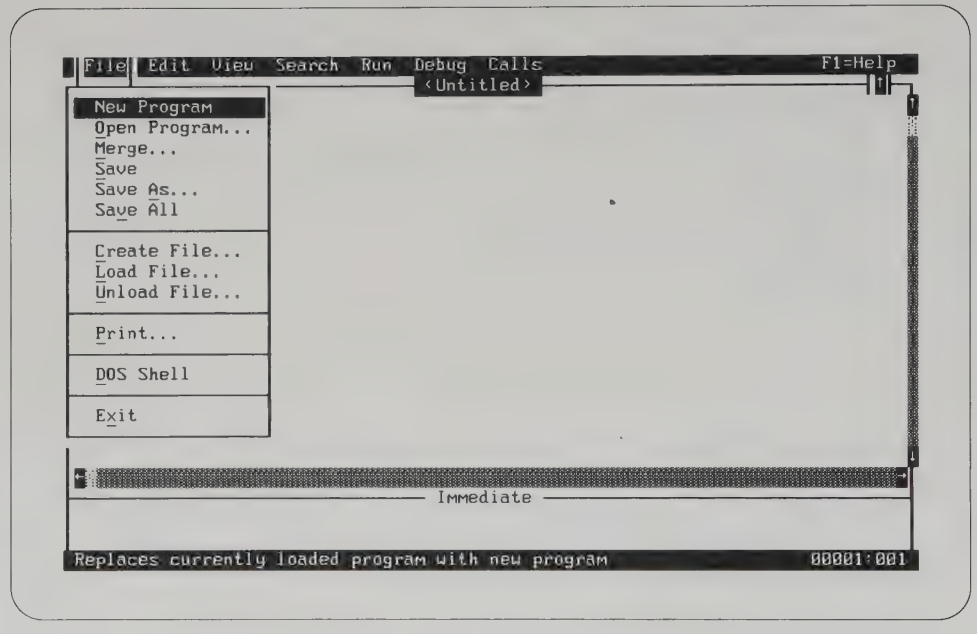


Figure 4.1 The File Menu

What you can do with a file depends greatly on how you load or create the file. QuickBASIC has four ways of dealing with files, and the following list briefly describes each of these ways (see Sections 4.2–4.6 for details on working with each kind of file):

- **Program**

  A program contains one or more BASIC statements that are translated by QuickBASIC into instructions for your computer.

  When you first start QuickBASIC and begin typing in the View window, QuickBASIC assumes that you are giving it program instructions, so it turns on its "smart editor" features. When the smart editor is on, each line of text is checked for proper BASIC syntax as soon as you press ENTER, and you see an error message if you don't get it right. The smart editor is also turned on when you

choose New Program or Open Program from the File menu, or when you start QuickBASIC with the name of a program you want to work on.

If you have programmed mostly in BASICA, you are probably used to thinking of one program as being synonymous with one file. If your programs tend to be small and simple, you can continue to program this way with QuickBASIC. See Section 4.2 for more information on loading, listing, and saving programs.

- **Module**

  A module is a discrete, self-contained program component. A program always consists of at least one module. When you make a program out of several modules, each module is saved on disk as a separate file when you save the program. Modules are handy if you have longer programs or if you use the same lines of code over and over in different programs.

  You don't have to use QuickBASIC's multiple-module feature when you write a program. You can still keep everything in one file, and when you save the program it is saved as one file. However, breaking a large program into separate modules makes it much easier for you to share common procedures among many different programs. Multiple-module programs can also be much larger, since each module can be 64K.

  Once a module is debugged, you can use it as a building block in many different programs. If you find yourself using the same **SUB** or **FUNCTION** procedures in different programs, you can put those procedures in a common module, and then load this module into your programs as needed. (Indeed, it is a short, easy step to turn such a module into a Quick library, so you won't even have to load the module each time you want to call its procedures. See Chapter 8, "Quick Libraries," for more information.)

  See Sections 4.3–4.4, and Chapter 6, "Creating and Running QuickBASIC Programs," to learn more about using modules.

- **Include**

  An "include" file is a file whose statements are compiled into your program when QuickBASIC encounters the file's name following a **$INCLUDE** metacommand. For example, the following line assimilates the contents of the DECLARES.BI include file:

```
' $INCLUDE: 'DECLARES.BI'
```

  Files opened as include files are subject to syntax checking and formatting. Note that QuickBASIC places some restrictions on the contents of include files.

  See Section 4.5 for more information on include files.

- **Document**

  A file opened as a document can be edited in QuickBASIC just as in a normal word processor. QuickBASIC does not consider a document a BASIC program, so it doesn't check documents for syntax, nor does it format text (for instance, convert BASIC keywords to capital letters). If you load a multiple-module program as a document, only the main module is loaded (see Section 4.4.1, "The .MAK File," for more information).

Note

Once you have loaded any of the preceding kinds of files into QuickBASIC, you can see the file's name in the list box of the View menu's SUBs dialog box.

While a file's name appears in this list box, you cannot load it again, even with a different classification. For example, if you load a file as a document, you cannot load it again as an include file until you unload the copy classified as a document.

4.2 Programs

As mentioned in Section 4.1, "Classifying Files in QuickBASIC," you are automatically working on a program when you start typing text in QuickBASIC's initial View window. To unload the current program and begin a completely new one, choose the New Program command from the File menu. To open an existing program for viewing or editing, choose the Open Program command from the File menu. Both these commands clear all previously loaded files.

4.2.1 Loading and Listing Programs (Open Program)

The Open Program command on the File menu loads a program from disk into memory. When a program uses include files, the include files are compiled but not displayed. (See Section 4.5 for information about displaying and editing include files.) When a program consists of several modules, all of the modules belonging to the program are loaded.

You can also use the Open Program command to list the files and directories on your system (see Section 4.2.1.1, "Specifying a File").

When you choose the Open Program command from the File menu, the dialog box shown in Figure 4.2 opens:

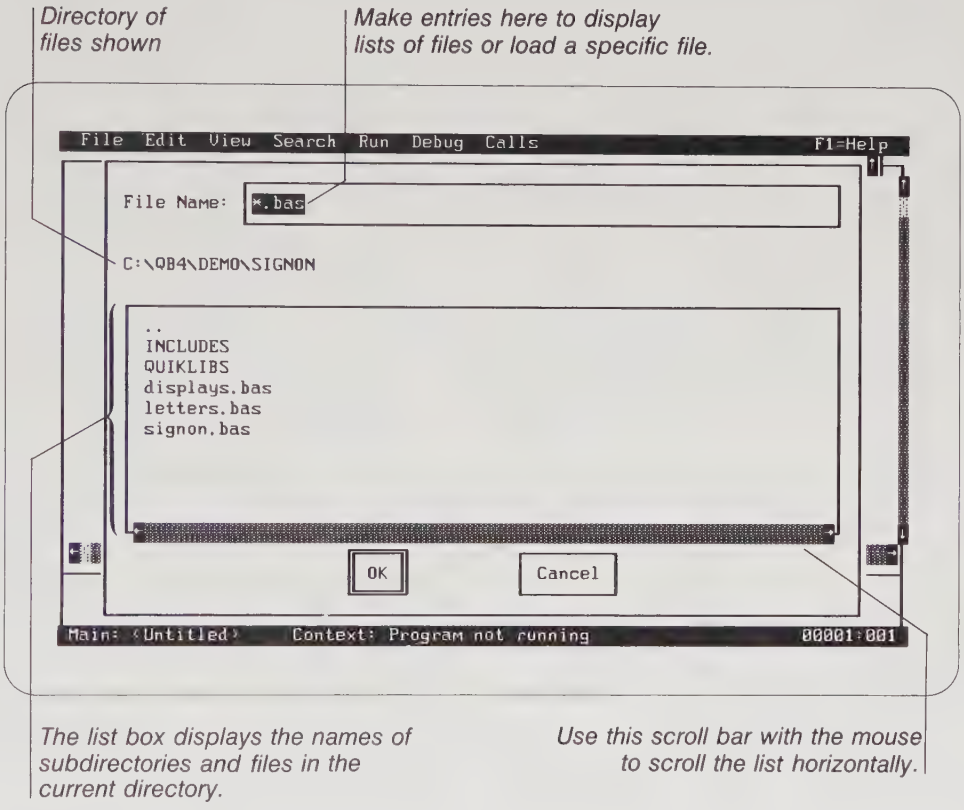


Figure 4.2 Open Program Dialog Box

Directory and file names are displayed in columns in the list box. Names of directories appear in capital letters at the top of the list. If all the names don't fit in the box, use one of the DIRECTION keys to move the cursor to the edge of the box and keep pressing it to scroll the list. Note that you can only scroll the list box left and right.

4.2.1.1 Specifying a File

The Open Program dialog box provides two ways to specify the file you want to load.

1. The first method is as follows:
   - Type the name of the program in the text box.

     QuickBASIC assumes the .BAS extension for all file names that don't have extensions. If you want to load a file that has no extension, type a period (.) immediately after the file name.

2. The following is a second method for specifying a file:

   a. Press the TAB key to move to the list box, then use one of the following techniques to move through the list box until the file you want to load is highlighted:

      — Use the DIRECTION keys.

      — Press the first letter of the name of the file you want to load. For example, if the files

      ```
      PROG1.BAS
      PROG2.BAS
      PROG3.BAS
      ```

      are in the current directory, then pressing "p" three times moves the input focus to PROG3.BAS.

   b. Press ENTER.

      The text of the program appears in a window with the file name in the title bar, ready for you to edit or run.

Note

To open a program with the mouse, double click the file name in the list box.

4.2.1.2 Listing Directory Contents

Use the Open Program dialog box to list the contents of any directory on your system. When you enter a directory name in the text box, or select a directory name in the list box, QuickBASIC lists all the files and subdirectories in the directory you select. (In contrast, when you enter or select a file name, that file is loaded into memory.) The list box displays file names in lowercase letters and directory names in uppercase letters.

You can use the DOS wild-card characters, ? and *, to list groups of files, as shown in the following examples:

Task	Action
Listing all files in the current directory	Type `*.*` in the text box.
Listing the files in the root directory of drive A	Type `A:\*.*` in the text box.
Listing all files in a subdirectory named \BIN	Move the highlight to BIN in the list box, press SPACEBAR.
Listing all files in the current directory having the .BI extension	Type `*.BI` in the text box.
Listing all files in the directory immediately above the current directory	Type `..` in the text box. Or, if you are in the list box already, press a DIRECTION key or the period (.) to select .. When ..* followed by an extension appears in the text box, press ENTER.

Note

The "current working directory" is the directory shown by the CD command from DOS. By contrast, when you select a directory from a list box, you are only listing the files it contains, not changing the current working directory at the DOS level. If you want to change the current working directory, make the Immediate window active, and use the **CHDIR** statement.

4.2.2 Saving a Program (Save, Save All, Save As)

The Save, Save All, and Save As commands all save files. The following list explains the differences among them:

Command	Result
Save	Writes the contents of the current module (the file being displayed in the active window) to a disk file.
Save All	Saves all currently loaded files that have changed since the last save. A currently loaded file is one whose name appears in the dialog box of the View menu's SUBs command. Save All saves all changed files and is useful when you are working with multiple modules.
Save As	Saves the current module with the name you specify. It is useful for saving a module under a new name and changing the format in which the file is saved. If you change the name, the old file still exists with the name it had the last time it was saved.

When you choose Save As from the File menu, the dialog box shown in Figure 4.3 opens:

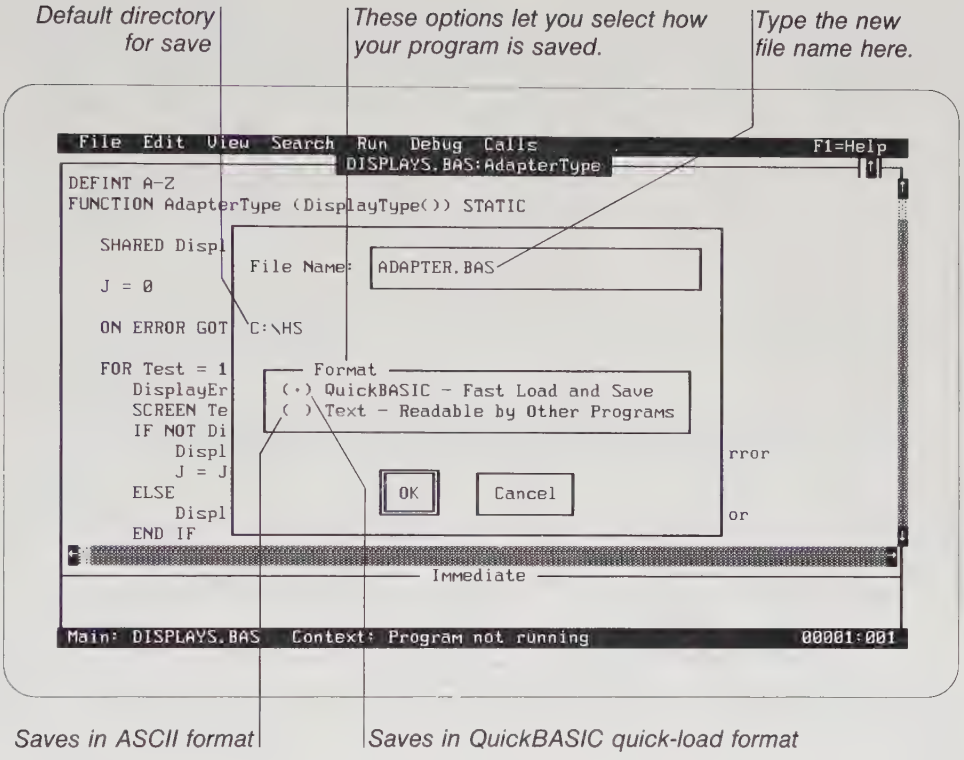


Figure 4.3 Save As Dialog Box

The following list explains the dialog box:

Option	Action
File Name text box	Holds the name of the file you want to save. If you are saving a file that already has a name, the file name appears selected in the text box. If you want to save the file with a different name, just begin typing. What you type replaces the old name. The next time you save the file, the new file name will appear in the text box.

QuickBASIC—Fast Load and Save option	Saves your program in QuickBASIC format. A program saved this way loads faster than one saved as a text file (see below), but can only be edited using QuickBASIC. Because it is a non-ASCII file, you won't be able to change it using another text editor. If you don't explicitly choose an option, this is the way QuickBASIC stores a new program. Documents and include files cannot be stored in this format.
Text—Readable by Other Programs option	Stores your file on the disk as an ASCII, or text, file. Text files can be read, modified, or printed by any editor or word processing program that reads ASCII files. Files you classify with the Document or Include option are always stored in this format.

The dialog box shown in Figure 4.3 also appears if you do any of the following:

- Choose Save or Save All while an unnamed program is in memory.
- Open another program while an unnamed program is in memory.
- Create either an executable file or a Quick library from an unnamed module.

4.3 Modules

When QuickBASIC loads a program, each file it loads is referred to as a module. A program may have many modules, but every program consists of at least one module, referred to in QuickBASIC as the "main module." Include files are not modules, although they may play an important part in a program.

When you load or create a file as a module, QuickBASIC formats it and checks its syntax.

To get a list of the modules currently in memory, choose the SUBs command from the View menu. The file names appear in capital letters in the SUBs dialog box.

4.3.1 Creating and Loading Modules (Load File)

The Load File command on the File menu loads single modules—as well as include files and document files—from disk into memory.

When you choose Load File from the File menu, the dialog box shown in Figure 4.4 appears:

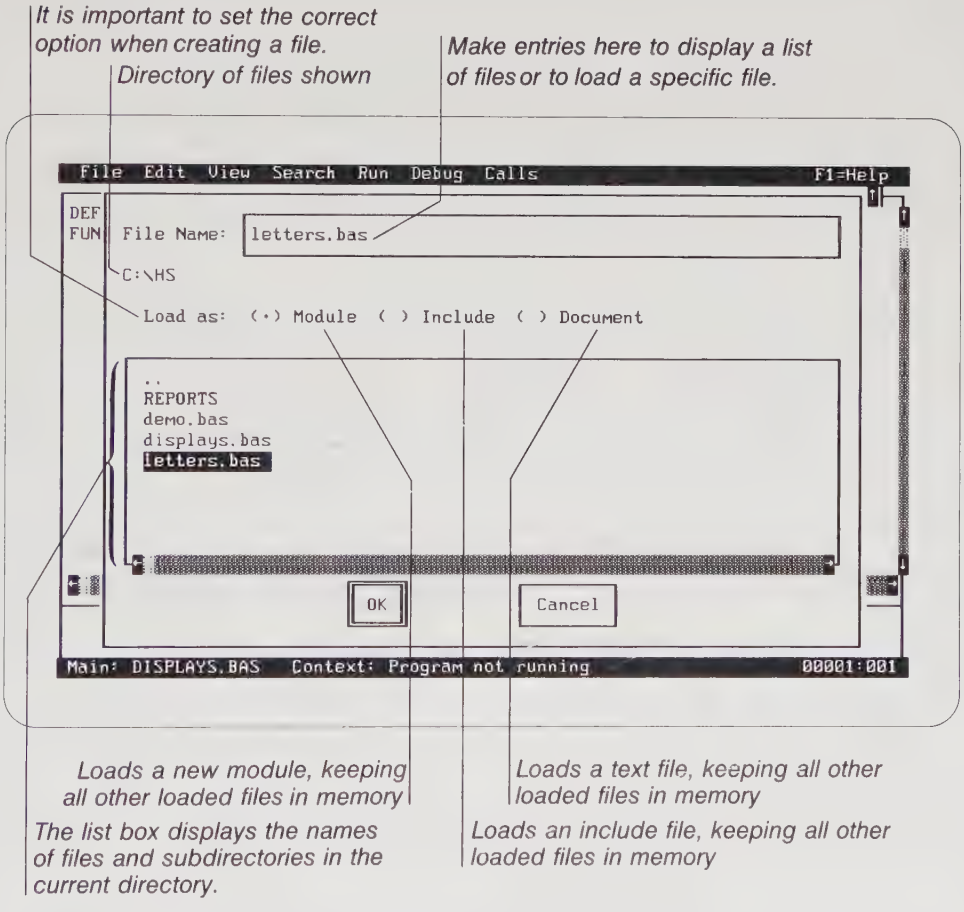


Figure 4.4 Load File Dialog Box

Note that since QuickBASIC considers a program to be one or more modules, the New Program and Open Program commands from the File menu also load files as modules. In the case of a single-module program,

these commands create or open only a main module. With a multiple-module program, Open Program loads the main module, plus any other modules that make up your program. See Section 4.4.1, "The .MAK File," for information on how this is done.

To begin a new module as part of the program currently in memory

1. Choose the Create File command from the File menu. The Create File dialog box appears (see Figure 4.5).

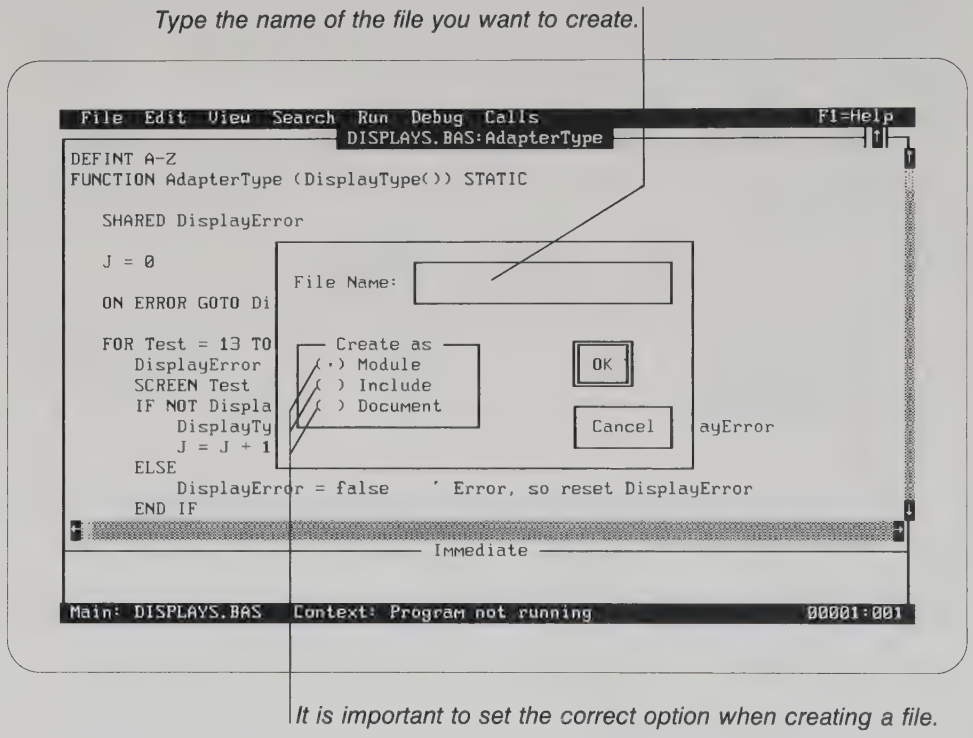


Figure 4.5 Create File Dialog Box

2. Enter the name of the module you want to create in the text box of the dialog box. Note that the Module option button is already selected.

3. Press ENTER to create the new module.

To incorporate modules from other programs into the program currently in memory

1. Choose the Load File command from the File menu.
2. Select the name of the module containing procedures you want to use in the current program. There are two ways to specify the file you want to load:
   - Type the name of the file, then press ENTER.
   - Press the TAB key to move to the list box, and then use the DIRECTION keys to move through the list box until the file you want to load is highlighted. (Pressing the first letter of the file name you want to load also moves the highlight alphabetically through all the file names beginning with that letter. For example, if you are looking for the file PROG1.BAS, pressing the "p" key eventually moves the input focus to the file with that name.)
3. Press ENTER to load the file as a module, since the Module option is already chosen.
4. Repeat steps 1–3 for each module you want to load.
5. Save the program using the Save All command, once all the modules are loaded.

4.3.2 Displaying Multiple Modules

Although QuickBASIC can display only two View windows at a time, it can load as many modules as will fit in memory. When you load a new module into memory, it is displayed in the active window. The module that was in the window is still in memory but is not displayed.

To display a loaded module in a View window

1. Choose the SUBs command from the View menu. The SUBs dialog box appears (see Figure 4.6).

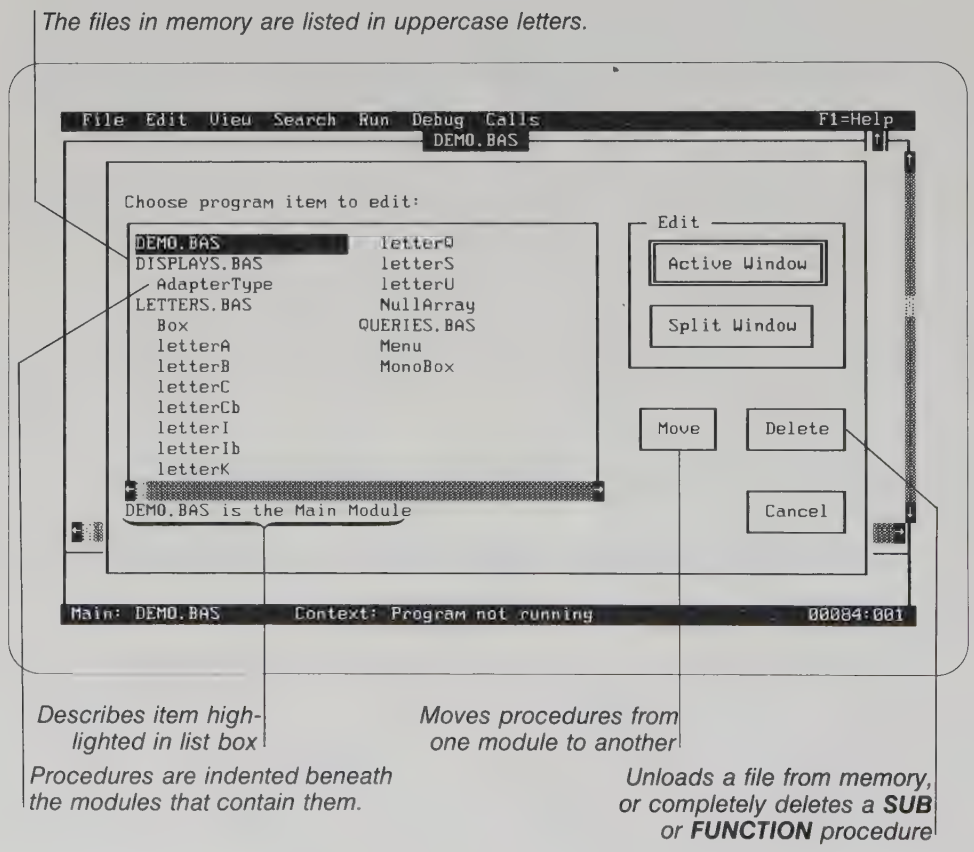


Figure 4.6 SUBs Dialog Box

2. Select the module (or specific procedure within the module) that you wish to display.

3. Press ENTER.

If you want to look at two modules or procedures, choose the Split Window command button to divide the screen into two View windows. The module you selected appears in the new window.

4.3.3 Saving Multiple Modules (Save All)

It is most convenient to use the Save All command from the File menu to save your work when you are working with multiple modules. Save All saves all modules that have changed since the last save. The Save command saves only the module associated with the active View window.

4.4 Changing the Main Module (Set Main Module)

Every program has at least one module. If a program has multiple modules, then the "main module" contains the first statement to be executed when the program is run.

A module can be part of many different programs, but it can be the main module of only one program (the program bearing its name).

The status bar shows the name of the main module of the currently loaded program. The main module is also the first entry in any list box that shows the modules in a program.

As you are editing your program you may find that you want to reorganize it and have execution begin with a module other than the one currently designated as the main module. To do this

1. Choose the Set Main Module command from the Run menu. The Set Main Module command's dialog box contains a listing of currently loaded modules (see Figure 4.7).

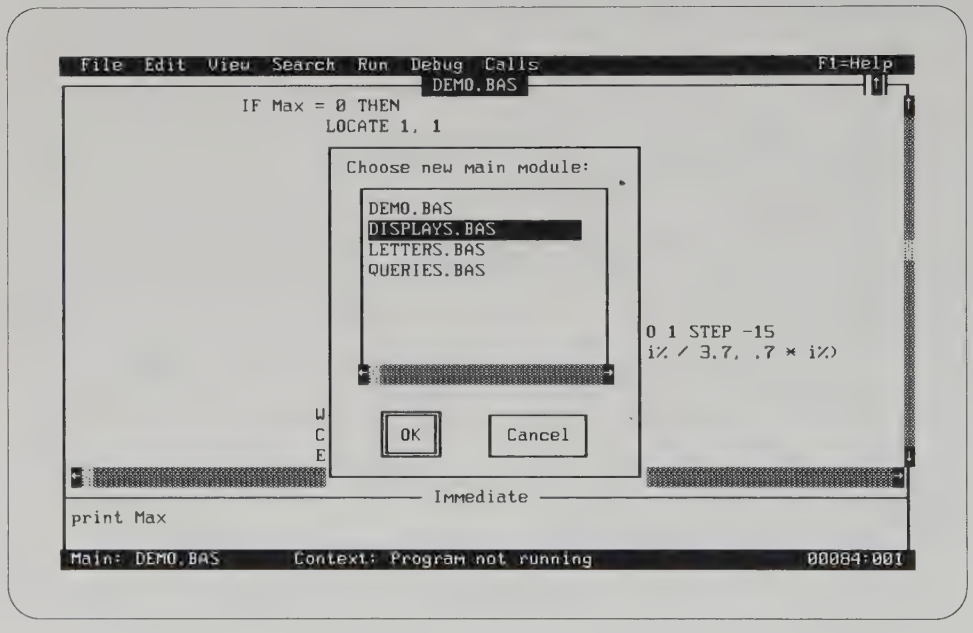


Figure 4.7 Set Main Module Dialog Box

2. Select the module you want to designate as the main module, and press ENTER.

4.4.1 The .MAK File

When you save a multiple-module program, QuickBASIC creates a special file that contains the names of all the modules in the program. The name of this file consists of the base name of the main module plus the extension .MAK, hence the term ".MAK file." The main module of a program is the first entry in the .MAK file.

When you reload the program, QuickBASIC uses the .MAK file to locate all of its modules. Thus, you should not delete a .MAK file, and if you move a multiple-module program to another directory you should also move the .MAK file.

Only program modules are included in a .MAK file. QuickBASIC does not add other types of files to the .MAK list. Therefore, if your program relies on include files, the program must specify each such file as an argument to a **$INCLUDE** metacommand.

The .MAK file is just a text file. If you want to edit it, load it in QuickBASIC using the Document option of the File menu's Load File command.

4.4.2 Removing Modules from a Program (Unload File)

Choose the Unload File command from the File menu to remove an entire module from your program. After you unload a module, the file containing the module still exists on the disk; but when you save your program it is no longer a part of the program, and the name of the module does not appear in the program's .MAK file.

If you inadvertently load a module, then decide you don't really want it in your program, follow these steps to unload it:

1. Choose the Unload File command from the File menu.
2. Highlight the name of the module that you no longer want in the program.
3. Press ENTER. The module is unloaded, but still exists as a disk file.

4.5 Include Files

Include files are text files compiled into a program with a **$INCLUDE** metacommand. They are similar to modules in that you can use the same include files in many different programs, as appropriate. However, unlike modules, include files cannot contain **SUB**, **END SUB**, **FUNCTION**, or **END FUNCTION** statements.

Include files are always saved in text format, even when you select the QuickBASIC—Fast Load and Save option from the Save As command dialog box.

Note

If you have QuickBASIC 2.0 or 3.0 programs that use include files to define **SUB** procedures, see Section 4.7, "Merging the Contents of Two Files," for an explanation of how to incorporate these procedures into your QuickBASIC Version 4.0 program.

Though you can use include files in any way you please, they are especially useful for consolidating declarative statements that affect several modules of a multiple-module program. Examples of declarative statements are the following:

Statement	Use
COMMON	Lists variables common to more than one module
DECLARE	Declares **SUB** or **FUNCTION** procedures and enables type checking for their arguments
DIM	Declares variable types
TYPE	Declares user-defined types

See the *BASIC Language Reference*, and Chapter 2, "Procedures: Subprograms and Functions," in *Programming in BASIC: Selected Topics*, for more information on these statements.

You can place **$INCLUDE** metacommands anywhere you want within your programs. However, when the include files contain declarative statements, the **$INCLUDE** metacommand is normally placed as close as possible to the beginning of the module so that the scope of the declarations encompasses the entire module, and because some declarative statements must precede executable statements.

4.5.1 Creating and Loading Include Files (Create File)

If you want to create an include file, choose the Create File command from the File menu, then select the Include option to classify the file as an include file.

The most important thing to remember when creating or loading any file is to classify it properly in the "Create as" or "Load as" options box. For example, when loading or creating an include file, if you do not explicitly select the Include option, QuickBASIC loads the file as a program module. Then, when you save the program, QuickBASIC enters the name of the include file into the .MAK file along with the names of your other program modules. The next time you open the program, the include file is loaded with the other module files, in addition to being compiled as an include file. Having an include-file name in your .MAK file serves no purpose and wastes memory.

4.5.2 Viewing and Editing Include Files (Include File, Included Lines)

Use the Include File and Included Lines commands from the View menu for viewing and editing include files (see Figure 4.8).

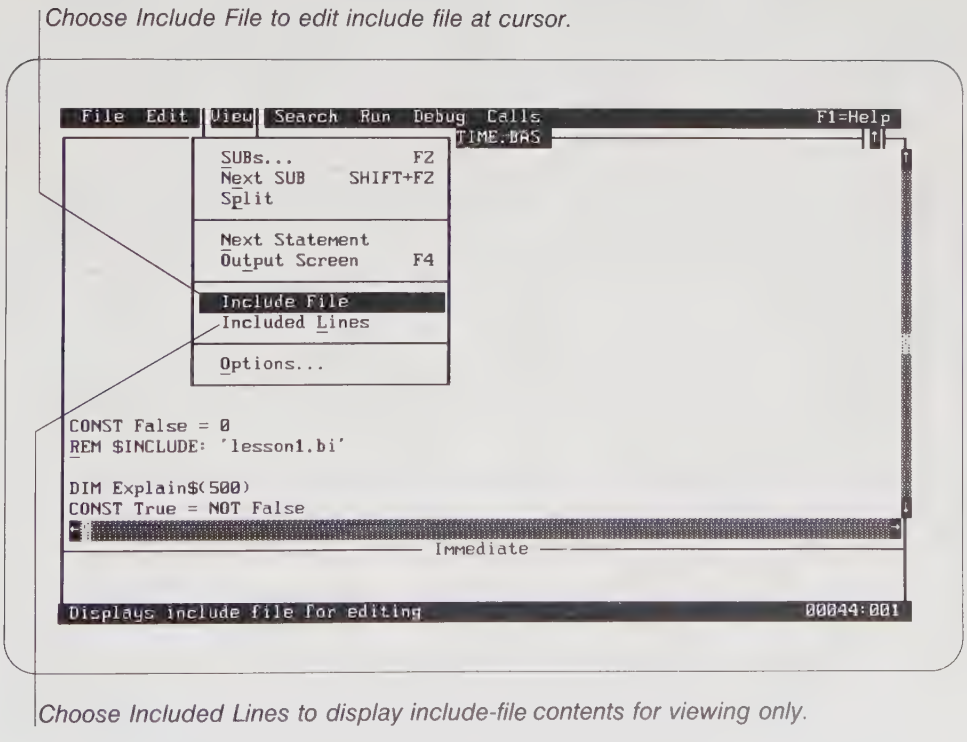


Figure 4.8 Include File Commands on View Menu

If you want to view, but not edit, an include file, choose the Included Lines command. This command is a "toggle"; that is, choosing Included Lines turns the option on if it's off, and off if it's already on. A check mark (√) appears next to the option if it is on. This lets you look at the include file's contents, which appear in high-intensity video beneath the appropriate **$INCLUDE** metacommand. Although you can scroll around in the file after choosing Included Lines, you cannot edit the file as long as this option is on. If you try to edit one of the included lines while they are highlighted, a dialog box appears and asks if you want to edit the include file. If you do, QuickBASIC loads the file into memory and places its text in the active View window.

If you try to edit lines other than those in the include file, a dialog box appears asking if you want to turn the Included Lines option off. If you press ENTER, the highlighted include-file lines disappear.

If you want to view and edit the include file, choose the Include File command. When the cursor is on a **$INCLUDE** metacommand line, this command moves the text of the include file into the active window for editing. Notice that QuickBASIC keeps the actual text of include files separate from the rest of the program text. This is similar to the way in which the text of **SUB** or **FUNCTION** procedures is separated from other program text. The difference is that QuickBASIC does not consider the text in an include file as subordinate to the rest of the program, as it does with procedures.

Once the include file exists, you may want to edit it without returning to the place in your program containing its referencing **$INCLUDE** metacommand. To do so

1. Split your screen with the Split command from the View menu.
2. Choose the Load File command from the File menu to open the include file. Be sure to select the Include option.
3. Edit the include file.
4. Choose the Save command from the File menu to save any changes you make to the include file.
5. Press F6 to move back to the part of the program in the other View window.
6. Choose the View menu's Split command again to close the extra window.

You must save the include file before changes can be incorporated for the next run of the program. QuickBASIC prompts you if you forget.

4.5.3 Nesting Include Files

There is no limit to the number of **$INCLUDE** metacommands that you can have among the modules of a program, but there is a limit to the extent of "nesting."

Nesting occurs when a **$INCLUDE** metacommand appears within another include file. Include files can be nested up to five levels. If you have a circular reference in an include file (that is, a **$INCLUDE** metacommand that references itself or a file in which it is referenced), you get the error message "Too many files" when you try to run your program.

Note

When QuickBASIC processes a **$INCLUDE** metacommand, only the current working directory (or a directory specified in the metacommand itself) is searched. If you want to keep all your include files in one central directory, you can make them accessible to QuickBASIC by consistently referencing them with a full or relative path name. For example, the following line directs QuickBASIC to look for the include file named declares.bi in the directory c:\include

```
' $INCLUDE: 'c:\include\declares.bi'
```

This could also be specified with a relative path, as in the following line:

```
' $INCLUDE: '..\include\declares.bi'
```

Although any extension, including .BAS, is legal for files incorporated into your program with the **$INCLUDE** metacommand, it is recommended that include files use some unique extension (such as .BI) to avoid confusion with program modules.

4.6 Documents

When you create or load a file as a document, you turn off the smart editor features of QuickBASIC; that is, QuickBASIC does not format the text that you enter, nor does it check syntax to make sure that you enter valid BASIC statements.

If you want to use QuickBASIC to edit a file that does not contain BASIC code, open the File menu, then choose the Create File command to create a new document or the Load File command to edit an existing document. Be sure to specify the file as a document by selecting the Document option in the resulting dialog box.

A file loaded as a document may represent a program module in other circumstances—that is, it may contain valid BASIC-language source code. However, when you load it into QuickBASIC as a document, you are instructing QuickBASIC not to treat it as program material. Consequently, you cannot run a program when it is loaded as a document. A file loaded as a document cannot be saved with the QuickBASIC—Fast Load and Save option of the Save As command. You may select that option, but QuickBASIC always saves a document in Text format. Conversely, if you save a program or module with the QuickBASIC—Fast Load and Save option, you get the error message "Bad file mode" if you try to load the file as a document. To load such a file as a document, you must first save it with the Text—Readable by Other Programs option.

To run a program after loading it as a document, reload the program by choosing the Open Program command from the File menu.

When the Load File and Create File dialog boxes appear on the screen, the Module option is already selected. Thus, if you want to load a file as an include file or document, you have to explicitly select the Include or Document option. You may unintentionally load files as modules when you first start using QuickBASIC. If you think this has happened, select the file's name in the View menu's SUBs dialog box and note the description beneath the list box. If the file is described with a classification different from the one that you intended, choose the Delete command button to unload it from memory. This has no effect on the file on disk, but allows you to load it again properly.

Note

Until you are very familiar with the QuickBASIC environment, use caution in deleting names from the list box of the SUBs dialog box. Remember that while modules exist independently as disk files, **SUB** and **FUNCTION** procedures exist only within modules. When you delete a procedure, you are not temporarily unloading it from memory; instead, you are deleting it from the disk file completely. Therefore, if you save a program after deleting a procedure name from the list box, the procedure is gone forever.

4.7 Merging the Contents of Two Files (Merge)

The Merge command on the File menu inserts a file before the cursor position in the active window. When you choose the Merge command from the File menu, the dialog box shown in Figure 4.9 opens:

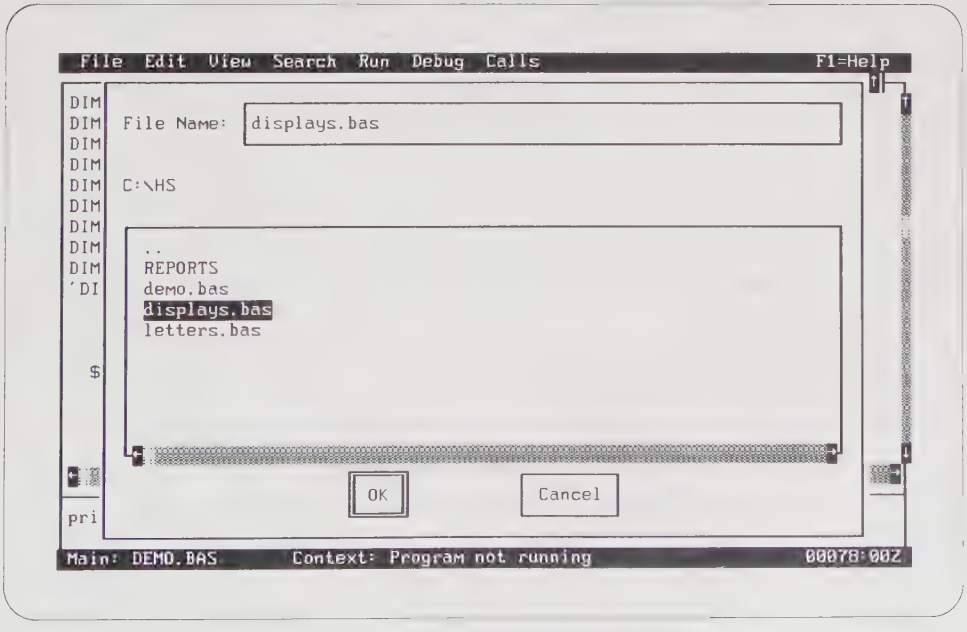


Figure 4.9 Merge Dialog Box

The Merge dialog box operates the same way the Open Program dialog box does. However, the Merge command inserts the file into the current module, whereas the Open Program command clears whatever is currently in memory, and then loads the program. See Sections 4.2.1.1–4.2.1.2 for an explanation on using the Open Program dialog box and listing the contents of the current directory.

QuickBASIC Version 4.0 does not permit **SUB** or **FUNCTION** procedures in include files. If you have programs created in QuickBASIC 2.0 or 3.0 that depend on subprograms in include files, use the Merge command to merge the include file with your main module. (Remember to delete the **$INCLUDE** metacommand, too.)

When you merge an include file containing a **SUB** procedure, the text of the procedure does not appear in the currently active window. However, as soon as you execute the Merge command, you can run your program.

To view or edit the merged **SUB** procedure, choose SUBs from the View menu, then select the merged procedure's name from the list box.

Rather than merging it, you may want to rewrite your **SUB** procedure and have it in a separate module. In this case some restructuring may be necessary if the **SUB** uses variables declared with the **SHARED** attribute. See Chapter 2, "Procedures: Subprograms and Functions," in *Programming in BASIC: Selected Topics*, and Chapter 4, "Programs and Modules," in the *BASIC Language Reference*, for more information.

4.8 Printing Files (Print)

The Print command on the File menu lets you print selected text, or text from the active window, the current module, or the current program. To use the Print command, your printer must be connected to LPT1.

When you choose the Print command from the File menu, the dialog box shown in Figure 4.10 opens:

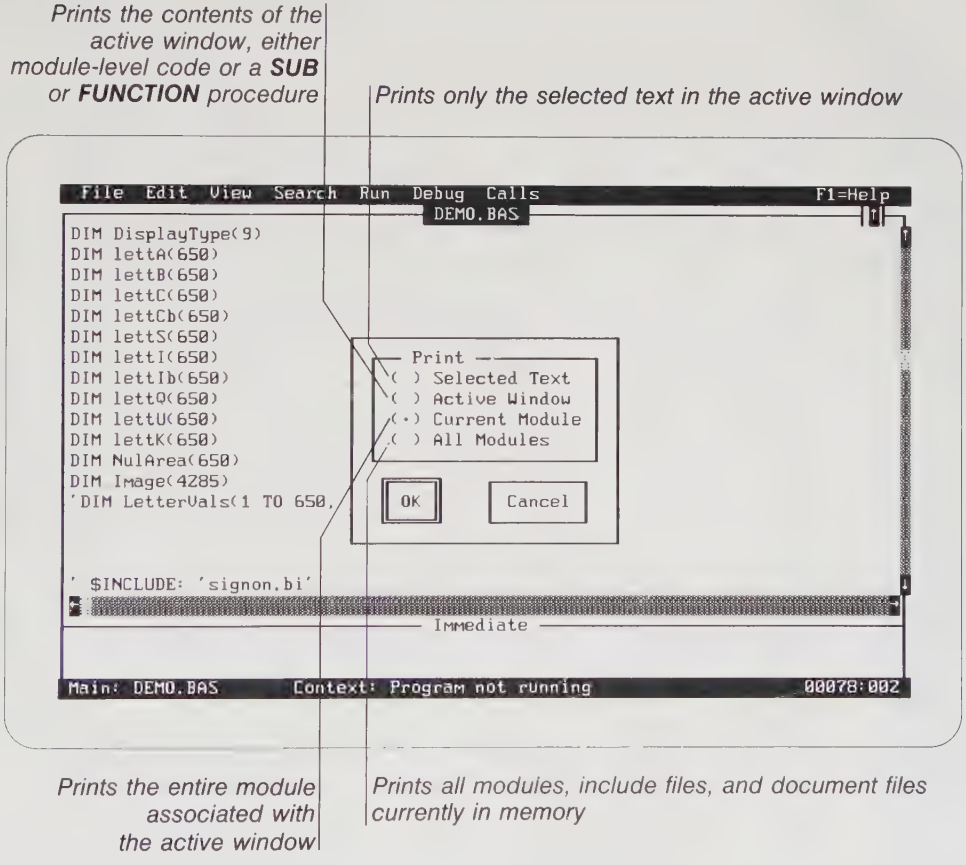


Figure 4.10 Print Dialog Box

Select the appropriate option from the Print options box to print your file.

CHAPTER 5

EDITING

5.1 Entering Text 113

5.2 Smart-Editor Features 113
- 5.2.1 When Is the Smart Editor On? 114
- 5.2.2 Automatic Syntax Checking 114
- 5.2.3 Turning Off Syntax Checking 115
- 5.2.4 Automatic Formatting 116

5.3 Inserting and Overtyping 116

5.4 Selecting Text 117

5.5 Deleting and Inserting Text 117
- 5.5.1 Using the Clipboard 117
- 5.5.2 The Clear, Cut, Copy, and Paste Commands 118

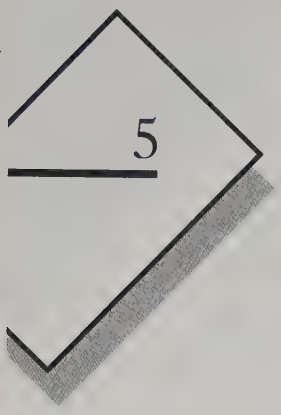
5.6 Undoing Your Last Edit 119

5.7 Moving and Copying Text 120

5.8 Searching and Replacing 121
- 5.8.1 Finding Text 122
  - 5.8.1.1 Find 123
  - 5.8.1.2 Selected Text 125
  - 5.8.1.3 Repeat Last Find 125
  - 5.8.1.4 Label 125
- 5.8.2 Replacing Selected Text (Change) 126

5.9 Using Place Markers in Text 129

5.10 Entering Special Characters 129
- 5.10.1 Entering High-Order ASCII Characters 129
- 5.10.2 Entering Control-Sequence Characters 130



CHAPTER

5.11 Indenting 131
5.12 Joining Lines 132
5.13 Copying Text from Other Files 132
5.13.1 Copying an Entire File 132
5.13.2 Copying Part of a File 133
5.14 Summary of Editing Commands 134

This chapter describes how to enter and edit program text using QuickBASIC editing commands and the Edit and Search menus. Before beginning this chapter you should be familiar with the QuickBASIC screen and the techniques described in Chapter 3, "Getting Around in the QuickBASIC Environment."

When you finish reading this chapter you will know how to

- Perform simple editing functions such as entering text and moving the cursor.
- Delete and insert text.
- Move and copy blocks of text.
- Search for and replace characters, words, or a group of words.
- Copy text from other files.

5.1 Entering Text

When typing a BASIC program, you can enter a line either by pressing the ENTER key at the end of the line or by moving the cursor off the line with a DIRECTION key or the mouse pointer.

5.2 Smart-Editor Features

Most compilers require you to write source code with a text editor, then compile the resulting text file. QuickBASIC, however, has always provided both a text editor and compiler in an integrated environment. QuickBASIC Version 4.0 goes a step further and combines a text editor, a special kind of compiler, and a debugger into a single smart editor. The smart editor includes special features, such as automatic syntax checking, that make it easier to enter and edit BASIC programs. When the smart editor is on, QuickBASIC performs the following actions each time you enter a line:

- Checks the line for syntax errors.
- Formats the line as needed.
- Translates the line to executable form, if the syntax is correct. This means that each line you enter is ready to run immediately.

5.2.1 When Is the Smart Editor On?

In most cases, you use the QuickBASIC editor to enter and edit program text. If you start QuickBASIC and simply begin typing, QuickBASIC assumes that you want to write BASIC-language statements, so the smart editor features are turned on automatically. The smart editor is also on when you

- Choose New Program or Open Program from the File menu.
- Start QuickBASIC with the base name of a program you want to work on.
- Choose Create File or Load File from the File menu, then select the Module or Include option.

You may occasionally need to edit a text file that consists of something other than BASIC-language statements; such files include QuickBASIC's .MAK file and document files your program may need. For this purpose, you can turn off the smart-editor features and use the QuickBASIC editor as an ordinary word processor. This can be done by choosing Create File or Load File from the File menu, then selecting the Document option.

5.2.2 Automatic Syntax Checking

When the smart editor is on, it checks the syntax of each line when you enter it. A syntax error indicates that the line contains something that is not a meaningful QuickBASIC statement. For instance, this line causes a syntax error because the keyword **GOTO** is not a valid argument to the **PRINT** statement:

```
PRINT GOTO
```

When an error occurs, QuickBASIC displays a dialog box containing an error message. Pressing ESC or SPACEBAR clears the message from the screen and positions the cursor where the error occurred. You must correct syntax errors before your program can run. If you move on without correcting the error, the message appears again when you run the program.

Note

When you clear a syntax-error message, the cursor is placed on the statement that caused the error. You can move the cursor to any part of the BASIC keyword, and then press SHIFT+F1 for on-line Help with the syntax of the statement.

Certain typing mistakes are not recognized as syntax errors when entered. For example, no error message appears if you mistype the **PRINT** keyword when entering this line:

```
pront
```

Although this doesn't look like a valid BASIC statement, QuickBASIC accepts the line until you try to run the program, at which time it flags the error. Until then, the statement is interpreted as a call to a **SUB** procedure named "pront."

5.2.3 Turning Off Syntax Checking

Syntax checking is always on unless you have opened the file with the Document option, as explained in the previous section. You can toggle syntax checking on and off with the Syntax Checking command from the Edit menu, shown in Figure 5.1. A check appears next to that command when syntax checking is toggled on.

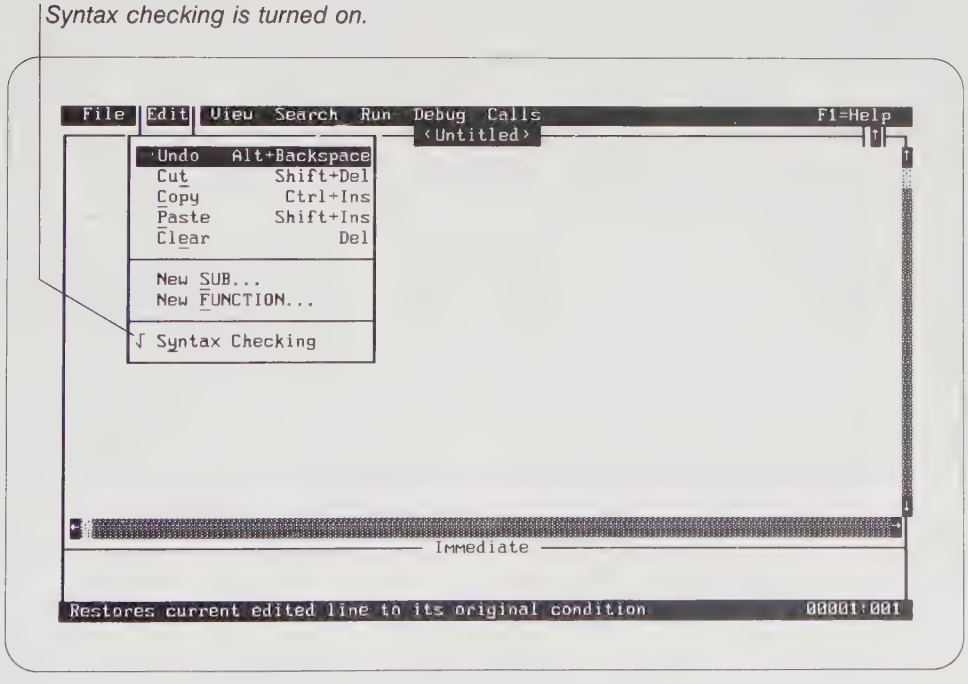


Figure 5.1 Syntax-Checking Toggle on Edit Menu

5.2.4 Automatic Formatting

In addition to checking for syntax errors, the smart editor automatically standardizes the format of each line as you enter it. After formatting

- All BASIC keywords appear in uppercase letters.
- Spaces appear before and after all BASIC operators.
- The names of variables and procedures are capitalized consistently.
- Punctuation is added where appropriate; for instance, the smart editor inserts semicolons between consecutive items in a **PRINT** statement if you do not include them yourself, and supplies a matching final quotation mark to the argument of a **PRINT** statement if you forget it.

QuickBASIC keeps track of the names of all variables and procedures in your program and capitalizes these names consistently. For example, say that you have a variable named myvar, which you enter first with all lowercase letters. If you later decide to change the name to MyVar with uppercase for both the M and V, QuickBASIC changes all instances of the name myvar in the module to MyVar. If you save or run the program, then return to the editor, you can see that the name myvar is changed to MyVar.

5.3 Inserting and Overtyping

New characters that you type are added to the text either by inserting or overtyping. If the insert toggle is set so the cursor is a single flashing line, the editor "inserts" each new character at the cursor position. When you use the insert toggle to change the cursor to a blinking box, each new character "overtypes" the character under the cursor, replacing it. Either INS or CTRL+V toggles the shape of the cursor.

5.4 Selecting Text

Although simple editing tools such as the BKSP and DIRECTION keys work well for small-scale editing jobs, it is often convenient to work with text in larger blocks. Before performing a large-scale editing operation, such as copying or deleting a block of text, you must indicate the portion of text you wish to affect by selecting (highlighting) it. Hold down the SHIFT key while pressing a DIRECTION key to select text. For instance, SHIFT+RIGHT selects text to the right of the cursor.

Holding down the SHIFT key while using any of the editing shortcut keys also selects text. For example, pressing SHIFT+END selects everything from the cursor to the end of the line. To restore the text to its unselected state, press any direction key.

Note

When you select text, the cursor is placed at the end of the selected text. If you begin typing while the text is still selected, what you type replaces the selected text.

If you copy text from one place to another, the original text remains selected. If you do not restore it to its unselected state, whatever you type next (while the selected text is in the active view window) replaces the selected text.

5.5 Deleting and Inserting Text

You can delete and insert text using the Edit menu commands. Section 5.5.1 explains the Clipboard, a temporary storage area for deleted text. Section 5.5.2 discusses the commands for deleting and inserting text.

5.5.1 Using the Clipboard

The Clipboard is a temporary storage area that holds text you copy or cut from the active window. Text does not accumulate in the Clipboard. Anything you place in the Clipboard replaces everything that was already there.

Text remains in the Clipboard until you end your QuickBASIC session or replace it with new text. (The New Program and Open Program commands don't erase the Clipboard.) Once the Clipboard contains some text, you can paste or insert that text as many times as you want.

Note

Cutting or copying a large block of text consumes a great deal of memory. When you use the Clipboard for large text blocks, you should empty it afterward by copying something small, such as a single character, into it.

5.5.2 The Clear, Cut, Copy, and Paste Commands

The Edit menu contains four commands—Clear, Cut, Copy, and Paste—that allow you to insert and delete blocks of text, as follows:

Command and Shortcut Key	Action
Clear (DEL)	Deletes selected text completely
Cut (SHIFT+DEL)	Deletes the selected text and places it in the Clipboard
Copy (CTRL+INS)	Places a copy of the selected text in the Clipboard, leaving the original text untouched
Paste (SHIFT+INS)	Inserts text from the Clipboard at the cursor

To use Clear, Cut, or Copy, first select the text you wish to delete or copy, and then choose Clear, Cut, or Copy from the Edit menu. The Paste command inserts text from the Clipboard into the current text. This command works only when the Clipboard contains some text, and its result depends

on whether any text is selected at the time. Note that if you have selected text in the active window, Paste *replaces* the selected text with the contents of the Clipboard. If no text is selected, Paste inserts the contents of the Clipboard to the left of the cursor, or if the block exceeds one line, on the line above the cursor.

Use these steps to paste text:

1. Move the cursor to the place where you want to insert the text. The new text will be inserted to the left of the cursor, or if the block exceeds one line, on the line above the cursor.
2. Choose the Paste command from the Edit menu (or press SHIFT+INS).

5.6 Undoing Your Last Edit

The Undo command from the Edit menu (ALT+BKSP) restores the current line to its original state, reversing all changes. You can restore a line as long as the cursor is on it. Once you move the cursor off the line, any changes you have made are permanent.

Undo works only when you have just made some change to the current line. If you try to choose Undo when the cursor rests on an unchanged line, nothing happens.

Note

You cannot use Undo to restore a line deleted with CTRL+Y because CTRL+Y removes the entire line. You can restore the line with the Paste command, since a line deleted with CTRL+Y is stored in the Clipboard.

5.7 Moving and Copying Text

You can move or copy blocks of text with the Cut, Copy and Paste commands on the Edit menu. Moving text places a copy of the selected text at a new location and deletes the original text block, whereas copying text places a copy of the selected text at a new location without disturbing the original.

Use these steps to move a block of text:

1. Select the text you want to move.
2. Choose the Cut command from the Edit menu (or press SHIFT+DEL) to delete the text. The selected text moves to the Clipboard, replacing what was there.
3. Move the cursor to the new location as follows, depending on the target location for the text:
   - To place the text in the currently active window, use the ordinary cursor-movement commands to move to the new location. Otherwise, use F6 to activate the window to which you want to move the text block.
   - If you want to place the text in another module, or in a **SUB** or **FUNCTION** procedure, first choose the SUBs command on the View menu, and then select the target module or procedure. When the procedure appears, move the cursor to the new location in that procedure, using the cursor-movement commands.
   - If you want to place the text in a program or module that is not currently in memory, first choose the New Program, Open Program, Load File, or Create File command from the File menu, and then move the cursor to the new location in that program or module, using the cursor-movement commands.
4. Choose Paste from the Edit menu (or press SHIFT+INS) to insert the text. The text remains in the Clipboard until the next Cut or Copy command, so you can insert the text in more than one location.

Copying a block of text involves the same general procedure: you must select the text you want to copy, store it in the Clipboard, move the cursor to the new location, and paste the text. However, you should use the Copy command (CTRL+INS) instead of the Cut command (SHIFT+DEL) when you store the selected text in the Clipboard.

5.8 Searching and Replacing

Figure 5.2 shows the Search menu, which allows you to find specific text anywhere in your program and optionally replace it with new text.

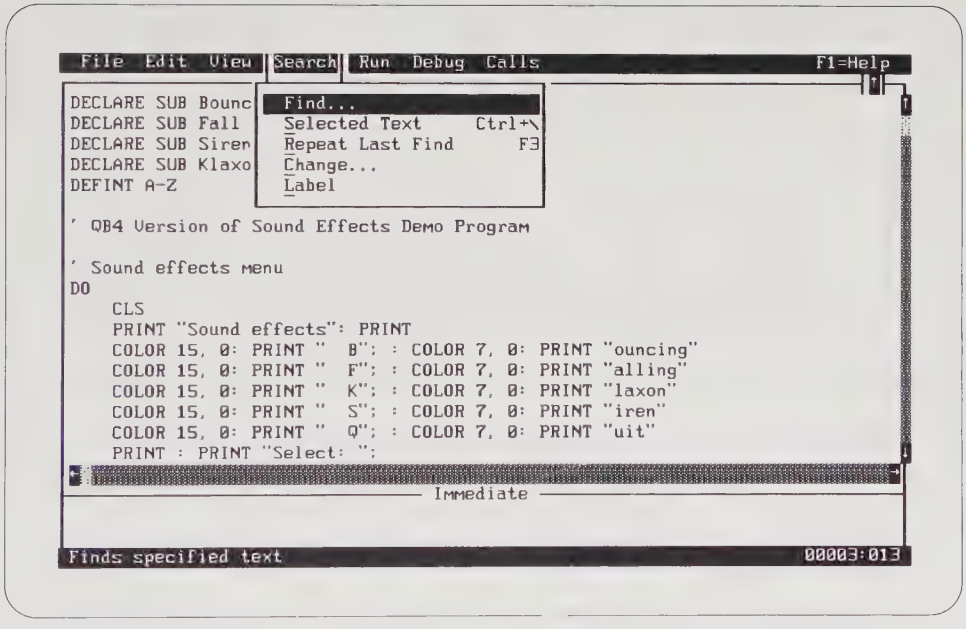


Figure 5.2 The Search Menu

The Search menu has several uses, particularly in long or complex programs. To move quickly to a distant part of the program, for instance, you can search for a line label or other text string which you know to be in that section, then resume editing at that spot. You can also search for a particular text string and replace it with new text.

To search or replace, you first tell QuickBASIC what to look for. There are several ways to specify the target of the search. If the cursor is currently anywhere in a word or on a space following the word, commands in the Search menu use that word as the target for the search. If the cursor is currently on a blank line, the last word searched for is the default target for the current search.

A search always begins at the cursor and moves forward until the specified text is found or the entire text has been searched without finding a match. Once a match is found, you again have several options: you can replace the target text with new text, continue searching, or abandon the search at that spot.

If a search reaches the end of the text without finding a match, it wraps around to the beginning of the text and continues forward until it finds a match or reaches the current cursor position. Thus, if a search fails to find a match anywhere in the text, you simply continue editing at the cursor.

Sections 5.8.1–5.8.2 describe the Search menu's commands in detail.

5.8.1 Finding Text

QuickBASIC offers several ways to specify the target of a search. You can search for a single character, a word, or a group of characters or words. You can also define the scope of the search. If the program has more than one module or procedure, you can specify which modules or procedures to include in the search. The following sections describe how to use the Search menu's commands to search for and change selected text.

5.8.1.1 Find

Use the Find command from the Search menu to search for a character, a word, or a group of characters and words in your program. When you choose the Find command, the Find dialog box opens (see Figure 5.3):

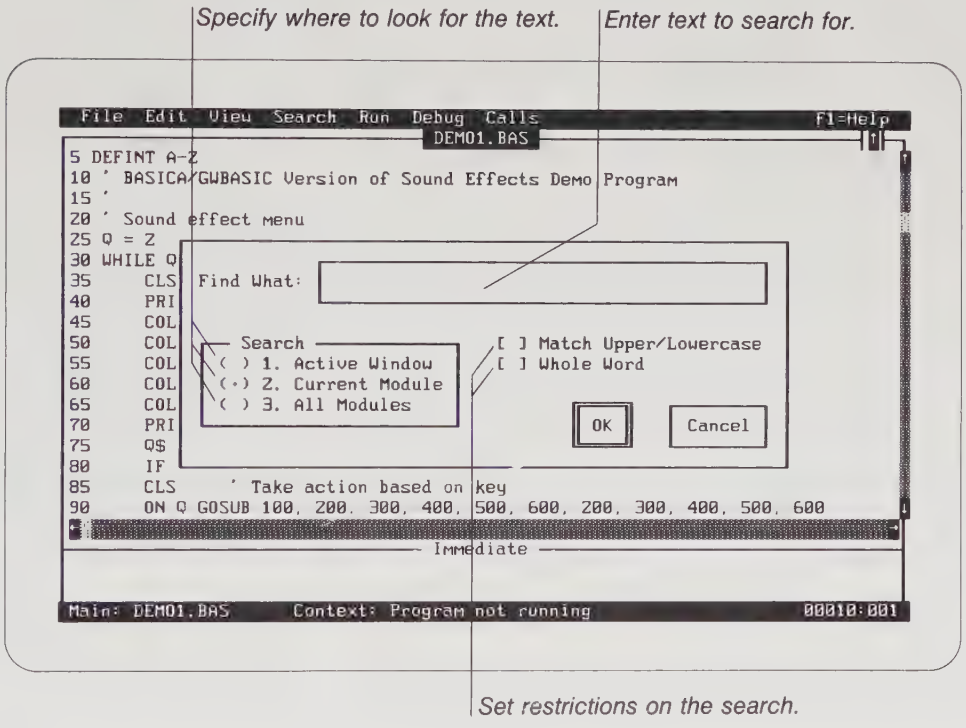


Figure 5.3 Find Dialog Box

To find text, perform the following steps:

1. Enter the text to search for in the Find What text box.
2. Use the Search option buttons to specify where to look for the text. The three options are as follows:
   - The Active Window option searches for text only in the active window.
   - The Current Module options searches for text only in the current module.
   - The All Modules option searches for text in all loaded modules.
3. Set restrictions on the search, if needed, by checking the Whole Word option, the Match Upper/Lowercase option, or both.
   - The Whole Word option finds the specified text only when it is surrounded by spaces, punctuation marks, or other characters not considered parts of a word. The characters A–Z, a–z, 0–9, and !, #, $, %, and & (commonly used in variable declarations) are considered parts of a word. For example, if CHR is the specified text, the Whole Word option locates CHR: but not CHR$.
   - The Match Upper/Lowercase option finds only instances of the specified text that match exactly. For example, if CHR is the specified text, the Match Upper/Lowercase option finds CHR but not Chr.
4. Press ENTER to execute the command.

If the text isn't found, the message "Match not found" appears. Press ENTER to remove the message box and continue.

5.8.1.2 Selected Text

The Selected Text command on the Search menu offers you another way to search for text: it looks for text that is highlighted in the active window. Follow these steps to use Selected Text:

1. Select the text you wish to search for. Note that the selected text must fit entirely on a single line.

2. Choose the Selected Text command from the Search menu (or press CTRL+\).

   The highlight moves to the next occurrence of the selected text.

5.8.1.3 Repeat Last Find

The Repeat Last Find command from the Search menu searches for the text specified in the previous Find command. The shortcut key is F3.

5.8.1.4 Label

The Label command on the Search menu looks for a line label by appending a colon to the currently selected text, or the text that you specified in the previous Find command. For example, if the word MyLab is the selected text, choosing Label causes QuickBASIC to search for the line label MyLab:.

5.8.2 Replacing Selected Text (Change)

Use the Change command on the Search menu to search for text and replace it with new text. When you choose the Change command, the dialog box shown in Figure 5.4 opens:

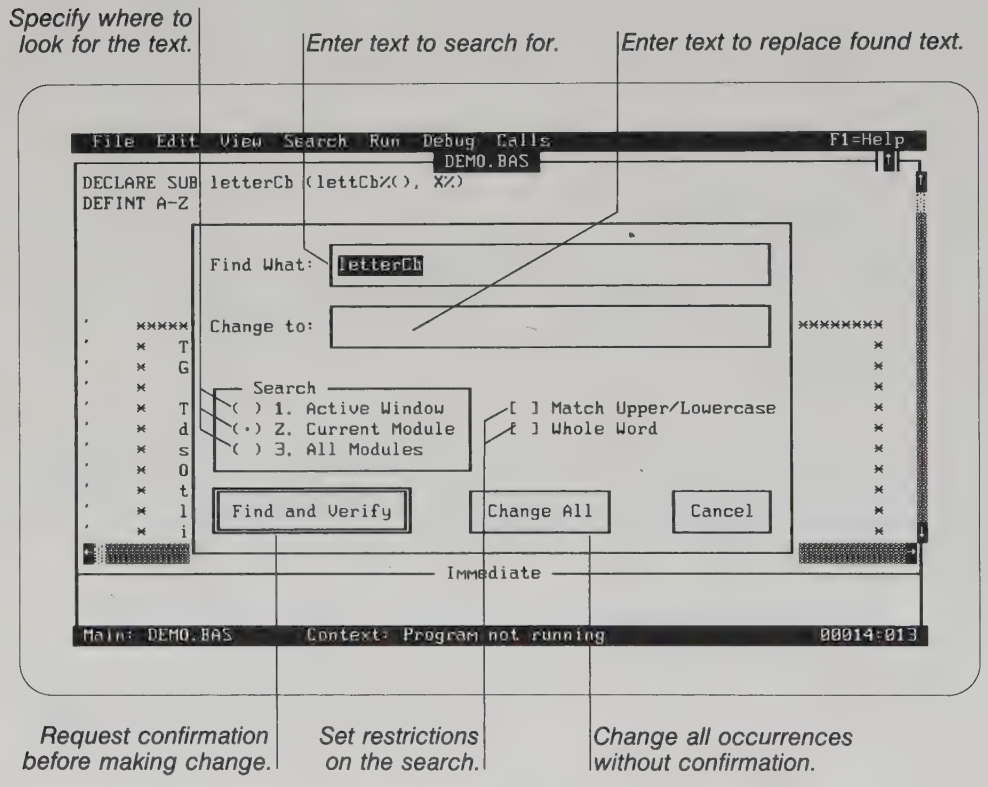


Figure 5.4 Change Dialog Box

To change text, perform the following steps:

1. Enter the text to find in the Find What text box.
2. Enter the replacement text in the text box labeled Change to.
3. Use the Search option buttons to specify where to look for the text to change. The three options are as follows:
   - The Active Window option searches for text only in the active window.
   - The Current Module options searches for text only in the current module.
   - The All Modules option searches for text in all loaded modules.
4. Set restrictions, if needed, on the search by checking the Whole Word option, the Match Upper/Lowercase option, or both.
   - The Whole Word option finds the specified text only when it is surrounded by spaces, punctuation marks, or other characters not considered parts of a word. The characters A–Z, a–z, 0–9, and !, #, $, %, and & (commonly used in variable declarations) are considered to be parts of a word. For example, if CHR is the specified text, the Whole Word option locates CHR: but not CHR$.
   - The Match Upper/Lowercase option finds only instances of the specified text that match exactly. For example, if CHR is the specified text, the Match Upper/Lowercase option finds CHR but not Chr.

5. Execute or cancel the command by choosing among the Find and Verify, Change All, or Cancel command buttons.
   - The Find and Verify command button finds the text, then displays three more command buttons: Change, Skip, and Cancel. The Change command button executes the change, the Skip button searches for the next occurrence (without changing anything), and the Cancel button stops the search and returns to where the search began (see Figure 5.5).

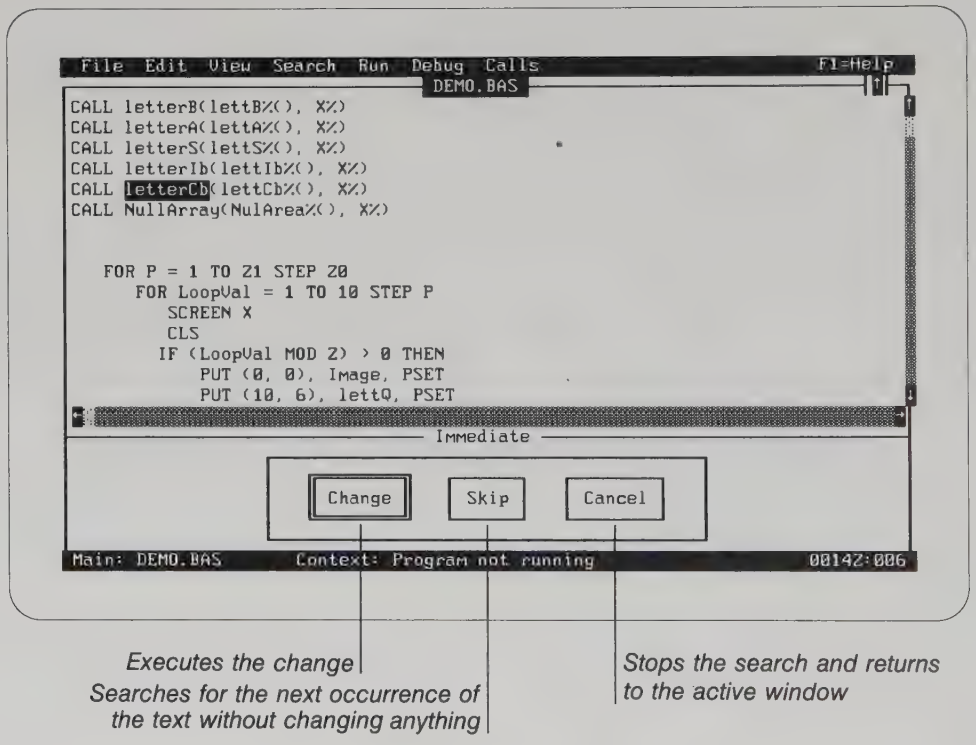


Figure 5.5 Change, Skip, and Cancel Command Buttons

- The Change All command button finds and replaces all occurrences of the specified text, without asking for confirmation.
- The Cancel command button cancels the command.

Important

The Undo command on the Edit menu cannot undo a change that you make with the Change command on the Search menu. Use caution with change operations, particularly Change All, which can make drastic changes in your program.

5.9 Using Place Markers in Text

If you are working on different parts of a large program, you can set place markers at different places in the program, then jump to each place marker when you need to work on that part of the program again. You can set up to four place markers, numbered 0–3, anywhere in your program. To use place markers

- Press CTRL+K *n* to set place marker number *n*.
- Press CTRL+Q *n* to move the cursor to place marker number *n*.

5.10 Entering Special Characters

QuickBASIC has procedures for entering the literal representations of special characters, including high-order ASCII characters and the literal ASCII characters associated with control sequences. (In general, entering such characters literally makes your programs more readable but less portable. For portability, use the **CHR$** function.)

5.10.1 Entering High-Order ASCII Characters

High-order ASCII characters are those corresponding to decimal 128–255. You can enter them literally in QuickBASIC (that is without using the **CHR$** function) by holding down the ALT key while you enter the appropriate decimal code on the numeric keypad. For example, holding down the ALT key while you enter 205 produces the double-line character at the cursor.

5.10.2 Entering Control-Sequence Characters

Many of the low-order ASCII characters corresponding to decimal 1–27 are associated with both literal characters and special actions. For example, the ASCII character represented by ALT+25 is associated with both the down-arrow character (↓) and ^Y (command to delete the current line). You may want to enter the down arrow in a print statement. One way to do this is to specify decimal 25 as the argument to the **CHR$** function. However, you can also enter the literal character itself by using the following method:

1. Hold down the CTRL key while pressing the letter P.

   This produces the ^P character on the QuickBASIC status bar. While this character is displayed, you can enter a control-key sequence to produce its literal character.

2. Hold down the CTRL key again, but this time press the letter corresponding to the literal character you want.

   This enters the literal ASCII character (as opposed to the special action of the control sequence) at the cursor.

For example, if you want to enter the down-arrow (↓) character within the quotation marks of a **PRINT** statement, you can enter it as CTRL+P+Y. If you tried to enter it as CTRL+Y or as ALT+25 while ^P was not visible on the status bar, your line would simply be deleted. Note that you cannot use this technique for entering the characters corresponding to the following: ^J, ^L, ^M, ^U. See Appendix A, "Keyboard Scan Codes and ASCII Character Codes," in the *BASIC Language Reference*, for a complete list of the ASCII characters.

5.11 Indenting

Use the SPACEBAR or TAB key to indent a single line of text. To indent a block of text, select the lines you wish to indent, then press TAB.

The QuickBASIC editor remembers the current indentation level and indents each new line automatically. That is, when you press ENTER to start the next line, the cursor appears immediately below the first character in the line above. This feature is useful when entering several indented lines, such as the body of a **FOR** loop. To cancel automatic indentation on a blank line, press HOME to move the cursor to the left margin. To move to the left margin of a line containing indented text, press CTRL+QS.

If you have a line that is indented too far to the right, you can remove the leading spaces for one indentation level by putting the cursor on the line and pressing SHIFT+TAB. If a block of text is selected at the time, the whole block is moved to the left.

The normal tab setting is eight spaces. You can change the tab setting as follows:

1. Choose the Options command from the View menu.
2. Move the input focus to the Tab Stops display option.
3. Enter the new tab-stop setting.
4. Press ENTER.

QuickBASIC uses individual spaces (rather than the literal tab character, ASCII 9) to represent indentation levels. The Tab Stops option in the View menu's Options dialog box sets the number of spaces per indentation level.

Some text editors use the literal tab character to represent multiple spaces when storing text files. The QuickBASIC editor treats literal tab characters in such files in one of the following ways:

1. Literal tab characters inside quoted strings appear as the character shown for ASCII character 9 (○) in the ASCII table in Appendix A, "Keyboard Scan Codes and ASCII Character Codes," in the *BASIC Language Reference.*
2. Outside quoted strings, tab characters indent the text following them to the next indentation level.

If you try to change the Tab Stops setting while such a program is loaded, QuickBASIC gives the following error message:

```
Cannot change tab stops while file is loaded
```

This is to prevent you from inadvertently reformatting old source files created with other editors. If you load a file and don't like the indentation, you can reset the indentation using the following procedure:

1. Choose the New Program command from the File menu to clear your file without saving it.
2. Choose the Options command from the View menu and set the Tab Stops option as described in the previous list.
3. Choose the Open Program command from the File menu and reload your program. When your program is reloaded, the indentations will reflect the new setting of the Tab Stops option.

Note that the preceding procedure works only for old programs. Text created within QuickBASIC cannot be reformatted this way because QuickBASIC never uses literal tab characters.

5.12 Joining Lines

Use the following procedure to join two lines of text:

1. Move the cursor to the first character on the second line.
2. Press BKSP.

5.13 Copying Text from Other Files

There are two ways to add the contents of other files to the current program, document, or include file. You can either copy an entire file, or just part of file, into the current text.

5.13.1 Copying an Entire File

To copy an entire file into the current text, use the Merge command on the File menu. The new text is inserted above the cursor.

5.13.2 Copying Part of a File

To copy part of a file into the current text:

1. Choose the Load File command from the File menu to load the file containing the text you want to copy.

   Only the module you just loaded is displayed. However, it is usually easier to copy text from one module to another if both modules are visible. To display both modules

   a. Choose the SUBs command from the View menu.

   b. Select the name of the module or procedure you wish to display.

   c. Choose the Split Window command button from the SUBs dialog box (or press ALT+S). This causes the View window to split in half, with the file you previously loaded in one window and the module or procedure you just selected in the other.

2. Press F6 or SHIFT+F6 to move the cursor to the window containing the text you want to copy.

3. Select the text you want to copy.

4. Choose the Cut or Copy commands from the Edit menu (or press SHIFT+DEL or CTRL+INS) to store the selected text in the Clipboard.

5. Press F6 or SHIFT+F6 to move the cursor back to the module where you want the new text to be copied.

6. Choose the Paste command from the Edit menu (or press SHIFT+INS). The text is inserted above the cursor.

7. Choose the Unload File command from the File menu to unload the file from which you copied the text.

5.14 Summary of Editing Commands

QuickBASIC's editor is designed to be flexible; it supports a keyboard interface familiar to users of other Microsoft products such as Microsoft Word. Many combinations of the CTRL key with other keys are familiar to users of WordStar and similar editors. Table 5.1 summarizes all the QuickBASIC editing commands. Sections 5.3–5.13 describe how to combine the actions summarized here to perform complex editing tasks, such as moving blocks of text.

Table 5.1
Editing Commands

Moving Cursor	Keypad Keys	WordStar-Style Key Combinations
Character left	LEFT	CTRL+S
Character right	RIGHT	CTRL+D
Word left	CTRL+LEFT	CTRL+A
Word right	CTRL+RIGHT	CTRL+F
Line up	UP	CTRL+E
Line down	DOWN	CTRL+X
First indentation level	HOME	---
Column 1 of current line	---	CTRL+Q S
Beginning of next line	CTRL+ENTER	CTRL+J
End of line	END	CTRL+Q D
Top of window	---	CTRL+Q E
Bottom of window	---	CTRL+Q X
Beginning of module/ procedure	CTRL+HOME	CTRL+Q R
End of module/procedure	CTRL+END	CTRL+Q C
Set marker	---	CTRL+K *n*†
Move to marker	---	CTRL+Q *n*†

Table 5.1 *(continued)*

Inserting	Keypad Keys	WordStar-Style Key Combinations
Insert mode on or off	INS	CTRL+V
Line below	END ENTER	---
Line above	---	HOME CTRL+N
Contents of Clipboard	SHIFT+INS	---
Tab at cursor or beginning of each selected line	TAB	CTRL+I
Control character at cursor position	---	CTRL+P CTRL+*key*‡
Deleting		
Current line, saving in Clipboard	---	CTRL+Y
To end of line, saving in Clipboard	---	CTRL+Q Y
Character left	BKSP	CTRL+H
Character at cursor	DEL	CTRL+G
Word	---	CTRL+T
Selected text, not saving in Clipboard	DEL	CTRL+G
Selected text, saving in Clipboard	SHIFT+DEL	---
Leading spaces for one indentation level of selected lines	SHIFT+TAB	---
Copying		
Selected text, saving in Clipboard	CTRL+INS	---

Table 5.1 *(continued)*

Scrolling	Keypad Keys	WordStar-Style Key Combinations
Up one line	UP	CTRL+W
Down one line	DOWN	CTRL+Z
Page up	PGUP	CTRL+R
Page down	PGDN	CTRL+C
Left one window	CTRL+PGUP	---
Right one window	CTRL+PGDN	---
Selecting		
Character left	SHIFT+LEFT	---
Character right	SHIFT+RIGHT	---
Current line	SHIFT+DOWN	---
Line above	SHIFT+UP	---
Word left	SHIFT+CTRL+LEFT	---
Word right	SHIFT+CTRL+RIGHT	---
Screen up	SHIFT+PGUP	---
Screen down	SHIFT+PGDN	---
Screen left	SHIFT+CTRL+PGUP	---
Screen right	SHIFT+CTRL+PGDN	---
To beginning of module/ procedure	SHIFT+CTRL+HOME	---
To end of module/procedure	SHIFT+CTRL+END	---

† Replace *n* with a number from 0–3 to identify the desired marker.

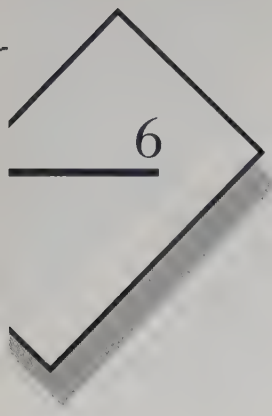
‡ The CTRL+P combination is followed by CTRL and the desired key.

CHAPTER 6

CREATING AND RUNNING QUICKBASIC PROGRAMS

6.1 Writing a Program....139
6.1.1 Creating a Main Module....140
6.1.2 Entering Your Program's Statements....140
6.1.3 Translating and Running in Memory....143
6.1.4 Returning to the Output Screen (Output Screen)....143
6.1.5 Testing Ideas in the Immediate Window....143
6.1.5.1 Characteristics of the Immediate Window....144
6.1.5.2 Statements Allowed in the Immediate Window....144
6.1.5.3 Doing Calculations....145
6.1.5.4 Testing Screen Output....145
6.1.5.5 Invoking Procedures....146
6.1.5.6 Changing Variable Values in a Running Program....146
6.1.5.7 Simulating Run-Time Errors....147
6.1.6 Programs with Command-Line Arguments (Modify COMMAND$)....148

6.2 Creating Executable Files from within QuickBASIC (Make EXE File)....148
6.2.1 Quick Libraries and Executable Files....151
6.2.2 Run-Time-Module Considerations....152
6.2.2.1 Using the Run-Time Module....152
6.2.2.2 Creating Stand-Alone Programs....152
6.2.3 Run-Time Error Checking in Executable Files....153
6.2.4 Floating-Point Arithmetic in Executable Files....153



CHAPTER

6.3 Using SUB and FUNCTION Procedures in Your Programs.....155
6.3.1 Creating a SUB or FUNCTION Procedure (New SUB, New FUNCTION).....155
6.3.2 Saving and Naming Procedures.....157
6.3.3 Viewing and Editing Modules and Procedures (SUBs).....158
6.3.4 Automatic Procedure Declarations 159
6.4 Combining Modules into a Program.....160

Whether your background is with BASIC interpreters or compilers, the preceding chapters provide all the information you need to begin writing programs in QuickBASIC. This chapter discusses the process of programming within the QuickBASIC environment and points out efficient ways to use QuickBASIC's power.

When you finish this chapter you will understand how to

- Enter program statements in QuickBASIC.
- Use the Immediate window to test ideas and modify data in a running program.
- Modify command-line arguments from within QuickBASIC.
- Create **SUB** and **FUNCTION** procedures.
- Interpret the **DECLARE** statements that QuickBASIC generates when you save a program.
- Use modules in addition to the main module.
- Combine modules into one complete program and then run that program.

6.1 Writing a Program

QuickBASIC differs from other BASIC-language products in many ways. Many of these differences are extensions to the BASIC language itself. However, the fundamental difference between QuickBASIC and other programming languages is in *how* you use it to write your programs. QuickBASIC includes a wider range of programming features than previous program-development environments, and all its features are completely integrated. That is, you don't have to quit one activity to do something else. For example, you can debug a program while you edit it.

The following list gives other examples of this seamless integration:

- Run what you have written at any time, no matter how complete or incomplete it is, just by pressing a key. Since each statement is translated to executable code as you enter it, there is no separate compiling step within the QuickBASIC environment.
- Suspend a running program, use debugging commands and techniques to analyze its behavior, edit the program, then continue from the point where you stopped. Thus, there is no separate debugging step.
- Write separate program modules, get them working individually, then load them together and run them as a single program.

Note

Once your program is debugged and running to your satisfaction, you can use the Run menu's Make EXE File command to create a stand-alone executable version of your program. In this case, you will notice that there is a separate compiling step, during which your program is also linked. See Section 6.2 for an explanation of how to create executable files from within QuickBASIC. See Chapter 9, "Compiling and Linking from DOS," for more information on how to compile outside of the QuickBASIC environment.

6.1.1 Creating a Main Module

In a single-module program, the main module *is* the program. Therefore, as soon as you have typed your first statement and have saved it, you have created a main module. In a multiple-module program, the main module contains the first statement to be executed when the program is run.

If you start QuickBASIC without specifying a file name, the title bar of the View window contains the word Untitled. If you save the contents of that View window, even if it is empty, the name under which you save it becomes the name of the main module of the program.

After you name the main module, the name appears in the title bar of the View window. It also appears on the left side of the status bar at the bottom of your screen, just beyond the word Main.

6.1.2 Entering Your Program's Statements

Using QuickBASIC, you can enter program statements just as you would with a text editor or a BASIC interpreter. However, each time you move off the line you are typing, QuickBASIC checks it for syntax errors. If there are no syntax errors, QuickBASIC translates the statement to executable code. If there are syntax errors, QuickBASIC displays a dialog box containing an appropriate error message.

Often you will see the error message "Syntax error," but some error messages are more specific. For example, if you were to enter the following line by mistake:

```
CALL proc1 ()
```

QuickBASIC would generate the message, "Expected: expression or BYVAL or SEG" (see Figure 6.1).

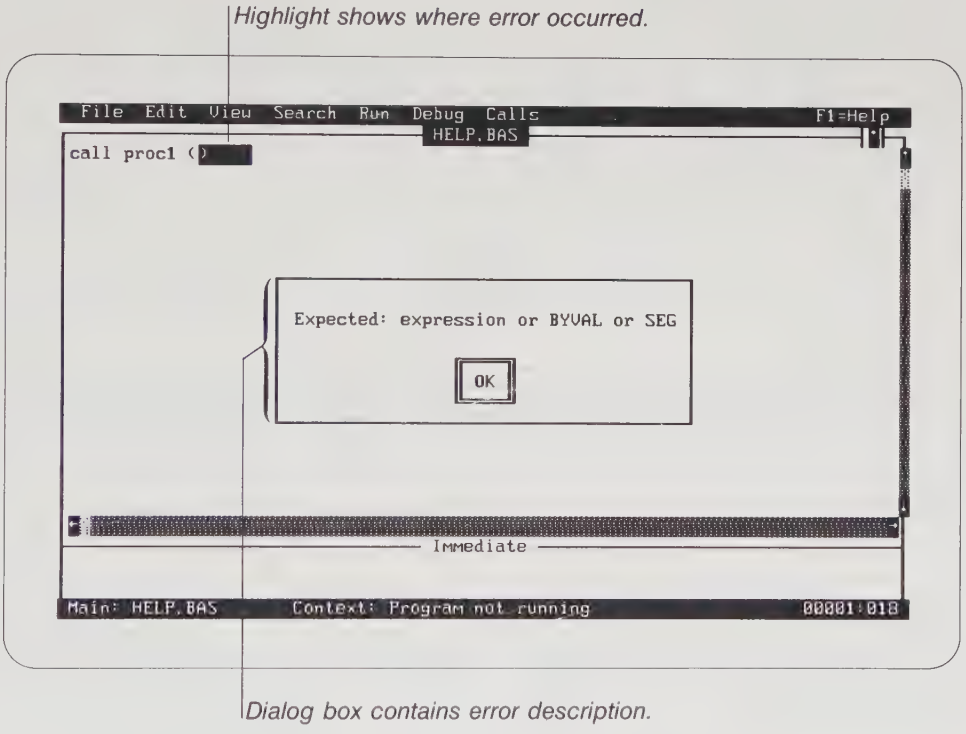


Figure 6.1 Error Message

This is because empty parentheses following a **SUB** procedure name are not valid syntax in a **CALL** statement (although they are perfectly valid in the **DECLARE** statement). If parentheses follow the name, QuickBASIC expects them to contain some argument, either in the form of an expression or an argument preceded by the keyword **BYVAL** or **SEG**. Dialog-box error messages are based on what you actually type, so if you make a typing error or confuse the syntax of two related statements, the resulting error message may seem confusing at first.

In the case above, if you place the cursor anywhere on the **CALL** keyword, then press SHIFT+F1, QuickBASIC displays a context-sensitive Help screen showing the syntax of the **CALL** statement (see Figure 6.2).

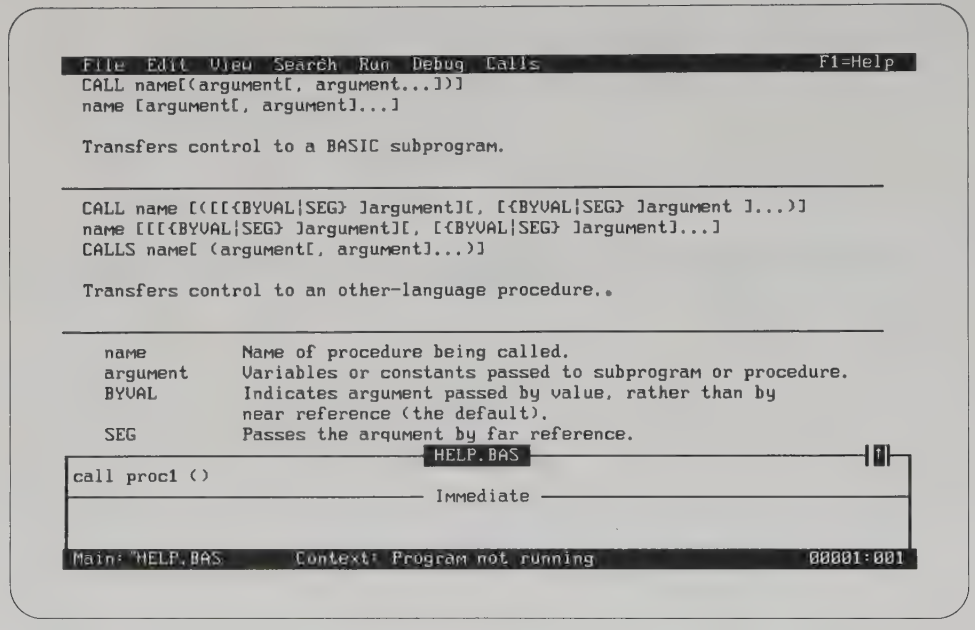


Figure 6.2 Context-Sensitive Help for CALL Statement

The syntax shows the **BYVAL** and **SEG** keywords used in the context of calling a procedure written in another language. Unless this was what you were trying to do, the only other possibility is a problem with parentheses. The general syntax at the top of the screen shows parentheses and arguments as optional, but if you use parentheses, the arguments are shown as mandatory. Therefore, removing the parentheses removes the error.

As a general rule, when QuickBASIC displays an error message that seems puzzling, you should use context-sensitive Help; or, consult the *BASIC Language Reference*, then evaluate the error message by comparing what you typed with the syntax of the statement.

6.1.3 Translating and Running in Memory

To test code you have entered in the View window, choose the Start command from the Run menu (or press SHIFT+F5). Because QuickBASIC has translated each line to executable code as it was entered, it translates the remainder of individual lines very quickly. Also, since everything is done in memory, QuickBASIC does not need to refer to disk files.

Even though each individual line of your program translates without any errors, you may still see error messages when you choose Start from the Run menu to run your program. For example, a **FOR** statement may not have a matching **NEXT** statement to complete the loop, or you may be trying to pass an illegal argument to a BASIC function. QuickBASIC does not detect these kinds of errors as you enter your program because they represent valid syntax. These errors only crop up when you try to run your program. See Appendix D, "Error Messages," in the *BASIC Language Reference*, if you get an error message other than "Syntax error" when you try to run your program.

6.1.4 Returning to the Output Screen (Output Screen)

When you run code that has screen output, QuickBASIC prints the message "Press any key to continue" on the last line of the output screen. After pressing a key and returning to the QuickBASIC editor, you can switch back and forth between the editor and the output screen by choosing the Output Screen command from the View menu (or pressing F4.)

6.1.5 Testing Ideas in the Immediate Window

The Immediate window is similar to "direct mode" in BASICA. In the BASICA direct mode, you can enter a single line containing one or more statements, then press ENTER to execute that line. This gives you the advantage of being able to execute statements directly without making them part of a program.

As you write your program, you can move to the Immediate window by pressing F6 until the Immediate window's title bar is highlighted. Once the Immediate window is active, you can type and edit statements as you would in the View window, then execute them immediately. The Immediate window provides a good way to test programming ideas before making them part of your program.

6.1.5.1 Characteristics of the Immediate Window

The Immediate window has the following characteristics:

- You can execute any statement or group of statements on a line by placing the cursor anywhere on the line and pressing ENTER.
- You can enter up to 10 lines in the Immediate window, then move back and forth among them, executing them in any order.

  After you have entered 10 lines, each new line scrolls the text in the Immediate window up one line.
- A line can have a maximum of 256 characters. Statements can be grouped on a line, but must be separated by colons (:).
- Each line is executed independently of all other lines in the window. However, lines can affect other lines. For example, the following lines contain two assignments to the integer variable x% , so the value printed by the second line corresponds to whichever assignment line was most recently executed:

  ```
  x% = 5
  print x%
  x% = 9
  ```
- You can expand the Immediate window by repeatedly pressing ALT+PLUS while it is the active window. Pressing CTRL+F10 while the Immediate window is active expands it to the full screen, but no matter how many lines there are, you can use only the first 10 lines.
- Code written in the Immediate window cannot be saved to disk unless you use the Cut or Copy commands from the Edit menu to move it into a View window so you can save it with the rest of the file.

6.1.5.2 Statements Allowed in the Immediate Window

Most QuickBASIC statements are allowed in the Immediate window, with the exception of those noted in the list below. For example, you can put an entire **FOR...NEXT** loop on one line:

```
for i% = 1 to 100 : ? i%, sqr(i%) : next i%
```

The following list shows the QuickBASIC keywords that you cannot use in the Immediate window:

COMMON	**ELSEIF**	**OPTION**
CONST	**END DEF**	**REDIM**
DATA	**END FUNCTION**	**SHARED**
DECLARE	**END IF**	**$STATIC**
DEF FN	**END SUB**	**STATIC**
DEF*type*	**END TYPE**	**SUB**
DIM	**FUNCTION**	**TYPE**
$DYNAMIC	**$INCLUDE**	

Note

Although the **END IF** statement is not allowed in the Immediate window, you can still use the single-line form of **IF...THEN...ELSE** there.

6.1.5.3 Doing Calculations

You can use the Immediate window to calculate complicated expressions, then display the result with the **PRINT** statement (or the ? shortcut character, which stands for **PRINT**—just as it does in BASICA). You can use any of BASIC's intrinsic functions such as **SIN** or **EXP** in these calculations.

You can also use the Immediate window to print the value returned by **FRE(–1)**. This tells you the amount of available memory when both QuickBASIC and your program are loaded.

6.1.5.4 Testing Screen Output

As a program grows in length and complexity, you can write code in the Immediate window and test it there first, before incorporating it into your program. This avoids running a program from the beginning each time you want to modify a small part of the code.

■ Example

You can use the following lines in the Immediate window to test the position of output to be written to the screen:

```
cls : row% = 1 : col% = 4*row% : LOCATE row%,col% : PRINT "."
```

By changing the value of the variable row%, then pressing ENTER, you can position the period in the **PRINT** statement in different places on the screen.

Sometimes a program can leave the output screen in an undesirable mode when it finishes running (for example, a graphics program may end without resetting the screen to 80-column text mode). If this happens, use the Immediate window to put the screen in the output mode you want. For example, the following lines restore the output screen to 80-column text mode and close any text viewport (a restricted horizontal slice of the screen) set in the program:

```
screen 0
width 80
view print
```

6.1.5.5 Invoking Procedures

You can isolate the effects of any individual **SUB** or **FUNCTION** procedure by calling it from the Immediate window. Use the Immediate window to execute only those statements between the **SUB...END SUB** or **FUNCTION...END FUNCTION** statements. For example, entering the following line in the Immediate window executes all statements in the procedure SubProgram1, including any calls it makes to other procedures:

```
call SubProgram1
```

6.1.5.6 Changing Variable Values in a Running Program

You can run a program up to a certain point, then stop it and use the Immediate window to assign a new value to a variable. You can then continue program execution. The value you assign in the Immediate window becomes the variable's value in the running program.

■ Example

The following steps show how you can use the Immediate window to change a variable in a running program:

1. Type this program in the View window:

```
FOR i% = 0 to 10000
   LOCATE 10, 20 : PRINT i%
   LOCATE 10, 27 : PRINT "  and still counting"
NEXT i%
```

2. Start the program, then use CTRL+BREAK to interrupt it.

3. Enter the following line in the Immediate window:

```
i% = 9900
```

4. Press F5 to continue the program.

   The loop now executes from 9900 to 10,000.

The context of each line of statements executed in the Immediate window is that of the highlighted statement in the active View window. For example, if the statement you just executed gave a variable called x% a value of 3, and you enter

```
PRINT x%
```

in the Immediate window, the number 3 is printed to the screen when you press ENTER.

Similarly, if the next statement to be executed in the View window is a module-level statement, you cannot access the variables of a **SUB** or **FUNCTION** from the Immediate window.

6.1.5.7 Simulating Run-Time Errors

Run-time errors, those that occur while a program is running, are associated with a numeric code. One way to find out which error message is associated with a given code is to refer to Appendix D, "Error Messages," in the *BASIC Language Reference*. Another way is to simulate the error itself. You can simulate an error by moving to the Immediate window and entering the given number as an argument to the **ERROR** statement. QuickBASIC then displays the dialog box for that error, as if the error had actually occurred in a running program.

6.1.6 Programs with Command-Line Arguments (Modify COMMAND$)

Many programs are written to process strings passed to them from the DOS command line. The **COMMAND$** function gives a program access to these command-line strings.

With this version of QuickBASIC, you can change any string passed to a program by choosing the Modify COMMAND$ command from the Run menu. Whatever you enter in the dialog box becomes the string returned by the **COMMAND$** function.

See the *BASIC Language Reference* for more information on command-line arguments and the **COMMAND$** function.

6.2 Creating Executable Files from within QuickBASIC (Make EXE File)

Once your program is running within QuickBASIC, you can make a version of the program that runs directly from the DOS command line, without QuickBASIC's support. The file for such a program is known as an executable file, and has the extension .EXE. Just entering the base name of an executable file after the DOS prompt runs the program.

When you create an executable file, QuickBASIC first uses the BASIC command-line compiler BC to compile your BASIC source code into an intermediate file known as an object file. QuickBASIC then uses a program called the Microsoft Overlay Linker (LINK) to join all the separately compiled modules of your program into a single executable file. LINK also combines the compiled object files created by BC with the supporting routines your program needs to execute properly. These routines are found in the run-time libraries BRUN40.LIB or BCOM40.LIB.

The following files must be available to QuickBASIC when you create an executable file. Make sure they are in the current directory or available through your PATH or LIB environment variables. (See Section 1.8 for more information on setting environment variables.)

Program	Purpose
BC.EXE	Creates linkable object files (files with .OBJ extensions) from your program.
LINK.EXE	Links the object files created by BC.
BRUN40.LIB	Provides supporting routines that allow your BASIC program to perform tasks such as accepting input or printing text on the screen. These routines are linked with your program by LINK.EXE. The BRUN40.LIB library is known as the run-time-module library. Any program linked with this library requires the presence of the run-time module, BRUN40.EXE, in order to run.
BCOM40.LIB	Does the same thing as BRUN40.LIB, except that executable files created with this library do not require the support of the run-time module, BRUN40.EXE. BCOM40.LIB is known as the alternate run-time-module library.

The following steps show you how to create an executable file from a program currently loaded in QuickBASIC:

1. Choose the Make EXE File command from the Run menu. The Make EXE File dialog box appears (see Figure 6.3).

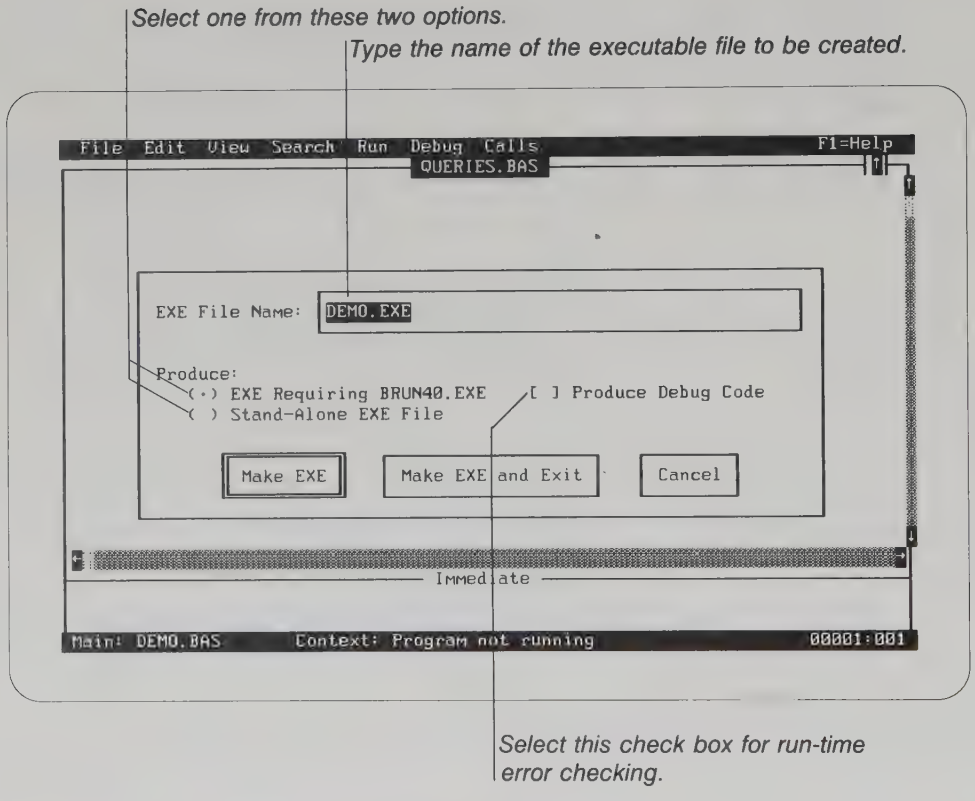


Figure 6.3 Make EXE File Dialog Box

2. Type a different base name in the text box if you want to rename your file; otherwise, leave the text box alone.

3. Select either the EXE Requiring BRUN40.EXE option or the Stand-Alone EXE File option (their respective advantages are fully discussed in Sections 6.2.2.1 and 6.2.2.2):
   - The EXE Requiring BRUN40.EXE option produces a small executable file, but it can only be run in combination with BRUN40.EXE, the run-time module.
   - The Stand-Alone EXE File option produces a file that runs without BRUN40.EXE.

4. Turn on the Produce Debug Code check box if you want your executable file to include code that generates error-handling messages and reports error locations at run time, or if you want your executable file to respond to CTRL+BREAK. Note, however, that this option does not produce CodeView-compatible executable files, and it also results in a larger and somewhat slower executable file. See Section 6.2.3, "Run-Time Error Checking in Executable Files," for an explanation of errors handled by this option.

5. Choose either the Make EXE or the Make EXE and EXIT command button:
   - The Make EXE command button creates the executable file, and then returns you to QuickBASIC.
   - The Make EXE and Exit command button creates the executable file, and then returns you to the DOS prompt.

6.2.1 Quick Libraries and Executable Files

If you used the /l command-line option to load a Quick library when you started QuickBASIC, the compiler searches for the stand-alone (.LIB) library that corresponds to the named Quick library when it creates an executable file.

Quick libraries provide fast access to your own custom library of procedures, but you can use them only inside the QuickBASIC environment. The compiler must have access to the corresponding stand-alone (.LIB) library to include the Quick library's procedures in an executable file. Therefore, it is a good idea to keep both kinds of libraries—Quick libraries and .LIB libraries—in the same directory. (The Make Library command on the Run menu makes both kinds of libraries automatically. See Chapter 8, "Quick Libraries," for more information on creating and using Quick libraries and Chapter 9, "Compiling and Linking from DOS," for an explanation of stand-alone libraries.)

6.2.2 Run-Time-Module Considerations

You can create two different types of executable files, depending on whether or not you want your executable program to require access to the run-time module BRUN40.EXE.

6.2.2.1 Using the Run-Time Module

Programs processed with the EXE Requiring BRUN40.EXE option need access to the run-time module BRUN40.EXE at run time. The run-time module contains code needed to implement the BASIC language. Using the run-time module provides the following advantages:

- The executable file is much smaller. If several programs are kept on the disk, considerable space is saved.
- **COMMON** variables and open files are preserved across **CHAIN** statements. This can be valuable in systems of programs that use shared data. In stand-alone programs, **COMMON** variables and open files are not preserved across **CHAIN** statements.
- The BRUN40.EXE run-time module resides in memory, so it does not need to be reloaded for each program in a system of chained programs.

6.2.2.2 Creating Stand-Alone Programs

When you use the Stand-Alone EXE File option, the executable file does not require access to the run-time module BRUN40.EXE. However, since files that are created with this option include the support routines found in BRUN40.EXE, they require more disk space than files created with the EXE Requiring BRUN40.EXE option. Also, files created with this option do not preserve variables listed in **COMMON** statements when a **CHAIN** statement transfers control to another program.

Files created with this option have the following advantages:

- You may save RAM space at run time if you have small, simple programs that do not require all the routines in the run-time module.
- Execution of the program does not require that the run-time module be on the disk at run time. This is important when you write programs that users will copy, since an inexperienced user may not know to copy the run-time module as well.

6.2.3 Run-Time Error Checking in Executable Files

If you turn on the Produce Debug Code check box in the Make EXE File dialog box when making an executable file, the following conditions are checked when the program is running:

- Arithmetic overflow. All arithmetic operations, both integer and floating-point, are checked for overflow and underflow.
- Array bounds. Array subscripts are checked to see if they are within the bounds specified in **DIM** statements.
- Line locations. The generated binary code includes additional tables so that run-time error messages can indicate the lines on which errors occur.
- **RETURN** statements. Each **RETURN** statement is checked for a prior **GOSUB** statement.
- CTRL+BREAK, the keystroke combination to halt program execution. After executing each line, the program checks to see if the user has pressed CTRL+BREAK. If so, the program stops.

  Note that you must compile a program with the Produce Debug Code check box turned on if you want it to respond to CTRL+BREAK. Otherwise, CTRL+BREAK halts a program only in the following two circumstances:

  1. When a user is entering data in response to an **INPUT** statement's prompt
  2. When the program's code explicitly checks for CTRL+BREAK

If you don't use the Produce Debug Code option, then array-bound errors, **RETURN** without **GOSUB** errors, and arithmetic overflow errors do not generate error messages. Program results may be unpredictable.

6.2.4 Floating-Point Arithmetic in Executable Files

When you compile a program into an executable file, the executable file does floating-point calculations more quickly and efficiently than the same program running in the QuickBASIC environment. This is because executable files are optimized for speed and accuracy by changing the order in which they do certain arithmetic operations. Executable files also make greater use of a numeric coprocessor (or emulation of a coprocessor's function) than do their in-memory counterparts.

One side effect of this extra accuracy is that you may notice a difference in the way a compiled program uses relational operators such as < or = to compare single- or double-precision values. For example, the following code fragment prints the word Equal when run in the QuickBASIC environment, but prints the words Not Equal when it is run as an executable file:

```
B!=1.0
A!=B!/3.0
.
.
.
IF A!=B!/3.0 THEN PRINT "Equal" ELSE PRINT "Not Equal"
```

This problem results from performing the calculation B!/3.0 inside a comparison. The compiled program stores the result of this calculation on the math coprocessor, which gives it a higher degree of accuracy than the value stored in the variable A!, causing the inequality.

You can avoid such problems in comparisons by doing all calculations outside of comparisons. The following rewritten code fragment produces the same results when run in the environment and as an executable file:

```
B!=1.0
A!=B!/3.0
.
.
.
Tmp!=B!/3.0
IF A!=Tmp! THEN PRINT "Equal" ELSE PRINT "Not Equal"
```

See Chapter 3, "Expressions and Operators," in the *BASIC Language Reference*, for more information on comparisons involving floating-point values.

6.3 Using SUB and FUNCTION Procedures in Your Programs

QuickBASIC supports two types of subordinate procedures, those that are defined within **SUB...END SUB** statements, and those that are defined within **FUNCTION...END FUNCTION** statements. This section describes how to create **SUB** and **FUNCTION** procedures. (See Chapter 2, "Procedures: Subprograms and Functions," in *Programming in BASIC: Selected Topics*, for more information on using procedures.)

Note

SUB and **FUNCTION** procedures are no longer permitted in include files, as they were in previous versions of QuickBASIC. See Appendix B, "Differences from Previous Versions of QuickBASIC," for information on adapting procedures originally used in include files for use in QuickBASIC modules.

6.3.1 Creating a SUB or FUNCTION Procedure (New SUB, New FUNCTION)

When you define a procedure, it becomes part of the module in the active View window. When you save that module as a file, any procedures it currently contains become part of the file. Procedures can be defined in the main module or other modules. If you decide after creating a procedure that you want it somewhere else, you can move it among modules, using the View menu's SUBs command. However, a procedure can appear in only one module of a program that invokes the procedure.

Procedures have their own commands on the Edit menu, shown in Figure 6.4. To create a **SUB** or **FUNCTION** procedure, follow these steps:

1. Choose the New SUB or the New FUNCTION command from the Edit menu (see Figure 6.4).

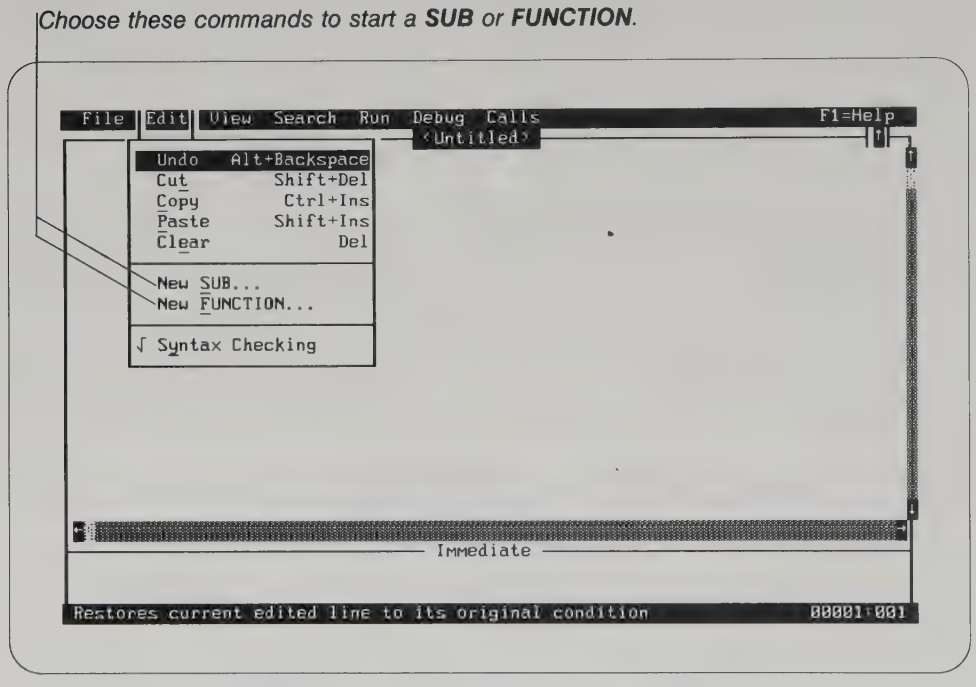


Figure 6.4 Edit Menu

2. Enter a name for the procedure in the text box.

3. Press ENTER.

   QuickBASIC opens a window and starts the procedure with the **FUNCTION...END FUNCTION** or **SUB...END SUB** statements, plus the name you entered in the text box.

When you start a procedure, the title bar identifies the contents of the View window with the name of the main module, followed by the name of your procedure.

There is a keyboard shortcut for creating procedures. Simply type either of the keywords **SUB** or **FUNCTION**, followed by the name of the procedure. When you press ENTER, QuickBASIC performs all the preceding steps automatically.

Important

When loading a program containing a **DEF***type* statement (other than the statement **DEFSNG**, the default BASIC data type) at the module level, QuickBASIC inserts a copy of your statement immediately above the **SUB** or **FUNCTION** statement in any procedure created within that module. If you later change your mind and want a different data type to prevail, simply replace the existing **DEF***type* statement with the one you want.

Executable statements and blank lines are not permitted above the **SUB** or **FUNCTION** statement in a procedure window. Therefore, if you create a blank line on which to enter a **DEF***type* statement (which is permitted), QuickBASIC displays an error message asking if a remark is OK. Press ENTER, and QuickBASIC places a comment delimiter at the beginning of the line. You can then delete or overtype the comment delimiter to enter your **DEF***type* statement.

If no **DEF***type* statement appears above the procedure definition, then the default data type for that procedure is single precision, because single precision is the default BASIC data type.

If you create a module and forget to put a **DEF***type* statement at the module level, any procedures you create within the module have single precision as their default data type, although there is no explicit indication of that above the procedure definition. If you later go back and put a **DEF***type* statement at the module level, it will not be copied to any pre-existing procedures. If you want a default data type other than single precision to prevail in such procedures, you must explicitly enter a **DEF***type* statement in the procedure. See Chapter 2, "Data Types," in the *BASIC Language Reference*, for more information on data types.

6.3.2 Saving and Naming Procedures

When you save your program, procedures are not saved as independent disk files the way modules are, so procedure names are governed by the naming conventions of QuickBASIC, rather than DOS. (Procedure names can be any valid BASIC name up to 40 characters long. The rules for naming procedures are the same as those for naming variables. See Chapter 2, "Data Types," in the *BASIC Language Reference*, for more information on variable names in BASIC.)

6.3.3 Viewing and Editing Modules and Procedures (SUBs)

You can see a list of modules and procedures in the current program by choosing the View menu's SUBs command (or pressing F2). Figure 6.5 shows the SUBs dialog box.

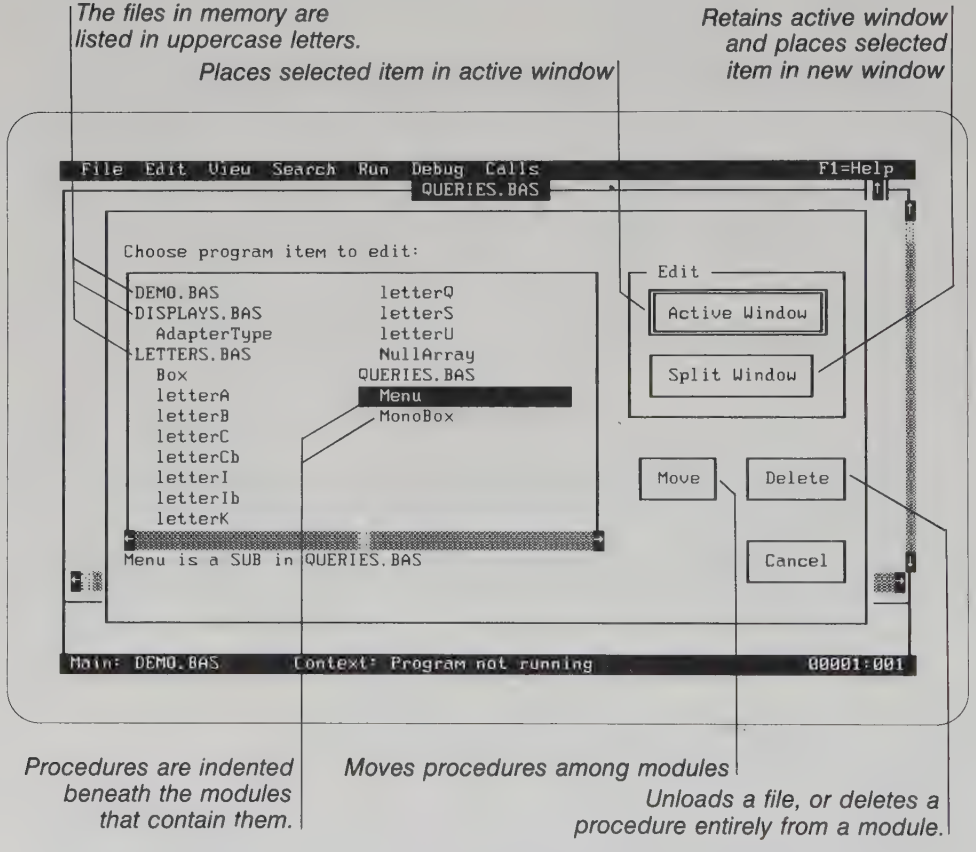


Figure 6.5 SUBs Dialog Box

The names of procedures are displayed in the list box, indented beneath the modules in which they are defined. To edit a **SUB** or **FUNCTION** procedure, move it into a View window as follows:

1. Move the highlight to the procedure's name.
2. Choose either the Active Window or the Split Window command button:
   - Active Window replaces the currently active window with the text of the chosen item.
   - Split Window puts the chosen item in a second window, if only one window was open. If two windows were previously open, the new item replaces whichever window was inactive.

If a module contains several procedures, you can cycle through them in alphabetical order by repeatedly choosing the Next SUB command from the View menu (or by using the shortcut-key sequence SHIFT+F2 to cycle forward, or CTRL+F2 to cycle backward).

6.3.4 Automatic Procedure Declarations

When you save a program as a disk file (or, if it is a multiple-module program, as a collection of disk files) QuickBASIC creates a declaration for each procedure in a program. A declaration consists of the **DECLARE** keyword, followed by either the **SUB** or **FUNCTION** keyword, then the name of the procedure, and a list of its formal parameters.

See Chapter 2, "Procedures: Subprograms and Functions," in *Programming in BASIC: Selected Topics*, and Chapter 2, "Data Types," and Chapter 4, "Programs and Modules," in the *BASIC Language Reference*, for more information on the purposes of declarations and how to use them in programs.

■ Example

In the next example, QuickBASIC generates the first statement when you save the program with the Save As command from the File menu.

```
DECLARE SUB DemoDec (First$, Second%)     ' This declaration at
                                          ' the module level
CALL DemoDec ("Hi there", 2)              ' is a result of this
                                          ' invocation
SUB DemoDec (First$, Second%) STATIC      ' of this procedure
   PRINT First$ " has " Second% "words"   ' definition.
END SUB
```

6.4 Combining Modules into a Program

A module is an individual disk file containing BASIC statements. A module can be a self-contained program, or it can be a file containing one or more **SUB** or **FUNCTION** procedures that are invoked from other modules.

With traditional compiler/linker program-development systems, the only way to create a multiple-module program is to compile each module as a separate object file, then link those object files to form one executable file on disk. However, in QuickBASIC you can create multiple-module programs in *memory* and run them just as you would a single-module program. For example, a new program can use several existing modules, each of which contains several procedures you want to use in the program.

The following steps show how to combine procedures from separate modules into a new program:

1. Start QuickBASIC, giving as a command-line argument the name you want your program to have. For instance, type

   ```
   qb program
   ```

   This will be the name of your main module.

2. Use the Load File command from the File menu to load each module containing procedures invoked by the new program.

3. Write the code for your main module. You can call procedures in any of the modules currently loaded.

4. Make sure that any include files needed by the program are in the current working directory, or that the $**INCLUDE** metacommand specifies either a full or relative path to the include file, as in the following examples:

```
' $INCLUDE: '..\include\graphics.bi'
' $INCLUDE: 'c:\include\graphics.bi'
```

5. Choose Start from the Run menu.

   Be sure the program runs as you want it to.

6. Choose Save All from the File menu to save everything as a single program.

In reality, there are numerous intermediate steps involved in getting several modules working as one program. For example, if one of the non-main modules contains a **FUNCTION** procedure, QuickBASIC generates the **DECLARE** statement within that module but you must copy the **DECLARE** statement into any module in which the procedure is invoked before it will work properly. A convenient way to do this is to copy all **DECLARE** statements into an include file, then use the $**INCLUDE** metacommand to include them in modules where they are needed. See Chapter 4, "Programs and Modules," in the *BASIC Language Reference*, for the language rules governing multiple-module programs. See Chapter 2, "Procedures: Subprograms and Functions," in *Programming in BASIC: Selected Topics*, for rules governing procedures.

CHAPTER 7

DEBUGGING WHILE YOU PROGRAM

7.1 Debugging with QuickBASIC.....165

7.2 Preventing Bugs.....166

7.3 QuickBASIC's Debugging Features.....168

7.3.1 Debugging Terms and Concepts.....168

7.3.2 The Debug Menu.....170

7.3.3 Tracing Program Execution: Beyond TRON and TROFF.....171

7.3.3.1 Single Step (F8).....171

7.3.3.2 Procedure Step (F10).....171

7.3.3.3 Animated Trace (Trace On).....171

7.3.3.4 History On.....172

7.3.3.5 History Back (SHIFT+F8).....172

7.3.3.6 History Forward (SHIFT+F10).....172

7.3.4 Breakpoints, Watchpoints, and Watch Expressions.....172

7.3.4.1 The Watch Window.....173

7.3.4.2 Breakpoints (F9).....175

7.3.4.3 Watchpoints.....175

7.3.4.4 Watch Expressions: Beyond PRINT.....176

7.3.5 Controlling Execution.....178

7.3.5.1 Continue (F5).....178

7.3.5.2 Executing to the Cursor (F7).....179

7.3.5.3 Set Next Statement.....179

7.3.5.4 Next Statement.....180

7.3.5.5 Restart.....180

7.4 Advanced Debugging.....180

7.4.1 The Calls Menu.....180

7.4.2 Compatibility with the CodeView® Debugger.....182

Chapter 5, "Editing," and Chapter 6, "Creating and Running QuickBASIC Programs," showed you how QuickBASIC catches syntax errors within a single line as you enter your program. This chapter shows you how QuickBASIC helps you track down other errors, those that result from flaws in the actual logic of a program.

When you finish this chapter you will understand how to perform the following debugging tasks:

- Use QuickBASIC to prevent bugs as you are writing your program.
- Isolate and correct bugs using QuickBASIC to:
  - — Trace program execution one statement at a time.
  - — Execute your program until it reaches a specified line or lines.
  - — Track variable values during program execution.
  - — Suspend execution temporarily when a variable or expression reaches a specified value or condition.
  - — Substitute new values, expressions, and whole sections of code within a suspended program, then continue the program.
  - — Replay the last 20 lines executed.

7.1 Debugging with QuickBASIC

Sooner or later, every programmer has to deal with debugging, the process of rooting out problems that either keep a program from running at all, or cause it to behave in undesirable ways. No matter how good a programmer you are, chances are you will find yourself debugging a program that looked—on paper—as though it would run just fine.

The debugging techniques you have used in the past for your BASIC programs might range from observation of a program's behavior and sophisticated guesswork to actual use of the debugging tools available with that version of BASIC. If you have programmed in BASICA, your tools were the **STOP**, **PRINT**, **CONT**, **TRON**, and **TROFF** statements. If you have programmed with a compiled version of BASIC, you may have debugged your compiled programs with a tool known as a "symbolic debugger," which allows you to track the flow of your program's logic while simultaneously watching what it does.

Both of these kinds of tools are limited. In BASICA, there is no way to monitor a variable or expression as your program is running, and any changes to a suspended program force you to restart the whole thing.

Symbolic debuggers are powerful tools, but before you can use a symbolic debugger, you must successfully compile your program. This compiling process can involve numerous iterations of the following loop:

- Attempting to compile your program
- Getting an error, accompanied by occasionally cryptic compilation messages
- Deciphering the error message, then editing your source file
- Recompiling

All of the above just gets you to the point of being able to use the symbolic debugger. Each time you isolate a problem with the debugger, you are thrown back into the loop described above.

As you will see, QuickBASIC combines the ease of debugging in BASICA with features of symbolic debuggers such as breakpoints, watchpoints, and watch expressions. Most importantly, you do not have to load a separate tool or invoke QuickBASIC with a special option to start debugging your program. When you have a bug, you can start tracking it immediately. Everything you need is right in the QuickBASIC environment.

Additionally, with QuickBASIC you don't have to recompile or relink your program when you change it. Most of the time you can run the program to the point where the problem occurs, suspend execution, fix the problem, and then continue running the program from the point where you stopped it. In QuickBASIC, you debug as you program.

7.2 Preventing Bugs

The best way to have a bug-free program is to prevent bugs before they happen. The following list describes three ways you can prevent bugs:

1. Design your program carefully before you even start typing.

   An important rule in good program design is to isolate different tasks that your program will perform—tasks such as sorting an array or tokenizing a line (getting individual words after stripping out punctuation and spaces)—and then write and debug **SUB** or **FUNCTION** procedures that perform those tasks. A small procedure is much easier to debug than a large program that doesn't use procedures.

   An additional benefit of procedures is that you can save them in a separate file called a module, then use them in other programs that perform the same tasks. (See Chapter 2, "Procedures: Subprograms and Functions," in *Programming in BASIC: Selected Topics*, and

Chapter 4, "Programs and Modules," in the *BASIC Language Reference*, for more information on procedures and modules.)

2. Use the Immediate window.

   As you program, you can use the Immediate window at the bottom of the QuickBASIC screen to isolate and test small pieces of code. When you get these pieces working well on their own, you can then move them up into the main body of the program. See Section 6.1.5 for more information on using the Immediate window.

3. Run your code often.

   Because QuickBASIC checks and translates each statement to executable code as you enter it, you can constantly test each new piece of code in the context of the rest of the program. This helps you catch simple bugs that could be a nuisance to track in a finished program.

   For example, if you isolated the loop in the following fragment, you would probably notice the error very quickly (the variable I% is not being incremented because the I refers to a completely different variable):

```
DIM Array$(1 TO 20)
.
.
.
I% = 1
DO
   INPUT Temp$
   IF Temp$ <> "" THEN
      Array$(I%) = Temp$
      I% = I + 1
   END IF
LOOP UNTIL Temp$ = "" OR I% > 20
```

   One way to test this would be to load the array with some string expressions, then type the following in the Immediate window:

```
FOR I% = 1 TO 20: PRINT Array$(I%): NEXT
```

   The output will be a big clue that something needs to be fixed.

   Another technique involves setting a "watch expression." That is, you can watch the value of the variable I% as you trace through the loop. Section 7.3.4.4 tells you how to watch an expression while a program is executing.

7.3 QuickBASIC's Debugging Features

This section describes QuickBASIC's fully integrated debugging features. Section 7.3.1 defines some key debugging terms and concepts. Section 7.3.2 presents the Debug menu. Sections 7.3.3–7.3.5 explain how to use QuickBASIC's debugging commands.

As you read these sections, keep in mind that these features are always available as you develop a program within the QuickBASIC environment. You don't have to switch back and forth between the editor and a debugging tool, or choose a special debugging mode.

7.3.1 Debugging Terms and Concepts

Before reading further, you should understand the following QuickBASIC debugging terms and concepts.

- **Tracing**

  Tracing lets you see which statement in your program is executing. This lets you see, for example, which branch of an **IF...THEN...ELSE** statement is taken. QuickBASIC includes tracing commands that let you do the following:

  — Run one statement at a time.

  — Animate execution so each statement appears highlighted as it executes.

  — Trace backward or forward through the last 20 statements executed.

- **Breakpoint**

  A breakpoint is a location in your program where you want execution to pause. With breakpoints, you can test which parts of a program are being run. You can also use breakpoints to stop the program at strategic places to examine a variable's value so you can see exactly what has happened.

- **Watchpoint**

  A watchpoint is an expression that stops a program when the expression becomes true (or nonzero).

- **Watch Expression**

  Watch expressions let you observe the values of variables or expressions in your program while it executes.

- **Watch window**

  The Watch window is the window that opens at the top of the QuickBASIC screen to let you track watchpoint conditions or variable values during program execution. See Section 7.3.4.1 for more information on the Watch window.

- **Context**

  The term "context" refers to the part of your program that is currently executing. The name of this part (or the words "Program not running," if appropriate) appears on the status bar at the bottom of the QuickBASIC screen, after the word "Context".

  The context of a watchpoint or watch expression refers to the part of your program that is in the active window when you add the expression to the Watch window. For example, suppose your program contains a **FUNCTION** procedure named Parser, and you want to monitor the value of a variable named Counter inside Parser. If you add the variable Counter to the Watch window when the procedure named Parser is in the active window, you will be able to see how Counter changes as your program is executing the Parser procedure. In fact, the name Parser precedes the Counter entry in the Watch window, because the debugging context of the variable Counter is the Parser procedure.

  The current value or condition of a variable or expression can only be displayed in the Watch window when the item's debugging context (shown in the Watch window) is the same as the program's context of execution (shown on the status bar). Otherwise, the message "Not watchable" is displayed.

Note

Even if your program uses a **SHARED** statement to share a variable between the module level and the procedure level, that variable is watchable only at the level that is active when you add the variable to the Watch window.

Nothing is watchable while a **DEF FN** function is active.

- **Immediate window**

  The Immediate window is the window at the bottom of the screen where you can execute QuickBASIC statements directly. The current context of execution determines which of a suspended program's variables are accessible within the Immediate window, and their default data types. For example, if the next statement to be executed in the suspended program is an **END FUNCTION**

statement, variables local to the function are accessible from the Immediate window. However, once the statement is executed, they are no longer accessible. See the entry for **DEF***type* in the *BASIC Language Reference* for an explanation of how QuickBASIC and BASICA differ in handling default data types.

7.3.2 The Debug Menu

Although the View, Run, and Calls menus contain commands that you often use when debugging, most of QuickBASIC's debugging commands appear on the Debug menu, shown in Figure 7.1.

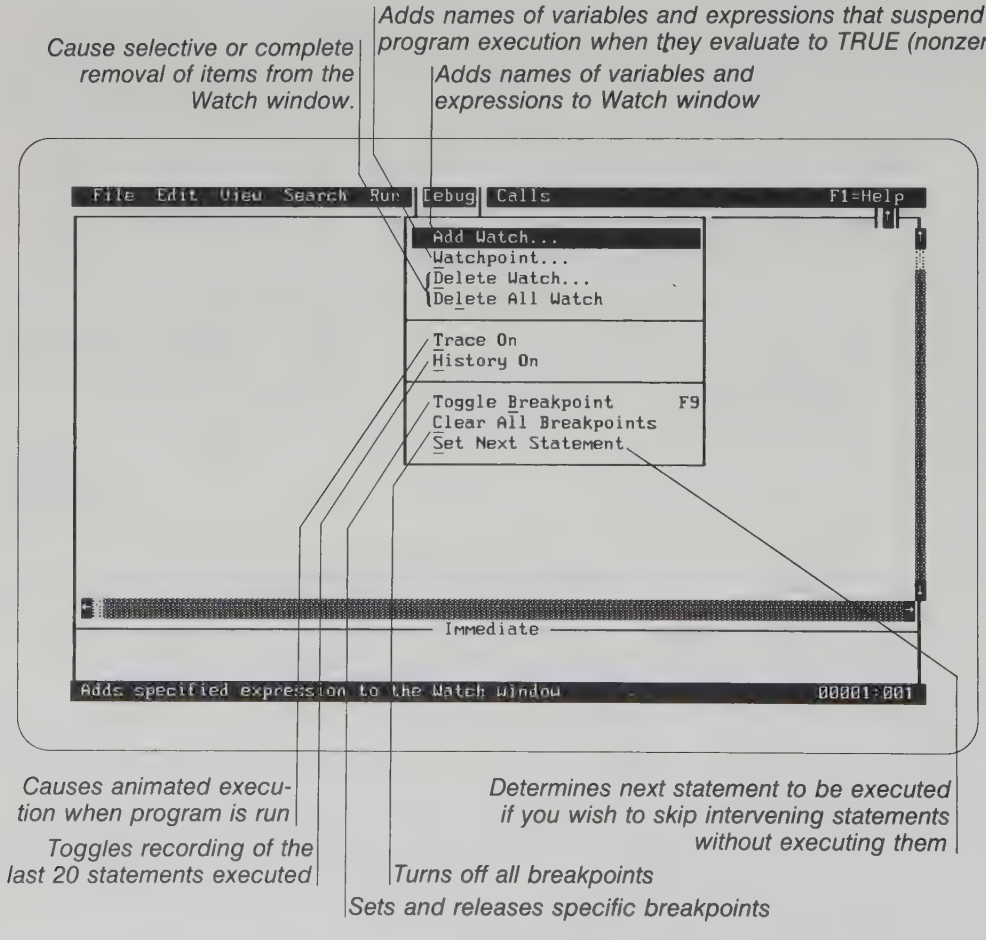


Figure 7.1 The Debug Menu

7.3.3 Tracing Program Execution: Beyond TRON and TROFF

Tracing allows you to watch your program's execution sequence. QuickBASIC includes several variations of tracing for different purposes. QuickBASIC's tracing capabilities are more powerful than what you get by using **TRON** and **TROFF** in BASICA.

7.3.3.1 Single Step (F8)

Activated with the F8 key, single step executes one statement of your program each time you press it. If the statement produces screen output, it appears briefly before QuickBASIC returns to the environment. (You can switch back and forth between the output screen and the QuickBASIC environment by choosing the Output Screen command or pressing F4.)

Single stepping traces through every line of **SUB** or **FUNCTION** procedures as they are invoked in your program.

7.3.3.2 Procedure Step (F10)

Activated with the F10 key, procedure stepping is similar to single stepping, except that it executes a procedure invocation as if the whole procedure were a single statement. This is sometimes referred to as "stepping over" a procedure.

With procedure stepping, all the statements in the procedure are executed, but you only see the result of the procedure call.

7.3.3.3 Animated Trace (Trace On)

You can turn on animation by choosing the Trace On command from the Debug menu. If you subsequently choose the Start or Continue command from the Run menu while the Debug menu's Trace On command has a check mark (√) next to it, QuickBASIC single steps through your program in continuous slow motion. Each statement is highlighted as it is executed. Used in combination with watchpoints, breakpoints (discussed in Section 7.3.4.2), and the CTRL+BREAK key combination (which suspends execution), this is a useful method for determining whether the general flow of your program is what you intended.

Note

The Trace On toggle command does not affect the action of the F8 and F10 tracing commands. They continue to step as usual.

7.3.3.4 History On

Turning on Trace On causes QuickBASIC to record the last 20 lines executed by your program, although you can also turn on this feature independently by choosing the History On command. History On allows you to trace backwards and forwards through the last 20 statements executed by your program. This is very useful when you get a run-time error and you want to see the effect of the 20 statements immediately preceding the error. Used in combination with the History key combinations (described in Sections 7.3.3.5–7.3.3.6), this type of stepping also allows you to follow the specific branching of a program through nested sequences of conditional structures like **IF...THEN...ELSE**. When History On is toggled on, execution speed is slower, though execution is not animated as with Trace On.

7.3.3.5 History Back (SHIFT+F8)

When either the Trace On or History On commands are checked, you can interrupt execution of your program by pressing CTRL+BREAK, then pressing SHIFT+F8 to step backwards through any of the last 20 statements executed.

7.3.3.6 History Forward (SHIFT+F10)

After stepping back through statements using the History Back keys, you can step forward through the same sequence of statements by pressing SHIFT+F10 once for each statement.

7.3.4 Breakpoints, Watchpoints, and Watch Expressions

You can set breakpoints and watchpoints to suspend execution of your program when certain events or conditions occur. You can set a watch expression to monitor the values of variables or expressions as your program executes.

7.3.4.1 The Watch Window

When you set a watchpoint or add a watch expression, the Watch window opens at the top of the screen. The name of each variable or expression that you are monitoring appears in the Watch window.

Figure 7.2 shows the QuickBASIC screen with entries in the Watch window, two View windows, and the Immediate window.

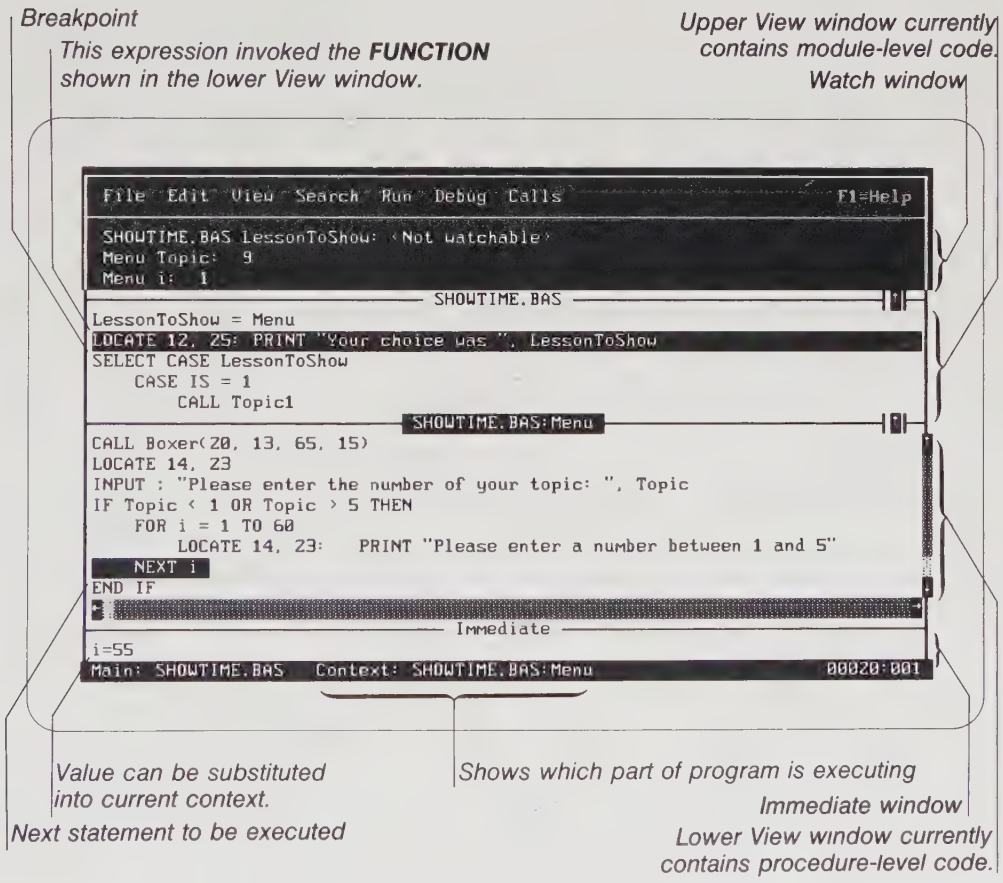


Figure 7.2 Debugging Features

The Watch window opens whenever you choose either the Add Watch or the Watchpoint command from the Debug menu. Each entry in the list contains the following information:

- The context of the expression.

  The context is either a module name (if you choose Add Watch or Watchpoint while module-level code is in the active View window) or a procedure name (if you choose Add Watch or Watchpoint while procedure-level code is in the active View window). Note that the context of an expression as shown in the Watch window remains constant, though the context (of execution) shown on the status bar changes according to the flow of the program. When these two contexts are different, the words "Not watchable" appear in the Watch window. See Section 7.3.1, "Debugging Terms and Concepts," for more information on the context of a watch expression or watchpoint.

- The expression you want to track.

- The value or condition (true or false) of the expression for as long as the flow of the program's logic is within the context of the expression. Note that since the program shown in Figure 7.2 is currently executing within the Menu procedure, "Not watchable," rather than a value or the words TRUE or FALSE, follows the variable LessonToShow in the Watch window. This occurs because the variable LessonToShow was entered in the watch window while the module-level code of SHOWTIME.BAS was in the active View window. Therefore, the debugger cannot access the variable while the current context of execution (shown on the status bar) is at the procedure level. However, both the variables Topic and i were entered into the Watch window while the Menu procedure was in the active View window, so their current values appear in the Watch window.

Program execution speed slows noticeably when there are many items in the Watch window.

The upper View window currently shows the module-level code in the main module of the program SHOWTIME.BAS. The highlighted line in this window is a breakpoint that will suspend execution of the program when control returns from the Menu procedure.

The lower View window shows the code of the Menu procedure. This second View window is now the active window. The highlighted statement in this window is the one that the program will execute next, and therefore determines the context shown on the status bar.

Statements in the Immediate window have the context of the next statement executed in the program. Therefore, if the user moved to the Immediate window at this point and executed the statement shown there, the loop counter in the Menu procedure's **FOR...NEXT** loop would assume the value 55. Such a technique is handy when editing the program in the View window would require you to restart the program.

The status bar at the bottom of the screen reflects the current state of program execution.

7.3.4.2 Breakpoints (F9)

A breakpoint is a specific location in your program's flow of control where you want to suspend execution. You can use a breakpoint to pause your program at strategic points where you suspect you may have a problem, then test the values of variables to confirm or refute your suspicion.

Set or clear a breakpoint with the Toggle Breakpoint command from the Debug menu (or press F9). The lines at which breakpoints are set are shown in red, or reverse video. Use the Clear All Breakpoints command if you want to disable all the breakpoints you have previously set in a program.

7.3.4.3 Watchpoints

Set a watchpoint with the Watchpoint command from the Debug menu. Don't confuse the Watchpoint command with the Add Watch command, which shows the actual value of a variable or expression during execution. (See Section 7.3.4.4 for a description of the Add Watch command.)

You can enter any variable name or expression in the Watchpoint dialog box, and the expression can contain any of QuickBASIC's relational operators (such as <>). If you only want to see when an expression switches from a zero value to a nonzero value, you don't have to use the <> relational operator explicitly. You can just add the expression to the Watch window, and BASIC suspends execution of the program as soon as the expression becomes a nonzero value. (BASIC always considers a nonzero value true, and a zero value false. See Chapter 1, "Control-Flow Structures," in *Programming in BASIC: Selected Topics*, for more information.)

If you set a watchpoint while a **SUB** or **FUNCTION** procedure is in the active View window, then the debugger describes the given expression as either TRUE or FALSE only when the program is executing that procedure. When the program is executing in another context, the message "Not watchable" appear beside the item.

Watchpoints are especially useful when you know a variable is taking on an unexpected value, but you don't know where this is happening in the flow of your program. It is very easy for a BASIC program with many **GOSUB** subroutines to alter the value of a variable. For example, if the UpdateWindow subroutine in the following loop accidentally uses a variable named I% , then any changes that occur to I% in UpdateWindow also affect this loop:

```
FOR I% = 1 TO 20
   GOSUB UpdateWindow
NEXT I%
```

In this case, if the loop ends unexpectedly, the user can add the watchpoint I% <> 1, and then continue execution by pressing F5. As soon as the expression I% <> 1 is true, program execution pauses, and the user knows the exact line where the variable is being inadvertently changed.

7.3.4.4 Watch Expressions: Beyond PRINT

Use the Add Watch command from the Debug menu to display the actual values of variables or expressions in the Watch window.

The capability to watch expressions saves you the trouble of executing **PRINT** statements repeatedly when tracking the value of a variable.

When you watch an expression during program execution, you see its value change immediately. Sometimes the words "Not watchable," rather than a value, follow the expression's name. This means that the debugger cannot access the variable at that point, because the variable was entered in the Watch window when a different part of the program was in the active View window.

Frequently it is not necessary to use the Watch window. If you are interested only in one instance of a variable's value, you can just enter its name as the argument to a **PRINT** statement in the Immediate window.

■ Example

The following example calculates the *n*th number in a sequence known as the Fibonacci sequence. In the Fibonacci sequence, each number is the sum of the two preceding numbers. The first two numbers in the sequence are 1 and 2. Here are the first 10 numbers in the sequence:

1, 2, 3, 5, 8, 13, 21, 34, 55, 89

Follow these steps to see a demonstration of animated tracing with a watch expression:

1. Start QuickBASIC, then choose the New Program command from the File menu and type the following lines:

```
DEFINT A-Z
INPUT N
PRINT Fib(N)
END
```

2. Choose the Add Watch command from the Debug menu, then enter N in the text box. This displays the value N has at the module level.

3. Create a **FUNCTION** procedure called "Fib" by choosing the New FUNCTION command from the Edit menu. (Note that DEFINT A-Z is automatically added above the first line of the **FUNCTION** definition because of the **DEF***type* statement you typed in step 1.) Now add lines so the procedure appears as follows:

```
FUNCTION Fib (N)
   IF N > 2 THEN
      Fib = Fib(N - 1) + Fib(N - 2)
   ELSE
      Fib = N
   END IF
END FUNCTION
```

4. Choose the Add Watch command from the Debug menu, then add N to the Watch window. This displays the value N has within the Fib procedure.

5. Turn on the Trace On command from the Debug menu.

6. Choose Start from the Run menu (or press SHIFT+F5) to execute the program.

7. Enter the value 8 after the input prompt.

8. Observe the Watch window to see how the value of N changes as the procedure recursively calls itself to calculate the eighth number in the Fibonacci sequence. (See Chapter 2, "Procedures: Subprograms and Functions," in *Programming in BASIC: Selected Topics*, for more information about recursive procedures.)

9. Choose the Output Screen command from the View menu (or press F4) to see the final value calculated.

To see how the History Back and History Forward keys work, follow these steps with the program given above:

1. Turn off the Trace On command from the Debug menu.
2. Turn on the History command from the Debug menu.
3. Set a breakpoint at the END FUNCTION statement.
4. Run the program.
5. Press SHIFT+F8 to backtrack through the last 20 statements when execution stops at the breakpoint you have set. Then press SHIFT+F10 to step forward through the same 20 statements.

7.3.5 Controlling Execution

With QuickBASIC's execution-control commands, you can run your program, suspend its execution, then continue. These commands allow you to execute from one statement to another, either with or without executing the statements in between.

7.3.5.1 Continue (F5)

Choose the Continue command from the Run menu or press F5 to start your program. When your program starts, it either continues from the last statement executed if execution is currently suspended, or starts from the beginning. The state of a program is shown in the center of the status bar on the bottom of the QuickBASIC screen after the word Context. If the program is running or suspended, Context displays the name of the current module (and procedure, if applicable). If the program is not running, that fact is shown as Context: Program not running.

If the Trace On command is turned on, the program is executed in slow-motion animation; otherwise, it is executed at normal speed. If the Watch window is open, or if the History On toggle is on, the program is executed at slower than normal speed, but is not animated unless Trace On is also toggled on.

You typically use the Continue command to start a program and then execute from one breakpoint or watchpoint to another. While execution is suspended, you can analyze the code and then make corrections to it, if necessary. When the corrections are made you can press F5 again to continue execution from that point. If your edit is such that the program can't continue if it incorporates the changes you have made, QuickBASIC issues a warning and asks if you want to continue without the changes, or restart the program from the beginning with the changes. If you don't want to restart, you can usually make necessary changes to program data temporarily in the Immediate window, then continue.

7.3.5.2 Executing to the Cursor (F7)

Pressing the F7 key executes your program from the current line to the line on which the cursor rests, provided that the line where the cursor rests is in the flow of the program's execution. That is, if the cursor is on an unreachable line (for example, a line within a procedure never invoked by the program or a line that has already been executed by the program), then pressing F7 is equivalent to pressing F5 (continue from the last statement executed).

Note

Pressing the right mouse button executes your program to the line on which the mouse pointer rests.

7.3.5.3 Set Next Statement

The Set Next Statement command from the Debug menu changes the execution sequence of your program so that the next statement executed is the one at the cursor. It does not execute the program to that statement, it simply jumps to it, without executing any intervening code. The effect is similar to that of a **GOTO** statement.

The same restrictions that apply to the **GOTO** statement apply to the Set Next Statement command. For example, you cannot branch to the middle of a procedure from the module level of the program.

7.3.5.4 Next Statement

The Next Statement command from the View menu places the cursor at the next statement to be executed when the program continues running. It does not execute the statement. Next Statement is useful when you have moved around in a suspended program and want to return to the point of suspension without executing the statement. In a program that is not running, the Next Statement command simply places the cursor at the program's first statement.

7.3.5.5 Restart

The Restart command from the Run menu resets all variables to zeros or null strings and highlights the first executable statement in your program. It is generally used when you don't want to continue execution. It is especially useful if you want to restart a program from the beginning and step through code that occurs prior to the first breakpoint or watchpoint. If you simply want to restart and execute to the first breakpoint or watchpoint, choose the Start command from the Run menu.

7.4 Advanced Debugging

As your programs increase in complexity, you will find more and more ways of using QuickBASIC's debugging features. Two of the most powerful are the Calls menu, and the ability to create executable files that are compatible with the Microsoft CodeView debugger.

7.4.1 The Calls Menu

The Calls menu differs from QuickBASIC's other menus in that it does not contain commands. If your programs do not include calls to procedures from within other procedures, it never shows anything except the name of your program's main module and the currently executing procedure.

If your program invokes procedures from within other procedures, the Calls menu shows those interprocedural calls. For example, you might have a sequence of invocations in which the module-level code (here represented by the program name, CALLSTAK) calls Procedure1, then Procedure1 calls Procedure2, Procedure2 calls Procedure3, and Procedure3 calls Procedure4. The Calls menu would then look like the one in Figure 7.3.

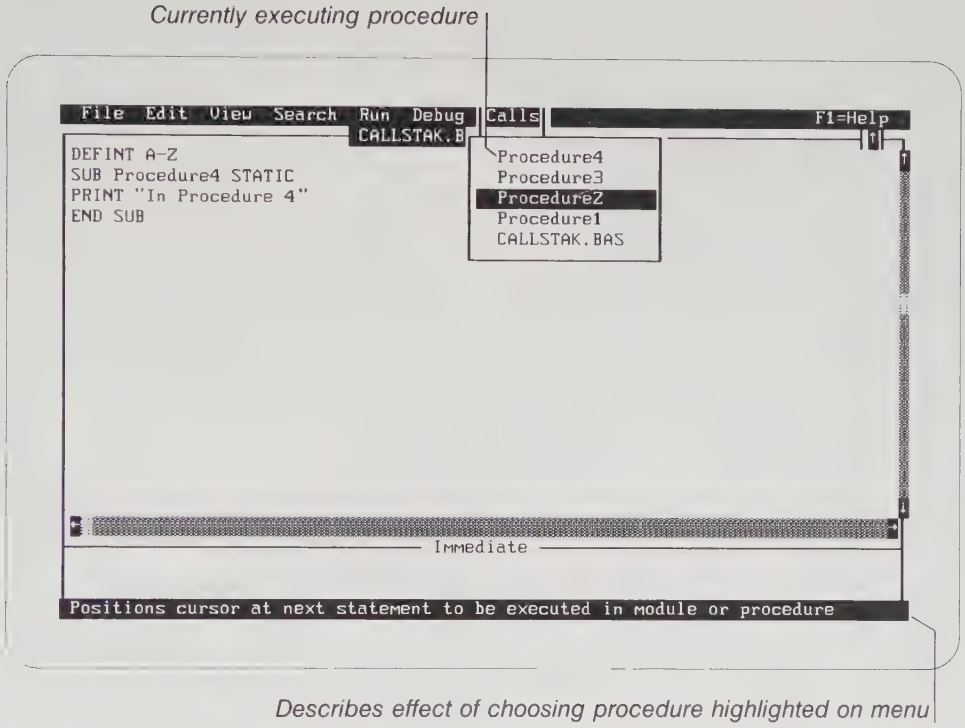


Figure 7.3 Sequence of Procedures on Calls Menu

The procedure at the top of the menu is the currently active procedure, and the one directly below it is the procedure to which it will return when it executes its **END SUB** or **END FUNCTION** statement. You can execute your program to any one of those procedures simply by choosing the procedure's name from the Calls menu, and then pressing F7. Therefore, in the illustration, since Procedure2 is highlighted, pressing ENTER, then pressing F7, would execute all the code between the Procedure4 entry at the top of the list and the next statement to be executed in Procedure2.

The list in the Calls menu is sometimes called a stack because the procedure on top is the most recently invoked. The Calls menu can display eight procedures, but the number of interprocedure calls in your program is limited only by the amount of stack space available. When there are more than eight such calls, the Calls menu shows only the most recent eight. To see the ones further down in the stack you must execute back towards the bottom.

You can experiment with the characteristics of the call stack using the Fibonacci example at the end of Section 7.3.4.4, "Watch Expressions." Because the Fib procedure is recursive, the number you enter at the prompt determines the depth of the call stack and the number of

procedures in the Calls menu. After interrupting execution (using either a Watchpoint or by pressing CTRL+BREAK) you can display the Calls menu and use the methods described above to execute between different invocations of the Fib procedure.

7.4.2 Compatibility with the CodeView® Debugger

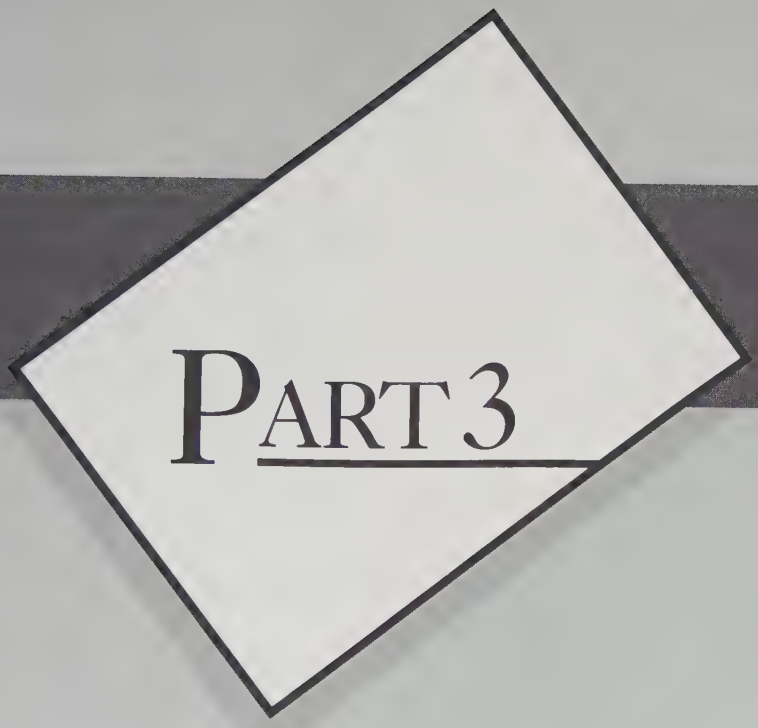
You can combine routines from other languages into your QuickBASIC programs by including them in Quick libraries. Although you cannot use QuickBASIC's debugging features to trace through procedures in a Quick library, you can use the Microsoft CodeView debugger to debug executable files you create with the BC command-line compiler. To create CodeView-compatible executable files, you must use the /zi option to the bc command and the /CO option to the link command.

The CodeView debugger is a very powerful tool. It is included with the Microsoft Macro Assembler (Version 5.0) and Microsoft C (Version 5.0). Using the CodeView debugger you can debug programs that combine routines from Microsoft BASIC, FORTRAN, Pascal, C, and Macro Assembler. The debugger shows the source code of each module—both in its original source form, and in its assembly-language form. See Chapter 8, "Quick Libraries," for more information on Quick libraries and Chapter 9, "Compiling and Linking from DOS," for more information on object files and command-line compiling and linking for CodeView compatibility.

See Appendix C, "Calling C and Assembly-Language Routines," for more information on how your BASIC programs can call routines written in C or assembly language.

Note

You cannot produce a CodeView-compatible executable file from within the QuickBASIC environment. The Produce Debug Code option in the Make EXE dialog box does not make an executable file CodeView compatible.



LIBRARIES AND TOOLS

CHAPTERS

8 Quick Libraries....187

9. Compiling and Linking from DOS....203

PART 3 ◇ LIBRARIES AND TOOLS

This portion of the manual explains advanced QuickBASIC features, such as libraries and separate compilation. A library is a special file that contains all the procedures from a module or group of modules. You'll learn how to create and use both Quick libraries and stand-alone libraries, and how to include routines written in other Microsoft languages in libraries. You'll also learn how to compile and link from the DOS command line and create executable files that are compatible with the Microsoft Codeview debugger.

CHAPTER 8

QUICK LIBRARIES

8.1 Types of Libraries 189

8.2 Advantages of Quick Libraries 190

8.3 Creating a Quick Library 191
- 8.3.1 Files Needed to Create a Quick Library 192
- 8.3.2 Making a Quick Library 192
- 8.3.3 Making a Quick Library from within the Environment (Make Library) 193

8.4 Using Quick Libraries 195
- 8.4.1 Loading a Quick Library 196
- 8.4.2 Floating-Point Arithmetic in Quick Libraries 197
- 8.4.3 Viewing the Contents of a Quick Library 197

8.5 The Supplied Library (QB.QLB) 197

8.6 The .QLB File-Name Extension 198

8.7 Making a Library from the Command Line 198

8.8 Using Routines from Other Languages in a Quick Library 199

8.9 Memory Considerations with Quick Libraries 200

8.10 Making Compact Executable Files 201

This chapter describes how to create and maintain libraries from within the QuickBASIC programming environment. A library is a file containing the contents of several modules under a single file name. When you finish this chapter you will know how to

- Make libraries from within the QuickBASIC environment.
- Load a Quick library when running a QuickBASIC program.
- View the contents of a Quick library.
- Add routines written in other languages to a Quick library.

8.1 Types of Libraries

QuickBASIC provides tools for creating two different types of libraries, which are identified by different file-name extensions:

Extension	Function
.QLB	The .QLB extension characterizes a Quick library, a special kind of library that permits easy addition to the BASIC language as recognized within the QuickBASIC programming environment. A Quick library can contain procedures written in QuickBASIC or other Microsoft languages such as Microsoft C.
.LIB	The .LIB extension characterizes a library, called a stand-alone (.LIB) library, that is created with the Microsoft Library Manager, LIB. When QuickBASIC makes a Quick library, it simultaneously creates a .LIB library containing the same procedures in a somewhat different form.

You can generate both types of libraries from within the programming environment or from the command line. You can think of a Quick library as a group of procedures that is "linked" to QuickBASIC when the library is loaded with QuickBASIC. Libraries with the .LIB extension are essentially compiled, but unlinked, procedures. They can either be added to a Quick library, or linked with a main module to create a file that is executable from the DOS command line.

This chapter discusses the use of command-line utilities for some common cases, but you should refer to Chapter 9, "Compiling and Linking from DOS," for a full explanation of using those utilities.

8.2 Advantages of Quick Libraries

Quick libraries facilitate program development and maintenance in many ways.

As development progresses on a project and modules become stable components of your program, you can add them to a Quick library, then set aside the source files for the original modules until you want to improve or maintain those source files. Thereafter you can load the library along with QuickBASIC, and your program has instant access to all procedures in the library.

Procedures in a Quick library behave like QuickBASIC's own statements. If properly declared, a **SUB** procedure in the Quick library can even be invoked without a **CALL** statement, making it look like a BASIC statement. See Chapter 2, "Procedures: Subprograms and Functions," in *Programming in BASIC: Selected Topics*, for more information on calling a **SUB** procedure with or without the **CALL** keyword.

Quick-library procedures can be executed directly from the Immediate window, just like BASIC statements. This means that you can test their effects before using them in other programs.

If you co-develop programs with others, Quick libraries make it easy to update a pool of common procedures. Similarly, if you wish to offer a library of original procedures for commercial distribution, all QuickBASIC programmers will be able to use them immediately to enhance their own work. You could leave your custom Quick library on a bulletin board for others to try before purchasing. Because Quick libraries contain no source code and can only be used within the QuickBASIC programming environment, your proprietary interests are protected while your marketing goals are advanced.

Note

Quick libraries have the same function as user libraries in QuickBASIC Versions 2.0 and 3.0. However, you cannot load a user library as a Quick library. You must recreate the library from the original source code, as described below.

8.3 Creating a Quick Library

A Quick library automatically contains all modules, both main and non-main, present in the QuickBASIC environment when you create the new library. It also contains the contents of any other Quick library which you loaded when starting QuickBASIC. If you load a whole program, but only want certain modules to be put in the library, you must explicitly unload those you don't want. You can unload modules with the File menu's Unload File command.

You can quickly determine which modules are loaded by checking the list box of the SUBs command on the View menu. However, this method does not show you which procedures a loaded library contains. The utility program QLBDUMP.BAS, described in Section 8.4.3, "Viewing the Contents of a Quick Library," allows you to list all the procedures in a library.

Only whole modules can be put into a library. That is, you cannot select one procedure from among many in a module. If you want to enter only certain procedures from a module, put the procedures you want in a separate module, then put that module into a library.

A Quick library must be self contained. That is, it must not refer to any procedures outside the library. Therefore, all procedures invoked by a procedure in the Quick library must also be placed in the Quick library.

With large programs, you can reduce loading time by putting as many routines as possible in Quick libraries. Putting many routines in Quick libraries is also an advantage if you plan to make the program into a stand-alone executable file later, since the contents of libraries are simply linked without recompiling.

Note

Your main module may or may not contain procedures. If it does, and you incorporate those procedures in the library, the entire main module goes in the library, too. This does not cause an error message, but the module-level code in the library can never be executed unless one of its procedures contains a routine (such as **ON ERROR**) that explicitly passes control to the module level. Even if that is the case, much of the module-level code may be extraneous. If you organize your procedures in modules that are frequently used together, your Quick libraries will likely be less cluttered with useless code.

8.3.1 Files Needed to Create a Quick Library

To create a Quick library, make sure that the proper files are available before you begin. If you don't have a hard disk, you should keep your files and the other programs on several floppy disks. QuickBASIC prompts you for a path name when it cannot find a file: when this happens, insert the correct disk and respond to the prompt.

Make sure that the following files are in the current working directory or accessible to QuickBASIC through the appropriate DOS environment variables:

File	Purpose
QB.EXE	Directs the process of creating a Quick library. If you are working only with QuickBASIC modules, you can do everything in one step from within the QuickBASIC environment.
BC.EXE	Creates object files from source code.
LINK.EXE	Links object files.
LIB.EXE	Manages stand-alone libraries of object modules.
BQLB40.LIB	Supplies routines needed by your Quick library. This library is a stand-alone library that is linked with objects in your library to form a Quick library.

8.3.2 Making a Quick Library

Most of the time you create Quick libraries from within the QuickBASIC environment. Occasionally, you may want to update a library or include routines from other Microsoft languages in the library. In these cases, you begin by constructing a base library of the non-BASIC routines from outside the environment by invoking LINK and LIB directly. Then you can add the most current QuickBASIC modules from within QuickBASIC.

8.3.3 Making a Quick Library from within the Environment (Make Library)

When making a library from within the QuickBASIC environment, the first consideration is whether the library to be made is totally new or an update of an existing library. If it is to be an update, you should start QuickBASIC with the /l command-line option, supplying the name of the library to be updated as a command-line argument. At the same time, you can also include the name of a program whose modules you want to put in the library. In this case QuickBASIC loads all the modules specified in that program's .MAK file.

If you load your program when starting QuickBASIC, be sure to unload any modules you don't want in the Quick library, including the main module (unless it contains procedures you want in the library).

Follow these steps to unload modules:

1. Choose the Unload File command from the File menu.
2. Select the module you want to unload from the list box, then press ENTER.
3. Repeat steps 1 and 2 until you have unloaded all unwanted modules.

Alternatively, you can simply start QuickBASIC, with or without a library specification, and load the modules you want one at a time from within the environment. In this case, you load each module using the Load File command from the File menu.

To load one module at a time with QuickBASIC:

1. Choose the File menu's Load File command.
2. Select the name of a module you want to load from the list box.
3. Repeat steps 1 and 2 until all you have loaded all the modules you want.

Once you have loaded the previous library (if any) and all the new modules you want to include in the Quick library, choose the Make Library command on the Run menu. The dialog box shown in Figure 8.1 appears.

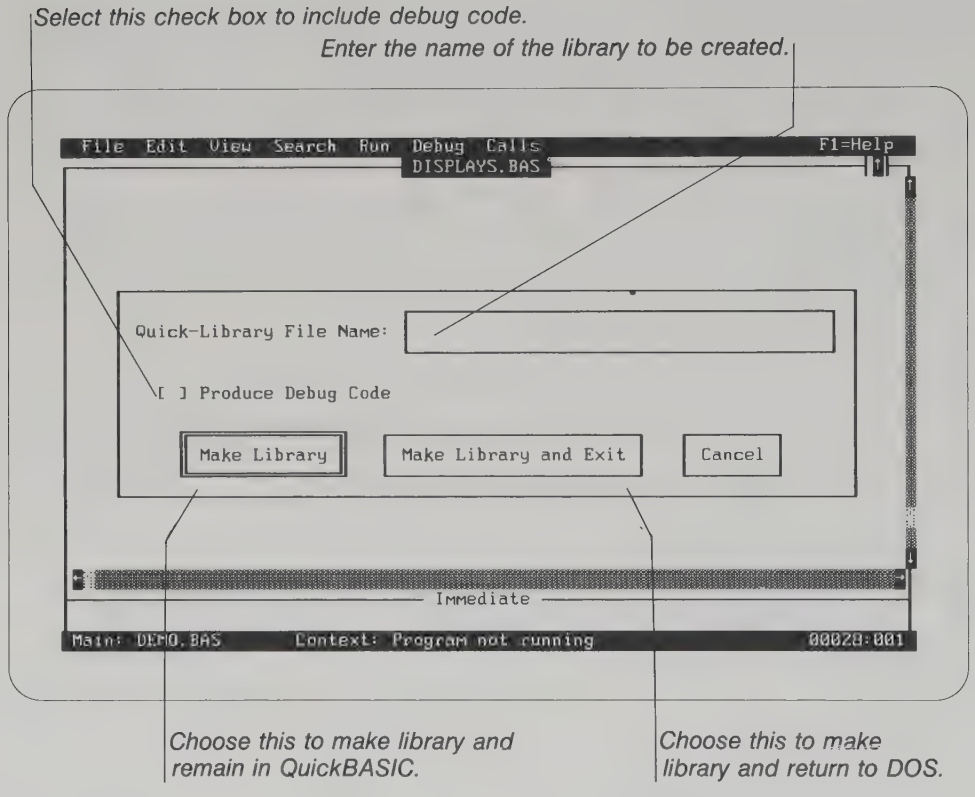


Figure 8.1 Make Library Dialog Box

To create a Quick library, perform the following steps:

1. Enter the name of the library you wish to create in the Quick-Library File Name text box.

   If you enter only a base name (that is, a file name with no extension), QuickBASIC automatically appends the extension .QLB when it creates the library. If you want your library to have no extension, add a terminating period (.) to the base name. Otherwise, you may enter any base name and any extension you like, consistent with DOS file-name rules.

2. Select the Produce Debug Code check box only if you are specifically trying to track a bug you believe to be in a library that you are updating. It makes your library larger, your program slower and gives only a small amount of error control, mostly dealing with checking of array bounds.

3. Create the Quick library:

   - Choose the Make Library command button if you want to remain in the environment after the Quick library is created.
   - Choose the Make Library and Exit command button if you want to return to the DOS command level after the Quick library is created.

Note

When you make a Quick library, be aware that if it is ever to be used with a nonlibrary module that needs to trap events such as keystrokes, then one of the modules in the library must contain at least one event-trapping statement. This statement can be as simple as **TIMER OFF**; but without it, events are not trapped correctly in the Quick library.

8.4 Using Quick Libraries

This section explains how to load a Quick library when you start QuickBASIC and how to view the contents of a Quick library. It also gives facts that you should remember when procedures within a Quick library perform floating-point arithmetic.

8.4.1 Loading a Quick Library

To load a Quick library, you must specify the name of the desired library on the command line when you start QuickBASIC using the following syntax:

qb [[*progname*]] **/l** [[*qlibname*]]

If you start QuickBASIC with the /l option and supply the name of a library (*qlibname* in this example), QuickBASIC loads the specified library and places you in the programming environment. The contents of the library are now available for use. If you start QuickBASIC with the /l option but don't specify a library, QuickBASIC loads the library QB.QLB (see Section 8.5).

You can also start QuickBASIC with the /run option followed by both a program name (*progname*) and the /l option. In this case, QuickBASIC loads both the program and the specified Quick library, then runs the program without stopping at the programming environment.

Note

When using Quick libraries to represent program modules, remember to update the .MAK file to keep it consistent with the modules in the evolving program. (This is done with the Unload File command from the File menu.) If the .MAK file is not up to date, it may cause QuickBASIC to load a module containing a procedure definition with the same name as one defined in the Quick library, which in turn causes the error message "Duplicate definition."

You can load only one Quick library at a time. QuickBASIC searches for the Quick library in the following three locations:

1. The place specified in the command-line argument, if any.
2. The current directory.
3. The path specified by the LIB environment variable. (See your DOS documentation for information about environment variables.)

■ Example

The following command starts QuickBASIC and runs the program named REPORT.BAS using the routines in the library FIGS.QLB:

```
qb /run report.bas /l figs.qlb
```

8.4.2 Floating-Point Arithmetic in Quick Libraries

BASIC procedures within Quick libraries represent code compiled with the BC command-line compiler. These procedures share significant characteristics with executable files. For example, both executable files and Quick libraries perform floating-point arithmetic faster and with a higher degree of accuracy than the same calculations performed within the QuickBASIC environment. See Section 6.2.4, "Floating-Point Arithmetic in Executable Files," for more information. See Chapter 3, "Expressions and Operators," in the *BASIC Language Reference*, for more information on using relational operators and expressions with floating-point values.

8.4.3 Viewing the Contents of a Quick Library

Because a Quick library is essentially a binary file, you cannot simply view its contents with a text editor to find out what it contains. Your distribution disk includes the QLBDUMP.BAS utility, which allows you to list all the procedures and data symbols in a given library. Follow these steps to view the contents of a Quick library:

1. Start QuickBASIC.
2. Load and run QLBDUMP.BAS.
3. Enter the name of the Quick library you wish to examine in response to the prompt. You do not need to include the .QLB extension when you type the file name; however, supplying the extension does no harm.

   If the specified file exists, and it is a Quick library, the program displays a list of all the symbol names in the library. In this context, symbol names correspond to the names of procedures in the library.

See Chapter 3, "File and Device I/O," in *Programming in BASIC: Selected Topics*, for a commented listing of QLBDUMP.BAS.

8.5 The Supplied Library (QB.QLB)

If you invoke QuickBASIC with the /l option, but do not supply a Quick-library name, QuickBASIC automatically loads a library named QB.QLB. This file is included on distribution disk 3. The file contains three routines, named INTERRUPT, INT86OLD, and ABSOLUTE, which provide

software-interrupt support for system-service calls, and support for **CALL** ABSOLUTE. To use the routines in QB.QLB, you must specify it (or another library into which those routines have been incorporated) on the command line when you invoke QuickBASIC. If you wish to use these routines along with other routines which you have placed in libraries, make a copy of the QB.QLB library and use it as a basis for building a library containing all the routines you need. See the entry for the **CALL** statement in the *BASIC Language Reference*, for a description of how to invoke these routines with the **CALL** statement.

8.6 The .QLB File-Name Extension

The extension .QLB is just a convenient convention. You can use any extension for your Quick library files, or no extension at all. However, in processing the /l *qlibname* option, QuickBASIC assumes that the listed *qlibname* has the .QLB extension if no other extension is specified. If your Quick library has no extension, you must specify it as *qlibname.*, or QuickBASIC searches for a file with your base name and the .QLB extension. A period (.) terminating the base name tells QuickBASIC that the file name has no extension.

8.7 Making a Library from the Command Line

After making a library from within the QuickBASIC environment, you will notice the appearance of extra files with the extensions .OBJ and .LIB. In creating Quick libraries, QuickBASIC actually directs the work of three other programs, BC, LINK and LIB, and then combines what they produce into both a Quick library and a stand-alone (.LIB) library. Once the process is complete, there is one object (.OBJ) file for each module in your Quick library, and a single library (.LIB) file containing an object module for each object file. The files with the extension .OBJ are now extraneous and can be deleted. However, files with the extension .LIB are very important and should be preserved. These parallel libraries are the files QuickBASIC uses to create executable files of your programs.

You can use the programs LINK and LIB to create both Quick libraries and stand-alone (.LIB) libraries from the command line in batch mode. If you want to use routines originally written and compiled in other languages in QuickBASIC, you *must* first put the other-language routines in a Quick library via the command line. Once the other-language routines are in the library, you can incorporate your BASIC modules from the command line or from within the QuickBASIC environment.

Professional software developers should be sure to deliver both the Quick and stand-alone (.LIB) versions of libraries to customers. Without the .LIB libraries, end users would not be able to use your library routines in executable files produced with QuickBASIC.

When you create a Quick library using the linker, the library BQLB40.LIB must always be specified after the third comma on the link command line or in response to the "Libraries" prompt.

8.8 Using Routines from Other Languages in a Quick Library

To place routines from other languages in a Quick library, you must start with precompiled or preassembled object files that contain the other-language routines you wish to use. Several other languages are suitable for this purpose, including Microsoft C, Microsoft Macro Assembler, Microsoft Pascal, Microsoft FORTRAN, and any other language that creates object files compatible with the Microsoft language family.

The following is a typical scenario for building a Quick library containing routines from other languages:

1. Suppose you begin with three modules, created in FORTRAN, C, and Macro Assembler. First you compile or assemble each module with the proper language translator to produce object files called here FOR.OBJ, C.OBJ, and ASM.OBJ.

2. You then link the object files with the LINK option, /Q, that instructs the linker to produce a Quick-library file, as shown in the following command line:

   ```
   LINK /Q FOR.OBJ C.OBJ ASM.OBJ, MIXED.QLB,,BQLB40.LIB;
   ```

   The linker interprets the entry that follows the names of the object files (in this case MIXED.QLB) as the target file name, the name by which the linked modules will be known. Thus, in this case, the Quick-library file is named MIXED.QLB.

3. Now create a parallel .LIB library, using the same object files you just used to make the Quick library. In this case the first name following the LIB command is the name of the target .LIB library:

   ```
   LIB MIXED.LIB+FOR.OBJ+C.OBJ+ASM.OBJ;
   ```

Important

It is easy to overlook step 3 when making a library that contains other-language routines, but this step is crucial if you hope to use the library to create a stand-alone executable file. Without these parallel stand-alone (.LIB) libraries, QuickBASIC cannot create an executable file containing their routines.

4. With the other-language routines now in a Quick library, and the original object files in a stand-alone library having the same base name, you can return to the QuickBASIC environment and build as many BASIC modules into the library as available memory permits.

See Chapter 9, "Compiling and Linking from DOS," for a complete description of the features of LINK and LIB. See Appendix C, "Calling C and Assembly-Language Routines," for information on writing other-language routines and BASIC procedures that are mutually callable.

Note

QuickBASIC provides a BASIC system-level function (**B_OnExit**). This function can be called from other-language routines to log a termination procedure that is called when a BASIC program is terminated or restarted while executing when a Quick library is present. See Section C.9 for a description of this routine.

8.9 Memory Considerations with Quick Libraries

Because a Quick library is essentially an executable file (although it cannot be invoked by itself from the DOS command line), it is quite large in comparison to the sum of the sizes of its source files. This puts an upper limit on the number of routines you can put in a Quick library. To determine how large your Quick library can be, add up the memory required for

DOS, QB.EXE, and your program's main module. An easy way to estimate these factors is to boot your machine, start QuickBASIC with your program, and enter this command in the Immediate window:

```
print fre (-1)
```

This command shows you the number of bytes of free memory, which indicates the maximum size for any Quick library associated with this program. In most cases the amount of memory required for a Quick library is about the same as the size of its disk file. One exception to this rule of thumb is a library with procedures that use a lot of strings, which may require somewhat more memory.

8.10 Making Compact Executable Files

As discussed above, when QuickBASIC creates a Quick library, it also creates a stand-alone (.LIB) library of object modules in which each object module corresponds to one of the modules in the Quick library. When you make an executable file, QuickBASIC searches the stand-alone .LIB library for object modules containing the procedures referenced in the program.

If an object module in the library does not contain procedures referenced in the program, it is not included in the executable file. However, a single module may contain many procedures, and if even one of them is referenced in the program all are included in the executable file. Therefore, even if a program uses only one of four procedures in a certain module, that entire module becomes part of the final executable file.

To make your executable files as compact as possible, you should maintain a "tight" library, that is, one in which each module contains only closely related procedures. If you have any doubts about what a library contains, list its contents with the QLBDUMP.BAS utility described in Section 8.4.3, "Viewing the Contents of a Quick Library."

CHAPTER 9

COMPILING AND LINKING FROM DOS

9.1 BC, LINK, and LIB 205

9.2 Advantages of Using BC 206

9.3 The Compiling and Linking Process 206

9.4 Compiling with the bc Command 207
- 9.4.1 Specifying File Names 208
  - 9.4.1.1 Uppercase and Lowercase Letters 208
  - 9.4.1.2 File-Name Extensions 209
  - 9.4.1.3 Path Names 209
- 9.4.2 Using bc Command Options 210

9.5 Linking 212
- 9.5.1 Defaults for the Linker 214
- 9.5.2 Specifying Files to the Linker 215
- 9.5.3 Specifying Libraries to the Linker 216
- 9.5.4 Memory Requirements for the Linker 217
- 9.5.5 Using Linker Options 218
  - 9.5.5.1 Viewing the Options List (/HE) 219
  - 9.5.5.2 Pausing during Linking (/PAU) 219
  - 9.5.5.3 Displaying Linker-Process Information (/I) 220
  - 9.5.5.4 Preventing Linker Prompting (/B) 220
  - 9.5.5.5 Creating Quick Libraries (/Q) 221
  - 9.5.5.6 Packing Executable Files (/E) 221
  - 9.5.5.7 Disabling Segment Packing (/NOP) 222
  - 9.5.5.8 Ignoring the Usual BASIC Libraries (/NOD) 222
  - 9.5.5.9 Setting Maximum Number of Segments (/SE) 222
  - 9.5.5.10 Creating a Map File (/M) 223

9.5.5.11 Including Line Numbers in a Map File (/LI) 225
9.5.5.12 Packing Contiguous Segments (/PAC) 225
9.5.5.13 Using the CodeView Debugger (/CO) 226
9.5.5.14 Distinguishing Case (/NOI) 226
9.5.6 Other link Command-Line Options 226
9.6 Managing Stand-Alone Libraries: LIB 227
9.6.1 Running LIB 228
9.6.2 Usual Responses for LIB 230
9.6.3 Command Symbols 230
9.6.4 Cross-Reference-Listing Files 233
9.6.5 Setting the Library-Page Size 233

This chapter explains why you may want to compile and link outside the QuickBASIC environment. When you finish this chapter you will understand how to

- Compile from the DOS command line with the bc command.
- Create executable files and link program object files with the link command.
- Create and maintain stand-alone (.LIB) libraries with the lib command.

9.1 BC, LINK, and LIB

In addition to QB.EXE, the Microsoft QuickBASIC package includes several programs that you may find useful in special circumstances:

Program	Function
BC.EXE	When you choose the Make EXE File or Make Library command from the Run menu, QuickBASIC invokes the BASIC command-line compiler (BC) to produce intermediate program files called object files that will be linked together to form your program or Quick library. BC is also available to you any time you want to compile programs outside of the QuickBASIC environment. You may prefer to use BC if you simply want to compile programs you have written with another text editor. However, you only *need* to use BC from the command line if your program is too large to compile in memory within the QuickBASIC environment, or if you want your executable files to be compatible with the CodeView debugger.
LINK.EXE	QuickBASIC uses the Microsoft Overlay Linker (LINK) to link object files produced by BC with the appropriate libraries to produce an executable file. Like BC, you can use LINK whenever you want to link object files or make Quick libraries.
LIB.EXE	You can use the Microsoft Library Manager (LIB) to create stand-alone libraries from the object files produced by BC. QuickBASIC itself uses LIB to create such libraries and then uses them when you choose the Make EXE File command from the Run menu.

9.2 Advantages of Using BC

The following are some reasons why you might want to use BC to compile and link outside of the QuickBASIC environment:

- To use a different text editor
- To create executable programs that can be debugged with the Microsoft CodeView debugger
- To create listing files for use in debugging a stand-alone executable program
- To use options not available within the QuickBASIC environment, such as storing arrays in row order
- To link with NOCOM.OBJ or NOEM.OBJ (files supplied with QuickBASIC), which reduce the size of executable files in programs that do not use the **COM** statement or are always used with a math coprocessor

9.3 The Compiling and Linking Process

To create a stand-alone program from a BASIC source file outside of the QuickBASIC environment, follow these steps:

1. Compile each source file, creating an object file.
2. Link the object files with one or more stand-alone libraries in order to create an executable file. The linker makes sure that all the procedure calls in the source files match up with the procedures in the libraries or with procedures in other object files before it creates an executable file.

You can use either of the two following methods of compiling and linking:

1. You can compile and link in separate steps by using the bc and link commands.
2. You can create a batch file containing all the compiling and linking commands. This method is most useful if you use the same options whenever you compile and link your programs.

Note

When QuickBASIC compiles and links your program from within the environment, the /E linker option is set automatically. However, when you use the link command outside the QuickBASIC environment, you must explicitly specify the /E option in order to minimize the size of the executable file and maximize program-loading speed.

Sections 9.4 and 9.5 explain how to compile and link in separate steps.

9.4 Compiling with the bc Command

You can compile with the bc command in either of the following two ways:

1. Type all information on a single command line, using the following syntax:

   bc *sourcefile* [,[*objectfile*] [,[*listingfile*]]] [*optionslist*][;]

2. Type

   ```
   bc
   ```

   and respond to the following prompts:

   ```
   Source Filename [.BAS]:
   Object Filename [basename.OBJ]:
   Source Listing: [NUL.LST]:
   ```

Table 9.1 shows the input you must give on the bc command line or in response to each prompt:

Table 9.1

Input to the bc Command

Field	Prompt	Input
sourcefile	"Source Filename"	The name of your source file
objectfile	"Object Filename"	The name of the object file you are creating
listingfile	"Source Listing"	The name of the file containing a source listing. The source-listing file contains the address of each line in your source file, the text of the source file, its size, and any error messages produced during compilation.
optionslist	Gives options after any response	Any of the compiler options described in Section 9.4.2, "Using bc Command Options"

9.4.1 Specifying File Names

The bc command makes certain assumptions about the files you specify, based on the path names and extensions you use for the files. The following sections describe these assumptions and other rules you should follow in specifying file names to the bc command.

9.4.1.1 Uppercase and Lowercase Letters

You can use any combination of uppercase and lowercase letters for file names.

■ **Example**

The bc command considers the following three file names to be equivalent:

```
abcde.BAS
ABCDE.BAS
aBcDe.Bas
```

9.4.1.2 File-Name Extensions

A DOS file name has two parts: the "base name," which includes everything up to (but not including) the period (.), and the "extension," which includes the period and up to three characters following the period. In general, the extension identifies the type of file (for example, whether the file is a BASIC source file, an object file, an executable file, or a stand-alone library).

BC and LINK use the file-name extensions explained in the following list:

Extension	Explanation
.BAS	BASIC source file
.OBJ	Object file
.LIB	Stand-alone library file
.LST	Listing file produced by BC
.MAP	File of symbols from the linked program
.EXE	Executable file

9.4.1.3 Path Names

Any file name can include a full or partial path name. A full path name starts with the drive name; a partial path name has one or more directory names preceding the file name, but does not include a drive name.

Giving a full path name allows you to specify files in different paths as input to the bc command and allows you to create files on different drives or in different directories on the current drive.

Note

For files that you are creating with BC you can give a path name ending in a backslash (\) to create the file in that path. When it creates the file, BC uses the default name for the file.

9.4.2 Using bc Command Options

Options to the bc command consist of either a forward-slash character (/) or a dash (-) followed by one or more letters. (In this manual, forward slashes are used for options.)

The bc command-line options are explained in the following list:

Option	Description
/a	Creates a listing of the disassembled object code for each source line and shows the assembly-language code generated by the compiler.
/ah	Allows dynamic arrays of records, fixed-length strings, and numeric data to occupy all of available memory. If this option is not specified, the maximum size is 64K per array. Note that this option has no effect on the way data items are passed to procedures.
/c:*buffersize*	Sets the size of the buffer receiving remote data using an asynchronous communications adapter for each communications port. (The transmission buffer is allocated 128 bytes for each communications port and cannot be changed on the bc command line.) This option has no effect if the asynchronous communications card is not present. The default buffer size is 256 bytes total for both ports; the maximum size is 32,767 bytes.
/d	Generates debugging code for run-time error checking and enables CTRL+BREAK. This option is the same as the Produce Debug Code option from the Run menu's Make EXE File command within the environment.
/e	Indicates the presence of **ON ERROR** with **RESUME** *linenumber* statements. (See also the discussion of the /x option in this list.)
/mbf	Causes the QuickBASIC conversion functions to treat IEEE-format numbers as Microsoft Binary format numbers. The intrinsic functions **MKS$**, **MKD$**, **CVS**, and **CVD** are converted to **MKSMBF$**, **MKDMBF$**, **CVSMBF**, and **CVDMBF**, respectively. See the *BASIC Language Reference* for more information about these functions.

/o	Substitutes the BCOM40.LIB run-time library for BRUN40.LIB. See Section 6.2, "Creating Executable Files from within QuickBASIC," for more information about using these libraries.
/r	Stores arrays in row-major order. BASIC normally stores arrays in column-major order. This option is useful if you are using other language routines that store arrays in row order.
/s	Writes quoted strings to the object file instead of the symbol table. Use this option when an "Out of memory" error message occurs in a program that has many string constants.
/v	Enables event trapping for communications (**COM**), lightpen (**PEN**), joystick (**STRIG**), timer (**TIMER**), music buffer (**PLAY**) and function keys (**KEY**). Use this option to check between statements for an occurrence of an event.
/w	Enables event trapping for the same statements as /v, but checks between lines for occurrence of an event.
/x	Indicates presence of **ON ERROR** with **RESUME**, **RESUME NEXT**, or **RESUME 0**.
/zd	Produces an object file containing line-number records corresponding to the line numbers of the source file. This option is useful when you want to perform source-level debugging using the Microsoft Symbolic Debug Utility (SYMDEB), available with the Microsoft Macro Assembler, Version 4.0.
/zi	Produces an object file containing debugging information used by the Microsoft CodeView debugger, available with Microsoft C, Version 5.0 and Microsoft Macro Assembler, Version 5.0.

9.5 Linking

After compiling your program, you must link the object file with the appropriate libraries to create an executable program. You can use the link command in any of the following ways:

- Give the input on a command line of the following form:

  link *objfile* [,[*exefile*][,[*mapfile*][,[*lib*]]][*linkopts*][;]]

  The command line cannot be longer than 128 characters.

- Type

  ```
  link
  ```

  and respond to the following prompts:

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

  To give more files for any prompt, type a plus sign (+) at the end of the line. The prompt reappears on the next line, and you can continue typing input for the prompt.

- Set up a file (called a "response file") containing responses to the link command prompts, and then type a link command of the following form:

  link @ *filename*

  Here, *filename* is the name of the response file. You can append linker options to any response or give options on one or more separate lines. The responses must be in the same order as the link command prompts discussed above. You can also enter the name of a response file after any linker prompt, or at any position in the link command line.

Table 9.2 shows the input you must give on the link command line, or in response to each prompt.

Table 9.2
Input to the link Command

Field	Prompt	Input
objfile	"Object Modules"	One or more object files that you are linking. The object files are separated by plus signs or spaces.
exefile	"Run File"	Name of the executable file you are creating, if you want to give it a name or extension other than the default. You should always use the .EXE extension, since DOS expects executable files to have this extension.
mapfile	"List File"	Name of the file containing a symbol map listing, if you are creating one.* You can also specify one of the following DOS device names to direct the map file to that device: AUX for an auxiliary device, CON for the console (terminal), PRN for a printer device, or NUL for no device (so that no map file is created). See Section 9.5.5.10 for a sample map file and information about its contents.
lib	"Libraries"	One or more stand-alone libraries (or directories to be searched for stand-alone libraries) separated by plus signs or spaces. The "Libraries" prompt allows you to specify up to 16 libraries; any additional libraries are ignored. See Section 9.5.3 for rules for specifying library names to the linker.
linkopts	Gives options after any response	Any of the linker options described in Sections 9.5.5.1–9.5.5.11. You can specify linker options anywhere on the command line.

*Another way to create a map file is to specify the /MAP option (see Section 9.5.5.10) to the link command.

If you are using a response file, each response must follow the rules outlined in Table 9.2.

9.5.1 Defaults for the Linker

You can choose defaults for any of the information that the linker needs in either of the following three ways:

1. To choose the default for any command-line entry, omit the file name or names before the entry and type only the required comma. The only exception to this is the default for the *mapfile* entry: if you use a comma as a placeholder for this entry, the linker creates a map file.

2. To choose the default for any prompt, simply press ENTER.

3. To choose the defaults for all remaining command-line entries or prompts, type a semicolon after any entry or prompt. The only required input is one or more object-file names.

The following list shows the defaults that the linker uses for executable files, map files, and libraries:

File Type	Default
Executable	Base name of the first object file given, plus the .EXE extension. To rename the executable file, you are required to specify only the new base name; if you give a file name with no extension, the linker automatically appends the .EXE extension.
Map	The special file name NUL.MAP, which tells the linker *not* to create a map file. To create a map file, you are required to specify only the base name; if you give a file name with no extension, the linker automatically appends the .MAP extension.
Libraries	Libraries named in the given object files. The default library is BRUN40.LIB. If you specify a library other than the default library, you are required to give only the base name; if you give a library name with no extension, the linker automatically appends the .LIB extension. See Section 9.5.3 for information about specifying libraries other than the default libraries.

Note

When linking a stand-alone executable file, if your program does not use the **COM** statement, your program will be about 4K smaller if you link with NOCOM.OBJ, a file supplied with QuickBASIC.

■ Examples

The following example shows a response file. It tells the linker to link together the four object modules FRAME, TEXT, TABLE, and LINEOUT. The executable file named FRAME.EXE, as well as the map file named FRAMESYM.MAP, are produced. The /PAUSE option causes the linker to pause before producing the executable file. This permits disk swapping, if necessary. The /MAP option tells the linker to include public symbols and addresses in the map file. The linker also links any needed routines from the library file, GRAF.LIB. See Sections 9.5.5.2 and 9.5.5.10 for more information on the /PAUSE and /MAP options.

```
FRAME TEXT TABLE LINEOUT
/PAUSE /MAP
FRAMESYM
GRAF.LIB
```

In the following example, the linker loads and links the object files FRAME.OBJ, TEXT.OBJ, TABLE.OBJ, and LINEOUT.OBJ, searching for unresolved references in the library file COBLIB.LIB. By default, the executable file is named FRAME.EXE. A map file that is called FRAMESYM.MAP is also produced.

```
LINK FRAME+TEXT+TABLE+LINEOUT, ,FRAMESYM, COBLIB.LIB
```

The example that follows illustrates how to continue any prompt by typing a plus sign (+) at the end of your response. The example links all of the given object files, then creates an executable file. Since a semicolon is typed as a response to the "Run File" prompt, the executable file is given the default name: the base name of the first object file given (FRAME), plus the .EXE extension. The defaults are also used for the remaining prompts; as a result, no map file is created, and the default libraries named in the object files are used for linking.

```
LINK

Object Modules [.OBJ]: FRAME TEXT TABLE LINEOUT+
Object Modules [.OBJ]: BASELINE REVERSE COLNUM+
Object Modules [.OBJ]: ROWNUM
Run File [FRAME.EXE]: ;
```

9.5.2 Specifying Files to the Linker

The rules for specifying file names to the linker are the same as the rules for specifying file names to the bc command: uppercase and lowercase letters can be used interchangeably, and file names can include path names to tell the linker to look for files or create files in the given path. See Section 9.4.1 for more information.

9.5.3 Specifying Libraries to the Linker

Ordinarily, you do not need to give the linker a stand-alone-library name. When the bc command creates object files, it places in each object file the name of the correct stand-alone library for that object file. When the object file is passed to the linker, the linker looks for a library with the same name as the name in the object file and links the object file with that library automatically.

To link object files with a stand-alone library other than the default, give the name of the nondefault library to the linker. You can give the library name in any of the following ways:

- After the third comma on the link command line. Commas follow the list of object-file names, the executable-file name, and the listing-file name. The final name is the library name.
- In response to the "Libraries" prompt of the link command.

The linker searches libraries you specify to resolve external references before it searches default libraries.

You might want to link with a stand-alone library other than the default to do any of the following:

- Link with additional stand-alone libraries.
- Link with libraries in different paths. If you specify a complete path name for the library, the linker only looks in that path for the library. Otherwise, it looks in the following three locations:
  1. The current working directory
  2. Any paths or drives you specify after the third comma on the command line
  3. The locations given by the LIB environment variable
- Ignore the library named in the object file. In this case, you must give the linker option /NOD in addition to specifying the library you want to use for linking. See Section 9.5.5.8 for more information about the /NOD option.

9.5.4 Memory Requirements for the Linker

The linker uses available memory for the linking session. If the files to be linked create an output file that exceeds available memory, the linker creates a temporary disk file to serve as memory. This temporary file is handled in one of the following ways, depending on the DOS version:

- The linker uses the directory specified by the DOS TMP environment variable, for the purpose of creating a temporary file. For example, if the TMP variable was set to C:\TEMPDIR, then the linker would put the temporary file in C:\TEMPDIR.

  If there is no TMP environment variable, or if the directory specified by TMP does not exist, then the linker puts the temporary file in the current working directory.

- If the linker is running on DOS Version 3.0 or later, it uses a DOS system call to create a temporary file with a unique name in the temporary-file directory.

- If the linker is running on a version of DOS prior to 3.0, it creates a temporary file named VM.TMP.

When the linker creates a temporary disk file, you see the message

```
Temporary file tempfile has been created.
Do not change diskette in drive, letter
```

where *tempfile* is ".\" followed by either the file name VM.TMP or a name generated by DOS, and *letter* is the drive containing the temporary file.

The message

```
Do not change diskette in drive
```

does not appear unless the drive named *letter* is a floppy-disk drive. If this message appears, do not remove the disk from the drive until the linking session ends. If you remove the disk, linker operations will be unpredictable, and you may see the following message:

```
unexpected end-of-file on scratch file
```

If you see this message, rerun the linking session.

The temporary file that the linker creates is a working file only. The linker deletes it at the end of the session.

Note

Do not give any of your own files the name VM.TMP. The linker displays an error message if it finds an existing file with this name.

9.5.5 Using Linker Options

All linker options begin with the linker's option character, the forward slash (/). Case is not significant in linker options; for example, /NOI and /noi are equivalent.

You can abbreviate linker options to save space and effort. The minimum valid abbreviation for each option is indicated in the syntax of the option. For example, several options begin with the letters "NO"; therefore, abbreviations for those options must be longer than "NO" to be unique. You cannot use "NO" as an abbreviation for the /NOIGNORECASE option, since the linker cannot tell which of the options beginning with "NO" you intend. The shortest valid abbreviation for this option is /NOI.

Abbreviations must begin with the first letter of the option and must be continuous through the last letter typed. No gaps or transpositions are allowed.

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

- A decimal number from 0 to 65,535.
- An octal number from 0 to 0177777. A number is interpreted as octal if it starts with a zero (0). For example, the number `10` is a decimal number, but the number `O10` is an octal number, equivalent to 8 in decimal.
- A hexadecimal number from 0 to 0xFFFF. A number is interpreted as hexadecimal if it starts with a zero followed by an x or an X. For example, `0x10` is a hexadecimal number, equivalent to 16 in decimal.

Linker options affect all files in the linking process, regardless of where the options are specified.

If you usually use the same set of linker options when you link, you can use the LINK environment variable to specify certain options each time you link. If you set this variable, the linker checks it for options and expects to find options listed exactly as you would type them on the command line. You cannot specify file-name arguments in the LINK environment variable.

Note

A command-line option overrides the effect of any environment-variable option with which it conflicts. For example, the command-line option /SE:256 cancels the effect of the environment-variable option /SE:512.

To prevent an option in the environment variable from being used, you must reset the environment variable itself.

■ Example

In the following example, the file TEST.OBJ is linked with the options /SE:256 and /CO. After that, the file PROG.OBJ is linked with the option /NOD, as well as with the options /SE:256 and /CO.

```
SET LINK=/SE:256 /CO
LINK TEST;
LINK /NOD PROG;
```

9.5.5.1 Viewing the Options List (/HE)

■ Option

/HE[LP]

The /HE option tells the linker to display a list of the available linker options on the screen.

9.5.5.2 Pausing during Linking (/PAU)

■ Option

/PAU[SE]

The /PAU option tells the linker to pause in the link session and display a message before it writes the executable file to disk. This allows you to insert a new disk to hold the executable file.

If you specify the /PAUSE option, the linker displays the following message before it creates the executable file:

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

The *letter* corresponds to the current drive. The linker resumes processing when you press the ENTER key.

Note

Do not remove the disk on which the list file is created or the disk used for the temporary file.

If a temporary file is created on the disk you plan to swap, you should press CTRL+C to terminate the linking session. Rearrange your files so that the temporary file and the executable file can be written to the same disk. Then try linking again.

9.5.5.3 Displaying Linker-Process Information (/I)

■ **Option**

/I[NFORMATION]

The /I option displays information about the linking process, including the phase of linking and the names of the object files being linked.

This option helps you determine the locations of the object files being linked and the order in which they are linked.

9.5.5.4 Preventing Linker Prompting (/B)

■ **Option**

/B[ATCH]

The /B option tells the linker not to prompt you for a new path name whenever it cannot find a library or object file that it needs. When this option is used, the linker simply continues to execute without using the file in question.

This option can cause unresolved external references. It is intended primarily for users who use batch or MAKE files to link many executable files with a single command and who do not want the linker to stop processing if it cannot find a required file. It is also useful when you are redirecting the link command line to create a file of linker output for future reference. However, this option does not prevent the linker from prompting for arguments missing from the link command line.

9.5.5.5 Creating Quick Libraries (/Q)

■ **Option**

/Q[UICKLIB]

The /Q option tells the linker to combine the object files you specify into a Quick library. When you start the QuickBASIC environment, you can give the /l option on the qb command line to load the Quick library. If you use the /Q option, be sure to specify BQLB40.LIB in the library list in order to include QuickBASIC Quick-library support routines.

See Chapter 8, "Quick Libraries," for more information about creating and loading Quick libraries.

Note

You cannot use the /EXEPACK option with the /Q option.

9.5.5.6 Packing Executable Files (/E)

■ **Option**

/E[XEPACK]

The /E option removes sequences of repeated bytes (typically null characters) and optimizes 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 reference changes when the executable image is loaded into memory and an actual address for the entry point is assigned.

Executable files linked with this option may be smaller and load faster than files linked without this option.

9.5.5.7 Disabling Segment Packing (/NOP)

■ **Option**

/NOP[ACKCODE]

The /NOP option is normally not necessary because code-segment packing is normally turned off. However, if an environment variable such as LINK turns on code-segment packing automatically, you can use the /NOP option to turn segment packing back off again.

9.5.5.8 Ignoring the Usual BASIC Libraries (/NOD)

■ **Option**

/NOD[EFAULTLIBRARYSEARCH]

When it creates an object file, BC includes the names of the "standard" libraries—libraries that the linker searches to resolve external references. The /NOD option tells the linker *not* to search any library specified in an object file to resolve external references.

In general, QuickBASIC programs do not work correctly without the standard QuickBASIC libraries (BRUN40.LIB and BCOM40.LIB). Thus, if you use the /NOD option, you should explicitly give the path name of the required standard library.

9.5.5.9 Setting Maximum Number of Segments (/SE)

■ **Option**

/SE[GMENTS]:*number*

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

For each segment, the linker must allocate space to keep track of segment information. When you set the segment limit higher than 128, the linker allocates more space for segment information. For programs with fewer than 128 segments, you can minimize the amount of storage the linker

needs by setting *number* to reflect the actual number of segments in the program. The linker displays an error message if this number is too high for the amount of memory the linker has available.

9.5.5.10 Creating a Map File (/M)

■ **Option**

/M[AP][:*number***]**

The /M option creates a map file. A map file lists the segments of a program in their order of appearance in the load module. The optional *number* field specifies the maximum number of public symbols that the linker can sort. By default, this limit is 2048. If the number of symbols exceeds this limit, then the linker generates only an unsorted list. When you specify a value for *number*, then *mapfile* contains a list of symbols sorted by address (assuming that *number* is large enough); however, it does not contain a list sorted by name. An sample map file is shown below:

```
 Start  Stop   Length Name                    Class
 00000H 01E9FH 01EA0H _TEXT                   CODE
 01EA0H 01EA0H 00000H C_ETEXT                 ENDCODE
.
.
.
```

The information in the columns labeled Start and Stop 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 column labeled Length gives the length of the segment in bytes. The column labeled Name gives the name of the segment, and the column labeled Class gives information about the segment type. See the *Microsoft MS-DOS Programmer's Reference* for information about groups, segments, and classes.

The starting address and name of each group appear after the list of segments. A sample group listing is shown below:

```
Origin   Group
01EA:0   DGROUP
```

In this example, DGROUP is the name of the data group.

The map file shown below contains two lists of global symbols: the first list is sorted in ASCII-character order by symbol name; the second, 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).

Many of the global symbols that appear in the map file are symbols used internally by the compiler and linker. These symbols usually begin with the characters B$ or end with QQ.

```
 Address          Publics by Name

 01EA:0096        STKHQQ
 0000:1D86        B$Shell
 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
.
.
.
```

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

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
```

A map file can also be specified by giving a map-file name on the link command line, or by giving a map-file name in response to the "List File" prompt.

9.5.5.11 Including Line Numbers in a Map File (/LI)

■ **Option**

/LI[NENUMBERS]

The /LI option creates a map file and includes the line numbers and associated addresses of the source program. If you are compiling and linking in separate steps, this option has an effect only if you are linking object files compiled with the /M option.

9.5.5.12 Packing Contiguous Segments (/PAC)

■ **Option**

/[NO]PAC[KCODE][:*number*]

The /PAC option tells the linker to group neighboring code segments. Code segments in the same group share the same segment address; all offset addresses are then adjusted upward as needed. As a result, many instructions that would otherwise have different segment addresses share the same segment address.

If specified, *number* is the size limit of groups formed by /PAC. The linker stops adding segments to a particular group as soon as it cannot add a segment to the group without exceeding *number*. At that point, the linker starts forming a new group with the remaining code segments. If *number* is not given, the default is 65,530.

Although the linker does not pack neighboring segments unless you explicitly ask for it, you can use the /NOPACKCODE option to turn off segment packing if, for example, you have given the /PAC option in the LINK environment variable.

9.5.5.13 Using the CodeView Debugger (/CO)

■ **Option**

/CO[DEVIEW]

The /CO option prepares an executable file for debugging using the CodeView debugger. If you are compiling and linking in separate steps, this option has an effect only if you are linking object files compiled with the /zi option of the bc command. Similarly, it should not be used in conjunction with the link command's /Q option, because a Quick library cannot be debugged with the CodeView debugger.

9.5.5.14 Distinguishing Case (/NOI)

■ **Option**

/NOI[GNORECASE]

The /NOI option tells the linker to distinguish between uppercase and lowercase letters; for example, the linker would consider the names ABC, abc, and Abc to be three separate names. When you link, do *not* specify the /NOI option on the link command line.

9.5.6 Other link Command-Line Options

Not all options of the link command are suitable for use with QuickBASIC programs. The following linker options can be used with Microsoft QuickBASIC programs; however, they are never required, since they request actions that the bc command or QuickBASIC performs automatically:

/CP[ARMAXALLOC]

Sets the maximum number of 16-byte paragraphs needed by the program when it is loaded into memory to *number*, an integer between 1 and 65,535, inclusive. The operating system uses this value when allocating space for the program before loading it. Although you can use this option on the link command line, it has no effect because, while it is running, your BASIC program controls memory.

/DO[SSEG]

Forces segments to be ordered using the defaults for Microsoft high-level language products. QuickBASIC programs always use this segment order by default.

/ST[ACK]:*number*

Specifies the size of the stack for your program, where *number* is any positive value (decimal, octal, or hexadecimal) up to 65,535 (decimal) representing the size, in bytes, of the stack. The standard BASIC library sets the default stack size to 2K.

Do not use the following linker options when linking object files compiled with BC. They are suitable only for object files created by the Microsoft Macro Assembler (MASM).

/DS[ALLOCATE]

Loads all data starting at the high end of the default data segment

HI[GH]

Places the executable file as high in memory as possible

/NOG[ROUPASSOCIATION]

Tells the linker to ignore group associations when assigning addresses to data and code items

/O[VERLAYINTERRUPT]:*number*

Specifies an interrupt number other than 0x3F for passing control to overlays

9.6 Managing Stand-Alone Libraries: LIB

The Microsoft Library Manager (LIB) manages the contents of stand-alone libraries.

A stand-alone library is made up of "object modules"—that is, object files that have been combined to form a library. Unlike an object file, an object module does not exist independently of the library it belongs to, and it does not have a path name or extension associated with its file name. Using LIB, you can

- Combine object files to create a new library.
- Add object files to an existing library.
- Delete or replace the object modules of an existing library.

- Extract object modules from an existing library and place them in separate object files.
- Combine the contents of two existing libraries into a new library.

When updating an existing library, LIB performs all of its operations on a copy of the library. This mechanism ensures that you have a backup copy of any library you update, in case of problems with the updated version of the library.

9.6.1 Running LIB

You can give the LIB command input in any of the following ways:

- Specify the input on a command line of the following form:

  LIB *oldlib* [/P[AGESIZE]:*number*] [*commands*][,[*listfile*][,[*newlib*]]][;]

  The command line cannot be longer than 128 characters.

- Type

  ```
  lib
  ```

  and respond to the following prompts:

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

  To give more files for any prompt, type an ampersand (&) at the end of the line. The prompt reappears on the next line, and you can continue typing input for the prompt.

- Set up a file with responses to LIB command prompts, known as a "response file," then type a LIB command of the following form:

  LIB @*filename*

  Here, *filename* is the name of the response file. The responses must be in the same order as the LIB prompts discussed above. You can also enter the name of a response file after any linker prompt, or at any position in the LIB command line.

Table 9.3 shows the input you must give on the LIB command line, or in response to each prompt.

Table 9.3
Input to the LIB Command

Field	Prompt	Input
oldlib	"Library name"	Name of the library you are changing or creating. If this library does not exist, LIB asks if you want to create it. Type the letter "y" to create a new library or the letter "n" to terminate LIB. This message is suppressed if you type command characters, a comma, or a semicolon after the library name. A semicolon tells LIB to perform a consistency check on the library; in this case, it displays a message if it finds errors in any library module.
/P:*number*	/P:*number* after "Library name" prompt	The library-page size. This sets the page size for the library to *number*, where *number* is an integer power of 2 between 16 and 32,768, inclusive. The default page size for a new library is 16 bytes. Modules in the library are always aligned to start at a position that is a multiple of the page size (in bytes) from the beginning of the file.
commands	"Operations"	Command symbols and object files that tell LIB what changes to make in the library.
listfile	"List file"	Name of a cross-reference-listing file. No listing file is created if you do not give a file name.
newlib	"Output library"	Name of the changed library that LIB creates as output. If you do not give a new library name, the original, unchanged library is saved in a library file with the same name but with a .BAK extension replacing the .LIB extension.

If you are using a response file, each response must follow the rules outlined in Table 9.3.

9.6.2 Usual Responses for LIB

LIB has its own built-in responses, sometimes called default responses. You can choose these usual responses for any of the information that LIB needs, in any the following ways:

- To choose the default for any command-line entry, omit the file name or names before the entry and type only the required comma. The only exception to this is the default for the *listfile* entry: if you use a comma as a placeholder for this entry, LIB creates a cross-reference-listing file.
- To choose the default for any prompt, simply press ENTER.
- To choose the defaults for all command-line entries or prompts that follow an entry or prompt, type a semicolon (;) after that entry or prompt. The semicolon should be the last character on the command line.

The following list shows the defaults that LIB uses for cross-reference-listing files and output libraries:

File	Default
Cross-reference listing	The special file name NUL, which tells the linker *not* to create a cross-reference-listing file
Output library	The *oldlib* entry or the response to the "Library name" prompt

9.6.3 Command Symbols

To tell LIB what changes you want to make to a library, type a command symbol such as +, −, −+, *, or −*, followed immediately by a module name, object-file name, or library name. You can specify more than one operation, in any order.

The following list shows each LIB command symbol, the type of file name to specify with the symbol, and what the symbol does:

Command	Meaning
+{*objfile* | *lib*}	Adds the given object file to the input library and makes that object file the last module in the library, if given with an object-file name. You can use a path name for the object file name. Since LIB automatically supplies the .OBJ extension, you can omit the extension from the object-file name. If given with a library name, the plus sign (+) adds the contents of that library to the input library. The library name must have the .LIB extension.
–*module*	Deletes the given module from the input library. A module name does not have a path name or an extension.
–+*module*	Replaces the given module in the input library. Module names have no path names and no extensions. LIB deletes the given module, then appends the object file that has the same name as the module. The object file is assumed to have an .OBJ extension and to reside in the current working directory.
**module*	Copies the given module from the library to an object file in the current working directory. The module remains in the library file. When LIB copies the module to an object file, it adds the .OBJ extension. You cannot override the .OBJ extension, drive designation, or path name given to the object file. However, you can later rename the file or copy it to whatever location you like.
–**module*	Moves the given object module from the library to an object file. This operation is equivalent to copying the module to an object file, as described above, then deleting the module from the library.

■ Examples

The example below uses the replace command symbol (–+) to instruct LIB to replace the HEAP module in the library LANG.LIB. The LIB program deletes the HEAP module from the library, then appends the object file HEAP.OBJ as a new module in the library. The semicolon at the end of the command line tells LIB to use the default responses for the remaining prompts. This means that no listing file is created and that the changes are written to the original library file instead of creating a new library file.

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

The examples below perform the same function as the first example in this section, but in two separate operations, using the add (+) and delete (–) command symbols. The effect is the same for these examples because delete operations are always carried out before add operations, regardless of the order of the operations in the command line. This order of execution prevents confusion when a new version of a module replaces an old version in the library file.

```
LIB LANG-HEAP+HEAP;

LIB LANG+HEAP-HEAP;
```

The example below causes LIB to perform a consistency check of the library file FOR.LIB. No other action is performed. LIB displays any consistency errors it finds and returns to the operating-system level.

```
LIB FOR;
```

The following example tells LIB to perform a consistency check on the library file LANG.LIB and then create a cross-reference-listing file named LCROSS.PUB.

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

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

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

The contents of the response file below cause LIB to delete the module HEAP from the LIBFOR.LIB library file, extract the module FOIBLES and place it in an object file named FOIBLES.OBJ, and append the object files CURSOR.OBJ and HEAP.OBJ as the last two modules in the library. Finally, LIB creates a cross-reference-listing file named CROSSLST.

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

9.6.4 Cross-Reference-Listing Files

A cross-reference-listing file helps you keep track of which routines are contained in a stand-alone library and the original object files from which they came. A cross-reference-listing file contains the following lists:

- 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 referenced.
- A list of the modules in the library. Under each module name is an alphabetical listing of the public symbols defined in that module.

9.6.5 Setting the Library-Page Size

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

You can set a different library-page size while you are creating a library or change the page size of an existing library by adding the following option after the *oldlib* entry on the **LIB** command line or after the name you type in response to the "Library name" prompt:

/P[AGESIZE]:*number*

The *number* specifies the new library-page size. It must be an integer value representing a power of 2 between the values 16 and 32,768.

The library-page size determines the number of modules the library can hold; thus, increasing the page size allows you to include more modules in the library. However, the larger the page size, the larger the amount of wasted storage space in the library (on average, *pagesize*/2 bytes). In most cases you should use a small page size unless you need to put a very large number of modules in a library.

The page size also determines the maximum possible size of the library. This limit is *number* * 65,536. For example, if you invoke LIB with the option /P:16, the library must be smaller than one megabyte (16 * 65,536 bytes).

APPENDIXES

A Converting BASICA Programs to QuickBASIC 237
B Differences from Previous Versions of QuickBASIC 243
C Calling C and Assembly-Language Routines 273
D Command Summary 335

APPENDIX A
CONVERTING BASICA PROGRAMS TO QUICKBASIC

A.1 Source-File Format 239
A.2 Cassette BASIC 240
A.3 Statements and Functions Prohibited by QuickBASIC 240
A.4 Statements Requiring Modification 240

The BASIC language understood by QuickBASIC is generally compatible with the BASIC interpreters used on IBM Personal Computers and compatibles: IBM Advanced Personal Computer BASIC Version A3.00 (commonly known as BASICA) and Microsoft GW-BASIC®. However, a few changes are required because of internal differences between QuickBASIC and these BASIC interpreters. Since the changes apply equally to both interpreters, this appendix uses the term BASICA to refer to both GW-BASIC and BASICA.

This appendix provides the following information on converting files:

- Compatible source-file format
- Statements and functions prohibited or requiring modification

The following sections describe only the changes required to compile and run a BASICA program with QuickBASIC Version 4.0. See *Programming in BASIC: Selected Topics* for information on how to use QuickBASIC features to enhance your existing BASICA programs.

A.1 Source-File Format

QuickBASIC Version 4.0 expects the source file to be in ASCII format or in QuickBASIC's format. If you create a file with BASICA, it must be saved with the ,A option; otherwise, BASICA compresses the text of your program in a special format that QuickBASIC cannot read. If this happens, reload BASICA and resave the file in ASCII format, using the ,A option. For example, the following BASICA command saves the file MYPROG.BAS in ASCII format:

```
SAVE "MYPROG.BAS",A
```

A.2 Cassette BASIC

QuickBASIC does not support the cassette device.

A.3 Statements and Functions Prohibited by QuickBASIC

The statements and functions listed below cannot be used in a QuickBASIC program because they perform editing operations on the source file, interfere with program execution, refer to the cassette device, or duplicate support provided by the QuickBASIC environment.

AUTO	LIST	NEW
CONT	LLIST	RENUM
DEF USR	LOAD	SAVE
DELETE	MERGE	USR
EDIT	MOTOR	

A.4 Statements Requiring Modification

If your BASICA program contains any of the statements listed in Table A.1, you probably need to modify your source code before you can compile and run the program. See the *BASIC Language Reference* for a detailed description of these statements.

Table A.1
Statements Requiring Modification

Statement	Modification
BLOAD/BSAVE	Memory locations may be different in QuickBASIC.
CALL *name*	The *name* argument is the name of the **FUNCTION** or **SUB** procedure being called.
CHAIN	QuickBASIC does not support the **ALL**, **MERGE**, **DELETE**, or *linenumber* options.
COMMON	**COMMON** statements must appear before any executable statements.
DEF*type*	**DEF***type* statements should be moved to the beginning of the BASICA source file.
DIM	All **DIM** statements declaring static arrays must appear at the beginning of QuickBASIC programs. See Chapter 2, "Data Types," in the *BASIC Language Reference* for information on **$DYNAMIC**.
DRAW	QuickBASIC requires that the **VARPTR$** function be used with embedded variables.
PLAY	QuickBASIC requires that the **VARPTR$** function be used with embedded variables.
RESUME	If an error occurs in a single-line function, QuickBASIC attempts to resume program execution at the line containing the function.
RUN	For executable files produced by QuickBASIC, the object of a **RUN** or **CHAIN** statement cannot be a .BAS file; it must be an executable file. The R option in BASICA is not supported. While running in the environment, the object of a **RUN** or **CHAIN** statement is still a .BAS file. **RUN** {*linenumber* | *linelabel*}, however, is supported in QuickBASIC; this statement restarts the program at the specified line.

APPENDIX B
DIFFERENCES FROM PREVIOUS VERSIONS OF QUICKBASIC

B.1 New Features 245
B.1.1 User-Defined Types 247
B.1.2 IEEE Format and Math-Coprocessor Support 247
B.1.2.1 Ranges of IEEE-Format Numbers 248
B.1.2.2 PRINT USING and IEEE-Format Numbers 249
B.1.2.3 Recompiling Old Programs with /mbf 249
B.1.2.4 Converting Files and Programs 250
B.1.3 On-Line Help 252
B.1.4 Long (32-Bit) Integers 252
B.1.5 Fixed-Length Strings 253
B.1.6 Syntax Checking on Entry 253
B.1.7 Binary File I/O 253
B.1.8 FUNCTION Procedures 253
B.1.9 Support for the CodeView Debugger 254
B.1.10 Other-Language Compatibility 254
B.1.11 Multiple Modules in Memory 254
B.1.12 ProKey™, SideKick®, and SuperKey® Compatibility 254
B.1.13 Insert and Overtype Modes 255
B.1.14 WordStar®-Style Keyboard Commands 255
B.1.15 Recursion 255
B.1.16 Error Listings during Separate Compilation 256
B.1.17 Assembly-Language Listings during Separate Compilation 256
B.2 Differences in the Environment 256
B.2.1 Choosing Commands and Options 257
B.2.2 Windows 257
B.2.3 New Menus 257
B.2.4 Menu Commands 258
B.2.5 Editing-Key Changes 261

B.3 Differences in Compiling and Debugging 262
B.3.1 Command-Line Differences 262
B.3.2 Separate Compilation Differences 263
B.3.3 User Libraries and BUILDLIB 263
B.3.4 Restrictions on Include Files 264
B.3.5 Debugging 264
B.4 Changes to the BASIC Language 266
B.5 File Compatibility 272

QuickBASIC Version 4.0 contains many new features and enhancements over previous versions of QuickBASIC. This appendix describes the differences between Versions 2.0 and 4.0 of QuickBASIC. Unless otherwise stated, differences between Versions 2.0 and 4.0 also apply as differences between Versions 3.0 and 4.0.

This appendix provides the following information on improvements in QuickBASIC Version 4.0:

- Product capabilities and features
- Environment enhancements
- Improvements in compiling and debugging
- Language changes
- File compatibility

B.1 New Features

This section describes all the new features of Microsoft QuickBASIC Version 4.0. The features are listed in Table B.1 and described in Sections B.1.1–B.1.17.

Table B.1
New Features of Microsoft QuickBASIC Version 4.0

Feature	QuickBASIC Version 2.0	QuickBASIC Version 3.0	QuickBASIC Version 4.0
User-defined types	No	No	Yes
IEEE format, math coprocessor support	No	Yes	Yes
On-line Help	No	No	Yes
Long (32-bit) integers	No	No	Yes
Fixed-length strings	No	No	Yes
Syntax checking on entry	No	No	Yes
Binary file I/O	No	No	Yes
FUNCTION procedures	No	No	Yes
CodeView support	No	No	Yes
Other-language compatibility	No	No	Yes
Multiple modules in memory	No	No	Yes
ProKey™, SideKick®, and SuperKey® compatibility	No	Yes	Yes
Insert/overtype modes	No	Yes	Yes
WordStar-style keyboard interface	No	No	Yes
Recursion	No	No	Yes
Error listings during separate compilation	No	Yes	Yes
Assembly-language listings during separate compilation	No	Yes	Yes

B.1.1 User-Defined Types

The **TYPE** statement allows you to create composite data types by mixing string elements with numeric elements. Such data types are similar to C-language structures. User-defined types are discussed in Chapter 2, "Data Types," in the *BASIC Language Reference*, and Chapter 3, "File and Device I/O," in *Programming in BASIC: Selected Topics*.

B.1.2 IEEE Format and Math-Coprocessor Support

Microsoft QuickBASIC now provides IEEE-format numbers and math-coprocessor support. When no coprocessor is present, QuickBASIC emulates the coprocessor.

Calculations done by programs compiled with QuickBASIC Version 4.0 are generally more accurate and may produce results different from programs run under BASICA or previous versions of QuickBASIC. Single-precision IEEE-format numbers provide an additional decimal digit of accuracy. When compared to Microsoft Binary format double-precision numbers, IEEE-format double-precision numbers provide an additional one to two digits in the mantissa and extend the range of the exponent.

There are two ways to use QuickBASIC Version 4.0 with your old programs and data:

1. Use the/mbf option. This way, you can use your old programs and data files without rewriting your programs or changing your files. See Section B.1.2.3 for more information.
2. Modify your data files and use QuickBASIC Version 4.0 to recompile your programs. In the long run, this ensures compatibility with future versions of QuickBASIC and may produce faster programs. Only random-access files containing binary format real numbers need to be changed. Files containing only integers or string data can be used without modification. See Section B.1.2.4 for more information on converting files and doing file input and output of real numbers.

Note

If assembly-language procedures that use real numbers are called from your program, they must be written so that they use IEEE-format numbers. This is the default for Microsoft Macro Assembler (MASM) Version 5.0. With earlier versions, be sure to use the /R command-line option or the 8087 assembler directive.

B.1.2.1 Ranges of IEEE-Format Numbers

IEEE-format numbers have a wider range than Microsoft Binary format numbers, as shown in the following list:

Type of Number	Range of Values
Single precision	$-3.37 * 10^{38}$ to $-8.43 * 10^{-37}$
	True zero
	$8.43 * 10^{-37}$ to $3.37 * 10^{38}$
Double precision	$-1.67 * 10^{308}$ to $-4.19 * 10^{-307}$
	True zero
	$4.19 * 10^{-307}$ to $1.67 * 10^{308}$

Single-precision values are accurate to approximately 7 digits. Double-precision values are accurate to either 15 or 16 digits.

Note that double-precision values may have three digits in the exponent. This may cause problems in **PRINT USING** statements. See Section B.1.2.2 for more information.

B.1.2.2 PRINT USING and IEEE-Format Numbers

Because double-precision IEEE-format numbers can have larger exponents than Microsoft Binary format double-precision numbers, you may need to use a special exponential format in **PRINT USING** statements. Use the new format if your program prints values with three-digit exponents. To print numbers with three-digit exponents, use five carets (^^^^^) instead of four carets to indicate exponential format:

```
PRINT USING "+#.######^^^^^", Circumference#
```

If an exponent is too large for a field, QuickBASIC replaces the first digit with a percent sign (%) to indicate the problem.

B.1.2.3 Recompiling Old Programs with /mbf

Old programs and files work with QuickBASIC Version 4.0 without modification if you recompile the programs with the/mbf command-line option.

For example, to compile the program named multrgrs.bas, enter the following command line:

```
bc multrgrs /mbf;
```

Then, link the program as you usually do.

To recompile using the Make EXE File dialog box, be sure to use the /mbf option on the qb command line when you start QuickBASIC. Then, compile as you normally would.

The /mbf option converts Microsoft Binary format numbers to IEEE format as they are read from a random-access file and converts them back to Microsoft Binary format before writing them to a file. This lets you do calculations with IEEE-format numbers while keeping the numbers in Microsoft Binary format in your files.

B.1.2.4 Converting Files and Programs

If you decide to convert your programs and data files rather than use the /mbf command-line option, you need to do two things:

1. Recompile your programs.
2. Convert your data files.

Your old QuickBASIC programs should compile without modification. However, do not use the /mbf option when recompiling. Your program will not work with your new data files if you use the /mbf option.

Data files containing real numbers need to be converted so that real numbers are stored in IEEE format. QuickBASIC Version 4.0 includes eight functions that help you do this.

Note

You do not need to convert your data files if they contain only integer and string data. Only files containing real numbers must be converted.

Version 4.0 has the familiar **CVS**, **CVD**, **MKS$**, and **MKD$** functions for reading and writing real numbers from or to random-access files. However, in Version 4.0 these functions deal with real numbers stored in your files in IEEE format, not Microsoft Binary format. To handle numbers in Microsoft Binary format, Version 4.0 includes the functions **CVSMBF**, **CVDMBF**, **MKSMBF$**, and **MKDMBF$**.

With these functions, you can write a short program that reads records from the old file (using Microsoft Binary format as necessary), converts the real-number fields to IEEE format, places the fields in a new record, and writes out the new record.

■ Examples

The following program copies an old data file to a new file, making the necessary conversions:

```
' Define types for old and new file buffers:
TYPE Oldbuf
   ObsId AS STRING*20
   XPos AS STRING*4
   YPos AS STRING*4
   FuncVal AS STRING*8
END TYPE

TYPE NewBuf
   ObsId AS STRING*20
   XPos AS SINGLE
   YPos AS SINGLE
   FuncVal AS DOUBLE
END TYPE

' Declare buffers:
DIM OldFile AS Oldbuf, NewFile AS NewBuf

' Open the old and new data files:
OPEN "OLDMBF.DAT" FOR RANDOM AS #1 LEN=LEN(OldFile)
OPEN "NEWIEEE.DAT" FOR RANDOM AS #2 LEN=LEN(NewFile)

I=1

' Read the first old record:
GET #1,I,OldFile

DO UNTIL EOF(1)

' Move the fields to the new record fields, converting
' the real-number fields:
   NewFile.ObsId=OldFile.ObsId
   NewFile.XPos=CVSMBF(OldFile.XPos)
   NewFile.YPos=CVSMBF(OldFile.YPos)
   NewFile.FuncVal=CVDMBF(OldFile.FuncVal)

' Write the converted fields to the new file:
   PUT #2,I,NewFile
   I=I+1

' Read next record from the old file:
   GET #1,I,OldFile
LOOP

CLOSE #1, #2
```

Each record in the two files has four fields: an identifier field, two fields containing single-precision real numbers, and a final field containing a double-precision real number.

Most of the conversion work is done with the functions **CVDMBF** and **CVSMBF**. For example, the following program line converts the double-precision field:

```
NewFile.FuncVal=CVDMBF(OldFile.FuncVal)
```

The eight bytes in the record field OldFile.FuncVal are converted from a Microsoft Binary format double-precision value to an IEEE-format double-precision value by the function **CVDMBF**. This value is stored in the corresponding field in the new record, which is later written to the new file by the **PUT** statement.

See the appropriate sections of the *BASIC Language Reference* for more information about **CVDMBF** and related functions.

B.1.3 On-Line Help

QuickBASIC Version 4.0 includes several kinds of on-line Help, including general Help, Help on menu commands, and context-sensitive Help on BASIC syntax.

B.1.4 Long (32-Bit) Integers

Long (32-bit) integers eliminate rounding errors in calculations with numbers in the range –2,147,483,648 to 2,147,483,647, and they provide much faster whole-number calculations in this range than do floating-point types.

B.1.5 Fixed-Length Strings

Fixed-length strings let you incorporate string data into user-defined types. See Chapter 2, "Data Types," in the *BASIC Language Reference*, and Chapter 4, "String Processing," in *Programming in BASIC: Selected Topics*, for more information.

B.1.6 Syntax Checking on Entry

If syntax checking is turned on, QuickBASIC checks each line for syntax, duplicate definition, and out of memory errors as you enter it. See Chapter 5, "Editing," for an explanation of syntax checking upon entry and other smart-editor features.

B.1.7 Binary File I/O

Version 4.0 provides binary access to files. This is useful for reading and modifying files saved in non-ASCII format. The major benefit of binary access is that it does not force you to deal with a file as a collection of records. If a file is opened in binary mode, you can move forward or backward to any byte position in the file, then read or write as many bytes as you want. Thus, different I/O operations on the same file can get or put a varying number of bytes—unlike random access, where the number of bytes is fixed by the predefined length of a single record.

See Chapter 3, "File and Device I/O," in *Programming in BASIC: Selected Topics*, for more information about accessing binary files.

B.1.8 FUNCTION Procedures

FUNCTION procedures allow you to place a function in one module and call it from a different module. See Chapter 2, "Procedures: Subprograms and Functions" in *Programming in BASIC: Selected Topics*, and the *BASIC Language Reference*, for more information about using **FUNCTION** procedures.

In previous versions, a **SUB** procedure and variable could have the same name. In QuickBASIC, Version 4.0, both **SUB** and **FUNCTION** procedures with the same name as a variable cause a "Duplicate definition" error message.

B.1.9 Support for the CodeView Debugger

You can use the command-line tools BC.EXE and LINK.EXE to create executable files compatible with the Microsoft CodeView debugger, a powerful tool included with the Microsoft Macro Assembler (Version 5.0) or Microsoft C (Version 5.0). Modules compiled with BC can be linked with modules compiled with other Microsoft languages in such a way that the final executable file can be debugged using the CodeView debugger. See Chapter 9, "Compiling and Linking from DOS," and Appendix C, "Calling C and Assembly-Language Routines," for more information.

B.1.10 Other-Language Compatibility

QuickBASIC Version 4.0 allows you to call routines written in other Microsoft languages using C or Pascal calling conventions. Arguments are passed by NEAR or FAR reference, or by value. Other-language code may be placed in Quick libraries or linked into executable files.

The PTR86 routine is not supported in QuickBASIC, Version 4.0; use the **VARPTR** and **VARSEG** functions instead.

B.1.11 Multiple Modules in Memory

QuickBASIC Version 4.0 allows you to load multiple program modules into memory simultaneously. Previous versions of QuickBASIC permitted only one module to be in memory at a time. Now you can edit, execute, and debug multiple-module programs within the QuickBASIC environment. See Chapter 4, "Managing Source Files," in this manual, and Chapter 2, "Procedures: Subprograms and Functions," in *Programming in BASIC: Selected Topics*, for more information about using multiple modules.

B.1.12 ProKey™, SideKick®, and SuperKey® Compatibility

You can use ProKey, SideKick, and SuperKey with the Microsoft QuickBASIC Version 4.0 program development environment. Other keyboard-reprogramming or desktop software may not work with QuickBASIC. Check with the suppliers or manufacturers of other products to find out about the product's compatibility with QuickBASIC Version 4.0.

B.1.13 Insert and Overtype Modes

Pressing INS toggles the editing mode between insert and overtype. When overtype mode is on, the cursor changes from a blinking underline to a block. Note that INS replaces the CTRL+O insert/overtype toggle in Version 3.0.

In insert mode, the editor inserts a typed character at the cursor position. In overtype mode, the editor replaces the character under the cursor with the character you type. Insert mode is the default mode.

B.1.14 WordStar®-Style Keyboard Commands

QuickBASIC supports many of the key sequences familiar to users of WordStar. A complete list of WordStar-style key commands appears in Chapter 5, "Editing," Appendix D, "Command Summary," and the QuickBASIC 4.0 Quick Reference Card.

B.1.15 Recursion

QuickBASIC Version 4.0 supports recursion, which is the ability of a procedure to call itself. Recursion is useful for solving certain problems, such as sorting. See Chapter 4, "Programs and Modules," in the *BASIC Language Reference*, and Chapter 2, "Procedures: Subprograms and Functions," in *Programming in BASIC: Selected Topics*, for more information about using recursion in QuickBASIC programs.

B.1.16 Error Listings during Separate Compilation

QuickBASIC Version 4.0 now displays descriptive error messages when you compile programs using the bc command. See Chapter 9, "Compiling and Linking from DOS," for more information about the bc command.

Using BC, you can redirect the error messages to a file or device to get a copy of the compilation errors.

■ **Examples**

The following examples show how to use the bc command line for error display:

Command Line	Action
`bc test.bas;`	Compiles the file named test.bas and displays errors on the screen
`bc test.bas; > test.err`	Compiles the file test.bas and redirects error messages to the file named test.err

B.1.17 Assembly-Language Listings during Separate Compilation

The bc command's /a option produces a listing of the assembly-language code produced by the compiler. See Section 9.4, "Compiling with the bc Command," for more information about this option.

B.2 Differences in the Environment

The QuickBASIC 4.0 programming environment provides more flexible command selection, additional window options, and many more menu commands than do previous versions of QuickBASIC. Sections B.2.1–B.2.5 describes the differences in the programming environment between Version 4.0 and earlier versions.

B.2.1 Choosing Commands and Options

QuickBASIC Version 4.0 gives you greater flexibility in how you choose commands from menus, and options from dialog boxes.

Version 4.0 allows you to select any menu by pressing the ALT key followed by a mnemonic key. Each menu command and dialog box option also has its own mnemonic key, which immediately executes the command or selects the item. The mnemonic keys appear red on a color monitor, highlighted on a CGA adapter/composite display combination, and underlined on a monochrome adapter/monochrome display combination.

In Version 4.0 the ENTER key functions the same way as the SPACEBAR did in earlier versions. You can press ENTER to execute a command from a dialog box.

B.2.2 Windows

Version 4.0 allows up to two work area windows (referred to as View windows) and a separate Immediate window. QuickBASIC Version 2.0 supported only two windows: the work area, and the error-message window. See Section 3.4 for more information about windows.

B.2.3 New Menus

QuickBASIC Version 4.0 has three new menus, listed in Table B.2.

Table B.2

New Menus in QuickBASIC Version 4.0

Menu	Description
Debug	Replaces the debugging commands in Version 3.0. See Chapter 7, "Debugging While You Program," for more information about the Debug menu.
Calls	Displays the sequence of calls nested within other calls. See Section 7.4.1 for more information about the Calls menu.
Help	Provides access to information about QuickBASIC editing and menu commands and BASIC syntax. See Section 2.4.5 for more information about the Help menu.

B.2.4 Menu Commands

The File, Edit, View, and Run menus, which appeared in previous versions of QuickBASIC, all have some new commands. Table B.3 lists the new commands in these menus in QuickBASIC Version 4.0.

Table B.3
New Menu Commands in QuickBASIC Version 4.0

Menu	Command	Description
File	New Program	Same as New in Version 2.0
	Open Program	Same as Load... in Version 2.0
	Merge	Inserts a file before the cursor position in the active window (see Section 4.7)
	Save As	Names and saves untitled program
	Save All	Replaces Auto Save in Version 2.0 (see Section 4.2.2)
	Create File	Creates a new module, include file or document (see Section 4.3.1)
	Load File	Loads a module, include file or document from disk (see Section 4.3.1)
	Unload File	Removes a module, include file or document from memory (see Section 4.4.2)
	DOS Shell	Same as Shell in Version 2.0
	Exit	Same as Quit in Version 2.0
Edit	Undo	Shortcut key is now ALT+BKSP.
	Cut	Shortcut key is now SHIFT+DEL.
	Copy	Shortcut key is now CTRL+INS.
	Paste	Shortcut key is now SHIFT+INS.
	Clear	Deletes without copying to Clipboard (see Section 5.5.2)
	New SUB	Creates new **SUB** procedures (see Section 6.3.1)
	New FUNCTION	Creates new **FUNCTION** procedures (see Section 6.3.1)
	Syntax Checking	Turns automatic syntax checking on and off (see Section 5.2.3)

Table B.3 *(continued)*

Menu	Command	Description
View	SUBs	Shows modules and procedures currently loaded. Edits, moves, and deletes modules and procedures (see Section 6.3.3).
	Next SUB	Pages through loaded subprograms and **FUNCTION** procedures in the current module (see Section 6.3.3)
	Split	Opens or closes a second View window (see Section 6.3.3)
	Next Statement	Moves cursor to next statement to be executed (see Section 7.3.5.4)
	Output Screen	Views program output (see Section 6.1.4)
	Include File	Edits include file (see Section 4.5.2)
	Included Lines	Views include file (see Section 4.5.2)
	Options	Dialog box enhanced to reflect the new Version 4.0 screen elements.
Run	Start	Same as Version 2.0
	Restart	Resets variable values in preparation for restarting single stepping (see Section 7.3.5.4)
	Continue	Continues execution after a suspension of execution (see Section 7.3.5.1)
	Modify COMMAND$	Sets the string read by **COMMAND$** (see Section 6.1.6)
	Make EXE File	Creates an executable file (see Section 6.2)
	Make Library	Creates a Quick library (see Section 8.3.3)
	Set Main Module	Specifies a new main module (see Section 4.4)

The following Version 2.0 menu commands are removed from Version 4.0:

- Errors (View menu)

  QuickBASIC Version 4.0 displays error messages as you enter or run the program.

- Next Error (Search menu)

  Because errors are noted as the program is entered or run, there is no need to search for error locations.

- Compile (Run menu)

  The Compile command is replaced by the Run menu's Start command, since your program is always ready to run in QuickBASIC Version 4.0.

- Compile... (Run menu)

  The Compile... command is replaced by the Run menu's Make EXE File command. It is no longer necessary to set compiler options when running in memory or when creating an executable file with the Make EXE File command.

The Version 3.0 Debug option is removed from the Run menu. In Version 4.0 you can debug at any time, using the debugging commands in the Debug menu.

B.2.5 Editing-Key Changes

The QuickBASIC Version 4.0 keyboard interface has been extended to include editing key sequences similar to those in the WordStar editor. See Section D.4 for a list of all of QuickBASIC's editing commands. The functions performed by the QuickBASIC Version 2.0 key sequences listed in Table B.4 are changed in QuickBASIC Version 4.0.

Table B.4

Editing-Key Changes

Function	QuickBASIC 2.0 Key	QuickBASIC 4.0 Key
Undo	SHIFT+ESC	ALT+BKSP
Cut	DEL	SHIFT+DEL
Copy	F2	CTRL+INS
Paste	INS	SHIFT+INS
Clear	---	DEL
Overtype	---	INS

B.3 Differences in Compiling and Debugging

In the QuickBASIC 4.0 programming environment, compiling and debugging are not separate operations. Your program is always ready to run, and you can debug your code in several ways while you program. This section describes the differences in compiling and debugging features between QuickBASIC Version 4.0 and previous versions.

B.3.1 Command-Line Differences

QuickBASIC Version 4.0 supports Version 2.0 command-line options only for the qb command, not the bc command. To compile a program outside of the QuickBASIC environment, use the bc command, which is described in Section 9.4.

Version 4.0 does not require any of the Compile options listed in Table 4.1 of the Microsoft QuickBASIC Version 2.0 manual. If you attempt to invoke QuickBASIC 4.0 using these as command-line options, an error message appears. Similarly, it was necessary to choose certain compiler options from the Compile dialog box in Version 3.0; this is now done automatically by Version 4.0. Table B.5 describes the way in which QuickBASIC Version 4.0 supports the functionality of these options:

Table B.5

qb and bc Options Not Used in QuickBASIC Version 4.0

Version 2.0	Version 4.0
On Error (/e)	Automatically set when an **ON ERROR** statement is present
Debug (/d)	Always on when you run a program within the QuickBASIC environment. When producing executable programs on disk or Quick libraries, use the Produce Debug Code option.
Checking between Statements (/v) and Event Trapping (/w)	Automatically set when an **ON** *event* statement is present
Resume Next (/x)	Automatically set when **RESUME NEXT** statement is present
Arrays in Row Order (/r)	Available only when compiling with bc
Minimize String Data (/s)	The default for qb. To turn off the Minimize String Data option, you must compile from the command line with bc.

The options listed in Table B.6 are now available for the qb and bc commands as noted:

Table B.6

New Options for the qb and bc Commands

Option	Description
/ah	Allows dynamic arrays of records, fixed-length strings, and numeric data to be larger than 64K each. If this option is not specified, the maximum size for each array is 64K. Note that this option has no effect on the way data items are passed to procedures. (This option is used with the qb and bc commands.)
/h	Displays the highest resolution possible on your hardware. For example, if you have an EGA, QuickBASIC displays 43 lines and 80 columns of text. (This option is used only with the qb command.)
/mbf	Converts Microsoft Binary format numbers to IEEE format. See Section B.1.2.3 for more information about this option. (This option is used with the qb and bc commands.)
/run *sourcefile*	Runs *sourcefile* immediately, without first displaying the QuickBASIC programming environment. (This option is used only with the qb command.)

B.3.2 Separate Compilation Differences

Version 4.0 does not allow separate compilation with the qb command. Use the bc command described in Section 9.4 to compile and link files without entering the programming environment.

B.3.3 User Libraries and BUILDLIB

User libraries created for previous versions are not compatible with Version 4.0. You must rebuild the library from the original source files.

In Version 4.0, user libraries are called Quick libraries. There is no change in their function or use. The file-name extension of these libraries is now .QLB instead of .EXE. The BUILDLIB utility is no longer required. Quick libraries are now created from within the programming environment or from the link command line. See Chapter 8, "Quick Libraries," for more information.

B.3.4 Restrictions on Include Files

Include files cannot contain **SUB** or **FUNCTION** procedure definitions. In such cases, use the Merge command from the File menu to insert the include file into the current module. When you merge an include file containing a **SUB** procedure, the text of the procedure does not appear in the currently active window. To view or edit its text, press F2, then select the procedure name in the list box. Once the text is merged, you can run the program.

Alternatively, you may decide to put your **SUB** procedure in a separate module. In this case, you must take one of the following two steps for any shared variables (variables declared in a **COMMON SHARED** or [**RE**]**DIM SHARED** statement outside the **SUB** procedure, or in a **SHARED** statement inside the **SUB** procedure), because variables declared this way are shared only within a single module:

1. Share the variables between the modules by listing them in **COMMON** statements at the module level in both modules.
2. Pass the variables to the **SUB** procedure in an argument list.

See Chapter 4, "Managing Source Files," for a detailed explanation of include files and modules in QuickBASIC. See Chapter 2, "Procedures: Subprograms and Functions," in *Programming in BASIC: Selected Topics*, and Chapter 4, "Programs and Modules," in the *BASIC Language Reference* for additional information on modules and procedures.

B.3.5 Debugging

QuickBASIC Version 4.0 helps you debug your programs faster by providing the following debugging features:

- Multiple breakpoints
- Watch expressions and watchpoints

- Improved program tracing
- Full-screen window that displays source-level program text during single-stepping
- The ability to change variable values during execution, then continue execution
- The ability to edit the program, then continue execution without restarting
- History
- Calls Menu

The debugging function-key sequences listed in Table B.7 are changed in QuickBASIC Version 4.0:

Table B.7
Debugging-Key Changes

Function	QuickBASIC Version 2.0 Key	QuickBASIC Version 4.0 Key
Trace	ALT+F8	F8
Step	ALT+F9	F10

Note that Animate mode is turned on when you toggle on the Trace On command from the Debug menu, then run your program.

See Chapter 7, "Debugging While You Program," for more information.

B.4 Changes to the BASIC Language

This section describes enhancements and changes to the BASIC language in Version 4.0 and earlier versions of QuickBASIC. Table B.8 lists the keywords affected by these changes and shows which version of QuickBASIC is affected by each change. A more detailed explanation of the changes appears after Table B.8. See the *BASIC Language Reference* for complete information about the BASIC language.

Table B.8

Changes to the BASIC Language

Keyword	QuickBASIC Version 2.0	QuickBASIC Version 3.0	QuickBASIC Version 4.0
AS	No	No	Yes
CALL	No	No	Yes
CASE	No	Yes	Yes
CLEAR	No	No	Yes
CLNG	No	No	Yes
CLS	No	Yes	Yes
COLOR	No	No	Yes
CONST	No	Yes	Yes
CVL	No	No	Yes
CVSMBF, CVDMBF	No	Yes	Yes
DECLARE	No	No	Yes
DEFLNG	No	No	Yes
DIM	No	No	Yes
DO...LOOP	No	Yes	Yes
EXIT	No	Yes	Yes
FILEATTR	No	No	Yes
FREEFILE	No	No	Yes
FUNCTION	No	No	Yes
GET	No	No	Yes

Table B.8 *(continued)*

Keyword	QuickBASIC Version 2.0	QuickBASIC Version 3.0	QuickBASIC Version 4.0
LCASE$	No	No	Yes
LEN	No	No	Yes
LSET	No	No	Yes
LTRIM$	No	No	Yes
MKL$	No	No	Yes
MKSMBF$, MKDMBF$	No	Yes	Yes
OPEN	No	No	Yes
PALETTE	No	No	Yes
PUT	No	No	Yes
RTRIM$	No	No	Yes
SCREEN	No	No	Yes
SEEK	No	No	Yes
SELECT CASE	No	Yes	Yes
SETMEM	No	No	Yes
STATIC	No	No	Yes
TYPE	No	No	Yes
UCASE$	No	No	Yes
VARPTR	No	No	Yes
VARSEG	No	No	Yes
WIDTH	No	Yes	Yes

The following section explains in more detail the differences in the keywords summarized in Table B.8.

Keywords	Explanation
AS	The **AS** clause allows the use of user-defined types in **DIM**, **COMMON**, and **SHARED** statements, and in **DECLARE**, **SUB**, and **FUNCTION** parameter lists.
CALL	The use of **CALL** is optional for calling subprograms declared with the **DECLARE** statement.
CLEAR	The **CLEAR** statement no longer sets the total size of the stack. In Version 4.0, the **CLEAR** statement sets only the stack size required by the program. QuickBASIC sets the stack size to the amount specified by the **CLEAR** statement, plus what QuickBASIC itself requires.
CLNG	The **CLNG** function rounds its argument and returns a long (four-byte) integer equal to the argument.
CLS	The **CLS** statement has been modified to give you greater flexibility in clearing the screen. Note that QuickBASIC no longer clears the screen automatically at the beginning of each program as was true in earlier versions.
COLOR, **SCREEN**, **PALETTE**, **WIDTH**	The **COLOR**, **SCREEN**, **PALETTE**, and **WIDTH** statements are extended to include screen modes available with the IBM PS/2 VGA and Multicolor Graphics Array (MCGA) cards.
CONST	**CONST** statements let you define symbolic constants to improve program readability and ease program maintenance.
CVL	The **CVL** function is used to read long integers stored as strings in random-access data files. **CVL** converts a four-byte string created with the **MKL$** function back to a long integer for use in your BASIC program.

CVSMBF, CVDMBF	The **CVSMBF** and **CVDMBF** functions convert strings containing Microsoft Binary format numbers to IEEE-format numbers.
DECLARE	The **DECLARE** statement allows you to call procedures from different modules, check the number and type of arguments passed, and call procedures before they are defined.
DEFLNG	The **DEFLNG** statement declares all variables, **DEF FN** functions, and **FUNCTION** procedures as having the long-integer type. That is, unless a variable or function has been declared in an **AS** *type* clause, or it has an explicit type-definition suffix such as % or $, it is a long integer by default.
DIM	The **DIM** statement's **TO** clause lets you specify subscripts of any integer value, giving you greater flexibility in array declarations.
DO...LOOP	**DO...LOOP** statements give you more powerful loops that allow you to write better structured programs.
EXIT	**EXIT {DEF | DO | FOR | FUNCTION | SUB}** statements provide convenient exits from loops and procedures.
FREEFILE, FILEATTR	The **FREEFILE** and **FILEATTR** functions help you write applications that do file I/O in a multiple-module environment.
FUNCTION	The **FUNCTION...END FUNCTION** procedure allows you to define a multiline procedure that you call from within an expression. These procedures behave much like intrinsic functions such as **ABS** or multiline **DEF FN** functions from QuickBASIC Versions 1.0–3.0. However, unlike a **DEF FN** function, a **FUNCTION** procedure can be defined in one module and called from another. Also, **FUNCTION** procedures have local variables and support recursion.

GET, PUT

For I/O operations, the syntax of the **GET** and **PUT** statements is expanded to include records defined with **TYPE...END TYPE** statements. This makes use of the **FIELD** statement unnecessary.

LCASE$, UCASE$, LTRIM$, RTRIM$

The following string-handling functions are available in Version 4.0:

Function	Return Value
LCASE$	A copy of the string with all letters converted to lowercase
UCASE$	A copy of the string with all letters converted to uppercase
LTRIM$	A copy of the string with all leading blanks removed
RTRIM$	A copy of the string with all trailing blanks removed

LEN

The **LEN** function has been extended to return the number of bytes required by any scalar or record variable, constant, expression, or array element.

LSET

The **LSET** statement is extended to include record variables as well as string variables. This allows you to assign one record variable to another record variable even when the records are not similar.

MKL$

The **MKL$** function is used to convert long integers to strings that can be stored in random-access data files. Use the **CVL** function to change the string back to a long integer.

MKSMBF$, MKDMBF$

The **MKSMBF$** and **MKDMBF$** functions convert IEEE-format numbers to strings containing a Microsoft Binary format number. This allows you to use data files created for versions prior to 4.0 with Version 4.0.

OPEN

The **OPEN** statement now opens two files with the same name for **OUTPUT** or **APPEND** as long as the path names are different. For example, the following is now permitted:

```
OPEN "SAMPLE" FOR APPEND AS #1
OPEN "TMP\SAMPLE" FOR APPEND AS #2
```

A binary file mode has been added to the **OPEN** statement syntax. See Chapter 3, "File and Device I/O," in *Programming in BASIC: Selected Topics*, for information about using this mode.

SEEK

The **SEEK** statement and function allow you to position a file at any byte or record. See Chapter 3, "File and Device I/O," in *Programming in BASIC: Selected Topics*, and the entries for the **SEEK** statement and function in the *BASIC Language Reference*, for more information.

SELECT CASE

SELECT CASE statements provide a way to simplify complex condition testing. The **CASE** clause of the **SELECT CASE** statement now accepts any expression (including variable expressions) as an argument; in previous versions, only constant expressions were permitted.

SETMEM

The **SETMEM** function facilitates mixed-language programming by allowing you to decrease the amount of dynamic memory allocated by BASIC so it can be used by procedures in other languages.

STATIC

Omitting the **STATIC** attribute from **SUB** and **FUNCTION** statements causes variables to be allocated when the procedures are called, instead of when they are defined. Such variables do not retain their values between procedure calls.

TYPE	The **TYPE...END TYPE** statement lets you define a data type containing elements of different fundamental types. This simplifies defining and accessing random-access file records so that omitting the **STATIC** keyword allows recursion.
VARPTR	The **VARPTR** function now returns the 16-bit integer offset of the BASIC variable or array element. The offset is from the beginning of the segment that contains the variable or array element.
VARSEG	The **VARSEG** function returns the segment address of its argument. This allows you to set **DEF SEG** appropriately for use with **PEEK**, **POKE**, **BLOAD**, **BSAVE**, and **CALL** ABSOLUTE. It also permits you to get the appropriate segment for use with **CALL** INTERRUPT when executing operating-system or BIOS interrupts.
WIDTH	A new argument on the **WIDTH** statement lets your programs use the extended line modes on machines equipped with EGA, VGA, and MCGA adapter cards.

Important

You can no longer conditionally execute **NEXT** and **WEND** statements using the single-line **IF...THEN...ELSE** statement. See the entry for the single-line **IF...THEN...ELSE** statement in the *BASIC Language Reference* for more information.

B.5 File Compatibility

All versions of QuickBASIC are source-code compatible; source code created for a previous version compiles under Version 4.0, except as noted in Section B.3.4, "Restrictions on Include Files." However, QuickBASIC 4.0 is *not* binary compatible with previous versions. You must recompile object files and user libraries created for previous versions of QuickBASIC.

APPENDIX C
CALLING C AND ASSEMBLY-LANGUAGE ROUTINES

C.1 Organizing Mixed-Language Programs 275

C.2 Mixed-Language Programming Elements 276
- C.2.1 Making Mixed-Language Calls 277
- C.2.2 Naming-Convention Requirement 280
- C.2.3 Calling-Convention Requirement 282
- C.2.4 Parameter-Passing Requirement 283
- C.2.5 Compiling and Linking 284
  - C.2.5.1 Compiling with Proper Memory Models 284
  - C.2.5.2 Linking with Language Libraries 285

C.3 BASIC Calls to C 286
- C.3.1 The BASIC Interface to Other Languages 286
  - C.3.1.1 The DECLARE Statement 286
  - C.3.1.2 Using ALIAS 287
  - C.3.1.3 Using the Parameter List 288
- C.3.2 Alternative BASIC Interfaces 289
- C.3.3 BASIC Calls to C 290
- C.3.4 Restrictions on Calls from BASIC 293
  - C.3.4.1 Memory Allocation 293
  - C.3.4.2 Incompatible Functions 294
  - C.3.4.3 Allocating String Space 295
  - C.3.4.4 Performing I/O on BASIC Files 296
  - C.3.4.5 Events and Errors 296

C.4 C Calls to BASIC 297
- C.4.1 The C Interface to Other Languages 297
- C.4.2 Calls from C to BASIC 299

C.5 Passing Data by Reference or Value 302
- C.5.1 BASIC Arguments 302
- C.5.2 C Arguments 303

C.6 Numeric and String Data in Mixed-Language Programming 304
C.6.1 Integer and Real Numbers 305
C.6.2 Strings 306
C.6.2.1 String Formats 306
C.6.2.2 Passing BASIC Strings 308
C.6.2.3 Passing BASIC Strings to C 309
C.6.2.4 Passing C Strings 310
C.7 Special Data Types 311
C.7.1 Arrays 311
C.7.1.1 Passing Arrays from BASIC 311
C.7.1.2 Array Indexing and Declaration 313
C.7.2 Storage of Structures and User-Defined Types 315
C.8 Pointers, Address Variables, and Common Blocks 316
C.8.1 Common Blocks 316
C.8.2 Using a Varying Number of Parameters 317
C.9 The B_OnExit Routine 319
C.10 Assembly-Language-to-BASIC Interface 321
C.10.1 Writing the Assembly-Language Procedure 321
C.10.1.1 Setting Up the Procedure 322
C.10.1.2 Entering the Procedure 323
C.10.1.3 Allocating Local Data (Optional) 323
C.10.1.4 Preserving Register Values 324
C.10.1.5 Accessing Parameters 325
C.10.1.6 Returning a Value (Optional) 327
C.10.1.7 Numeric Return Values Other Than Two- and Four-Byte Integers 327
C.10.1.8 Exiting the Procedure 329
C.10.2 Calls from BA 330
C.10.3 Using CDECL in Calls from BASIC 332
C.10.4 The Microsoft® Segment Model 333

Mixed-language programming is the process of combining programs from two or more source languages. For example, mixed language programming allows you to use assembly language to enhance your QuickBASIC programs. You can develop most of your program rapidly using QuickBASIC, then use assembly language for routines that are executed many times and must run with utmost speed. Similarly, you can call your own C-language functions from within QuickBASIC programs, and you can call many of the standard Microsoft C library routines as well.

This appendix assumes that you know the languages you wish to combine, and that you know how to write, compile, and link multiple-module programs with these languages.

When you finish this appendix, you will understand

- General issues important in mixed-language programming.
- Interlanguage calling between BASIC, C, and assembly language.
- Passing parameters in interlanguage calls.
- Differences in how BASIC and C handle numeric and string data.
- Using structures and user-defined types in calls between C and BASIC.

Note

See the "Introduction" of *Programming in BASIC: Selected Topics* for the typographic conventions used in this appendix.

C.1 Organizing Mixed-Language Programs

The way you organize mixed-language programs depends on whether you run your program from within the QuickBASIC program development environment, or compile from the DOS command line using **BC**. When your program calls other-language routines from within the environment, the other-language routines must first be compiled and linked into a Quick library, and the Quick library must be loaded with QuickBASIC.

Note

It is especially important that other-language procedures be thoroughly debugged *before* being incorporated in a Quick library. QuickBASIC does not "step into" Quick-library procedures when tracing through a program, so debugging them from within the environment is not possible.

The Microsoft CodeView debugger, provided with Microsoft C, Version 5.0; Microsoft Macro Assembler (MASM), Version 5.0; Microsoft Pascal, Version 4.0; and Microsoft FORTRAN, Version 4.0; allows you to debug mixed-language programs.

If you compile and link your program from the DOS command line, your other-language routines do not have to be part of a library. However, the Microsoft Library Manager (LIB) is provided for this purpose if you find it more convenient. See Chapter 9, "Compiling and Linking from DOS," for information on compiling, linking, and managing libraries.

C.2 Mixed-Language Programming Elements

Microsoft languages have special keywords that facilitate mixed-language programming. To use these keywords, you must understand the basic issues involved.

Sections C.2.1–C.2.4 describe the elements of mixed-language programming: how languages differ and how to resolve these differences. If you understand the principles described in these sections, then you may want to turn directly to later sections in this appendix. Nevertheless, you still may find it helpful to refer to these sections occasionally.

Section C.2.1 presents the basic context of a mixed-language call and describes when and how you make such a call.

Sections C.2.2–C.2.4 present the three fundamental mixed-language programming requirements:

- Naming-convention requirement
- Calling-convention requirement
- Parameter-passing requirement

Section C.2.5 presents the compile-time and link-time issues, including use of memory models.

C.2.1 Making Mixed-Language Calls

Mixed-language programming always involves a function or procedure call. For example, a BASIC main module may need to execute a specific task that you would like to program separately. However, instead of calling a BASIC subprogram, you decide to call a C function.

Mixed-language calls necessarily involve multiple modules. Instead of compiling all of your source modules with the same compiler, you use different compilers. In the example mentioned above, you could compile the main-module source file with **BC**, another source file (written in C) with the C compiler, and then link the two object files using **LINK**. Alternatively, you could compile the C function, and then link it into a Quick library and call the function from a program running within the QuickBASIC environment.

Note

Object files produced with either the Microsoft C Optimizing Compiler or Microsoft QuickC™ are suitable for use in mixed-language programs.

Figure C.1 illustrates the syntax of a mixed-language call in which a BASIC main module calls a C function.

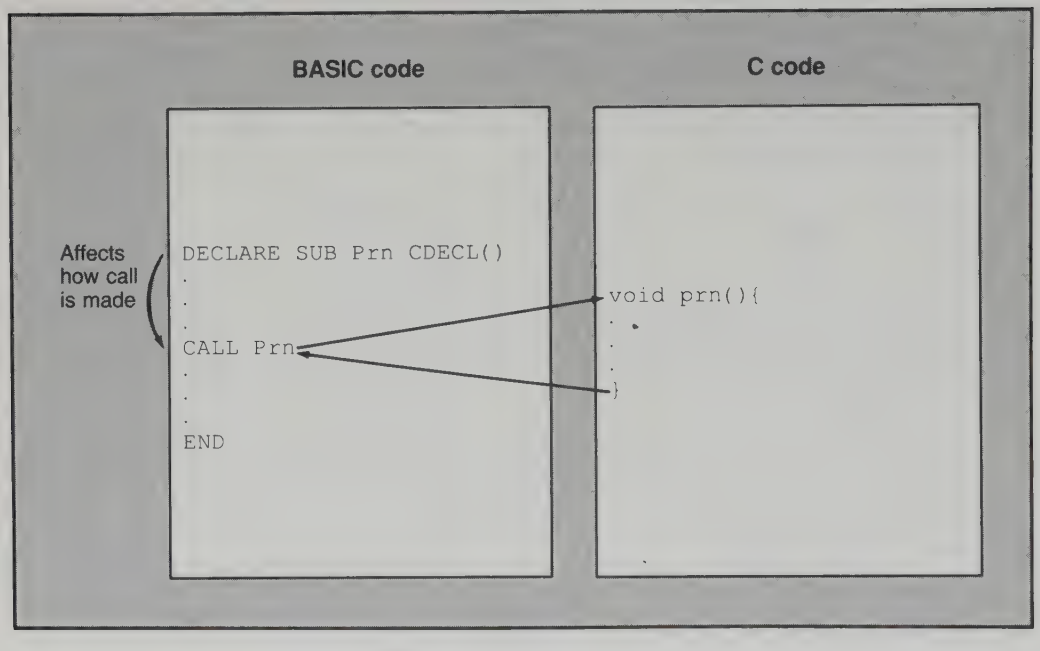


Figure C.1 Mixed-Language Call

In Figure C.1, the BASIC call to C is `CALL Prn`. The form is similar to a call to a BASIC subprogram. However, there are two differences between this mixed-language call and a call between two BASIC modules:

1. The subprogram `Prn` is actually implemented in C, using standard C syntax.

2. The implementation of the call in BASIC is affected by the presence of a **DECLARE** statement, which uses the **CDECL** keyword in order to create compatibility with C.

The **DECLARE** statement is an example of a mixed-language "interface" statement. Each language provides its own form of interface. See Section C.3.1.1, "The DECLARE Statement," in this manual, and the discussions of the **DECLARE** statement in the *BASIC Language Reference*, and *Programming in BASIC: Selected Topics*, for more information.

Despite syntactic differences, QuickBASIC **FUNCTION** and **SUB** procedures are very similar to C functions and assembly-language procedures. The principal difference is that C functions, QuickBASIC **FUNCTION** procedures, and assembly-language procedures can all return values, whereas QuickBASIC **SUB** procedures cannot.

Note

In this appendix, "routine" refers to any C function, QuickBASIC **SUB** or **FUNCTION** procedure, or assembly-language procedure that can be called from another module.

Table 1.1 shows the correspondence between routine calls in different languages.

Table C.1

Language Equivalents for Routine Calls

Language	Return Value	No Return Value
BASIC	**FUNCTION** procedure	**SUB** procedure
C	Function	(**void**) function
Macro Assembler	Procedure	Procedure

For example, a BASIC module can make a **SUB** procedure call to a C function declared with the **void** keyword in place of a return type. BASIC should make a **FUNCTION**-procedure call in order to call a C function that returns a value; otherwise, the call can be made, but the return value is lost.

Note

BASIC **DEF FN** functions and **GOSUB** subroutines cannot be called from another language.

C.2.2 Naming-Convention Requirement

The term "naming convention" refers to the way a compiler alters the name of the routine before placing it into an object file.

It is important that you adopt a compatible naming convention when you issue a mixed-language call. If the name of the called routine is stored differently in each object file, then the linker is unable to find a match and reports an unresolved external.

Microsoft compilers place machine code into object files; but they also place there the names of all routines and variables that need to be accessed publicly. That way, the linker can compare the name of a routine called in one module to the name of a routine defined in another module and recognize a match. Names are stored in ASCII format. You can see precisely how they are stored if you use the **DEBUG** utility to dump an object file's bytes.

The BASIC naming convention translates each letter to uppercase and drops the BASIC type-declaration characters (%, &, !, #, or $). BASIC recognizes the first 40 characters of any name.

C uses a different convention; the C compiler does not translate any letters to uppercase, but it inserts a leading underscore (_) in front of the name of each routine. C recognizes only the first 31 characters of a name. If a name is longer than the language recognizes, additional characters are simply not placed in the object file. Also, when the mixed-language keyword **CDECL** is specified in the BASIC **DECLARE** statement, periods within a name are converted to underscores by BASIC (in addition to adding the leading underscore).

Differences in naming conventions are taken care of for you automatically by mixed-language keywords, as long as you do not use the **/NOI** linker option (which causes the linker to distinguish between "Prn" and "prn"). With C modules, this means that you must be careful not to rely upon differences between uppercase and lowercase letters in naming your C functions.

Figure C.2 illustrates a complete mixed-language development example, showing how naming conventions enter into the process.

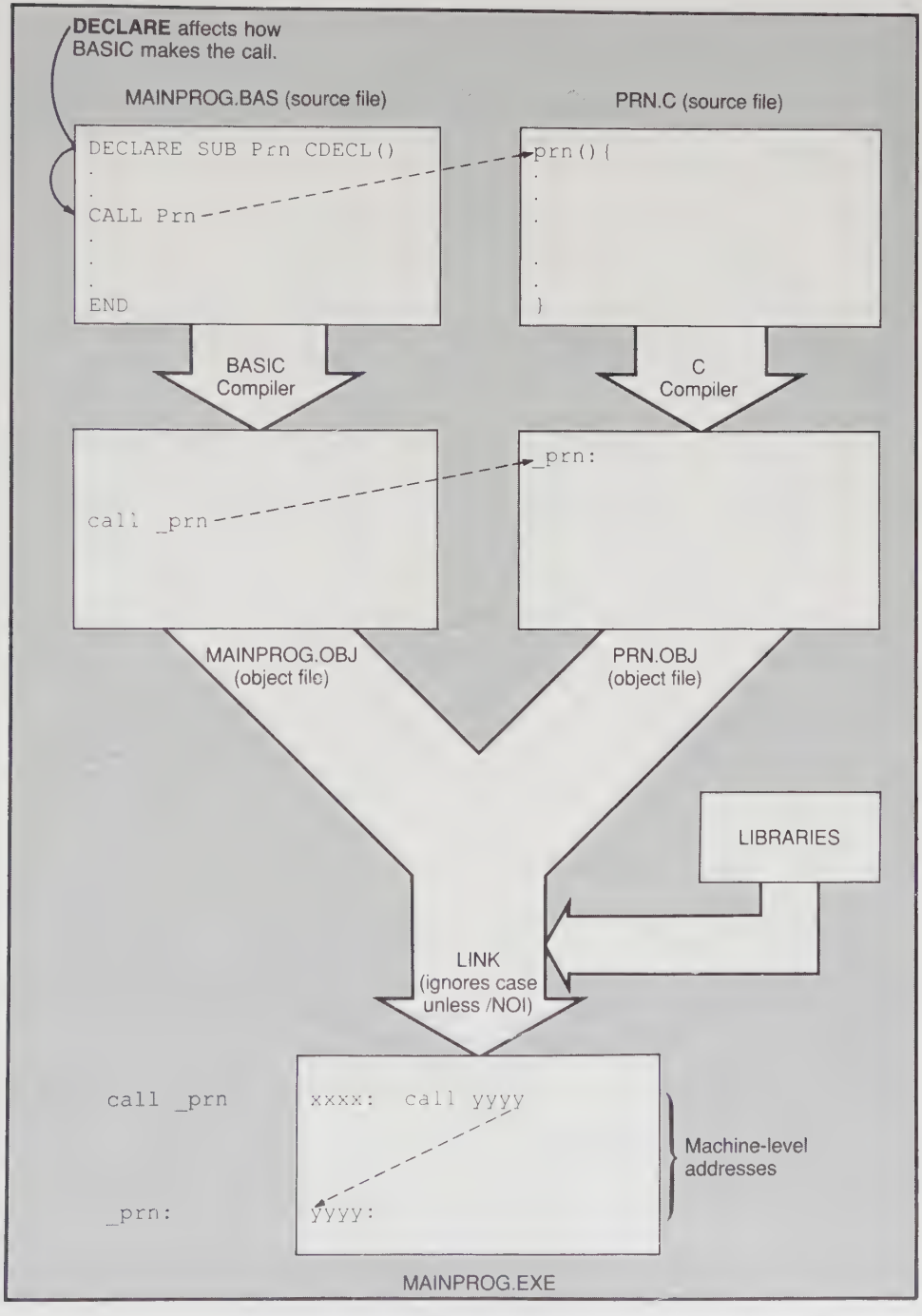


Figure C.2 Naming Convention

In the example above, note that the BASIC compiler inserts a leading underscore in front of `Prn` as it places the name into the object file. It does this because the **CDECL** keyword directs the BASIC compiler to use the C naming convention. BASIC also converts all letters to lowercase when this keyword is used. (Converting letters to lowercase is not part of the C naming convention, but it is consistent with the programming style of most C programs.)

C.2.3 Calling-Convention Requirement

The term "calling convention" refers to the way a language implements a call. The choice of calling convention affects the actual machine instructions that a compiler generates in order to execute (and return from) a function or procedure call.

The calling convention is a low-level protocol. It is crucial that the two routines concerned (the routine issuing a call and the routine being called) recognize the same protocol. Otherwise, the processor may receive inconsistent instructions, thus causing unpredictable behavior.

The use of a calling convention affects programming in two ways:

1. The calling routine uses a calling convention to determine in what order to *pass* arguments (parameters) to another routine. This convention can either be the default for the language, or specified in a mixed-language interface. In the following example, the **CDECL** keyword in the BASIC declaration of the C function overrides the default BASIC convention and causes the parameters to be passed in the order in which a C function normally expects to receive them:

   ```
   DECLARE Func1 CDECL (n%, m%)
   ```

2. The called routine uses a calling convention to determine in what order to *receive* the parameters passed to it. With a C function, this convention can be specified in the function definition.

   In the following example the **fortran** keyword in the function definition overrides the default C convention, and causes the C function to receive the parameters consistent with the default BASIC convention:

   ```
   int fortran func2 (int x, int y)
                  {
                   /* body of function would go here */
                  }
   ```

In other words, the way the function is declared in BASIC causes the BASIC *call* to use a certain calling convention, while the C function *definition* specifies or assumes some calling convention. The two conventions must be compatible. It is simplest to adopt the convention of the

called routine. For example, a C function would use its own convention to call another C function but must use the BASIC convention to call BASIC. This is because BASIC always uses its own convention to receive parameters. Because the BASIC and C calling conventions are different, you may specify a mixed-language interface in the caller or the called routine, but not in both.

C.2.4 Parameter-Passing Requirement

Section C.2.3 discussed the overall protocol (the calling convention) that two routines use to communicate with each other; this section discusses the way an individual piece of data (a parameter) is actually sent.

If your routines do not agree on how a parameter is to be sent, then a called routine will receive incorrect data. It is also possible that the program could cause the system to behave unpredictably.

Microsoft compilers support three methods for passing a parameter:

Method	Description
By near reference	Passes a variable's near (offset) address. This method gives the called routine direct access to the variable itself. Any change the routine makes to the parameter is reflected in the calling routine.
By far reference	Passes a variable's far (segmented) address. This method is similar to passing by near reference, except that a longer address is passed.
By value	Passes only the variable's value, not address. With this method, the called routine knows the value of the parameter but has no access to the original variable. Once the routine terminates, changes to a value parameter have no effect on the value of the parameter in the calling routine.

The fact that there are different parameter-passing methods has two implications for mixed-language programming:

1. You must be sure that the called routine and the calling routine use the same method for passing parameters. Usually, you must check the parameter-passing defaults used by each language, and if necessary make adjustments. Each language has keywords or language features that allow you to change parameter-passing methods.
2. Program logic may require that you use a particular parameter-passing method rather than using the defaults of any language.

Table C.2 summarizes the parameter-passing defaults for BASIC and C.

Table C.2

Parameter-Passing Defaults

Language	Near Reference	Far Reference	By Value
BASIC	All	---	---
C	Near arrays	Far arrays	Non-arrays

See Section C.3, "BASIC Calls to C," and Section C.4, "Calls to BASIC," for information on overriding the defaults.

C.2.5 Compiling and Linking

After you have written your source files and resolved the issues raised in Sections C.2.2–C.2.4, you are ready to compile individual modules and then link them together.

C.2.5.1 Compiling with Proper Memory Models

With BASIC, no special options are required to compile source files that are part of a mixed-language program.

With Microsoft C, however, not all memory models are compatible with other languages. BASIC uses only far-(segmented) code addresses. Therefore, you must always compile C modules in medium, large, or huge models, because these models also use far code addresses. Compiling in a small or compact model causes the mixed-language program to behave unpredictably, as soon as a call is made to or from C. (This problem can be avoided if you apply the **far** keyword to a C function definition in a small- or compact-model program. This specifies that the function uses a far call and return.)

Note

Microsoft QuickC uses medium model when you choose the Compile command from the Run menu and Obj from the Output Options. However, when compiling from the command line with either the **QCL** or **CL** commands, you must specify the correct memory model; otherwise small model is used and your C object files will not be compatible with your BASIC object files.

The preceding paragraph concerns the size of code addresses. Differences in the size of data addresses can be resolved through compiler options or in the source code. Choice of memory model affects the default data-pointer size in C, although this default can be overridden with **near** and **far**. Choice of memory model in C also affects whether data objects are located in the default data segment. If a data object is not located in the default data segment, it cannot be directly passed by near reference.

C.2.5.2 Linking with Language Libraries

In many cases, linking modules compiled with different languages can be done easily. Any of the following measures ensures that all of the required libraries are linked in the correct order:

- Put all language libraries in the same directory as the source files.
- List directories containing all needed libraries in the **LIB** environment variable.
- Let the linker prompt you for libraries.

In each of the above cases, assuming the BASIC module was specified first on the **link** command line (and the **/NOD** linker option is *not* specified), the linker finds libraries in the order that it requires them. If you enter the libraries on the **link** command line, the BASIC libraries must precede all others, because BASIC requires that its libraries be searched before those of any other language.

Note

When linking BASIC modules with those of other languages, the first module specified must be the BASIC main module This also determines the default order in which libraries are searched.

C.3 BASIC Calls to C

Microsoft BASIC supports calls to routines written in Microsoft C. Section C.3 describes the necessary syntax for calling C. For simplicity in illustrating concepts, only integers are used as parameters in these examples.

Section C.4 describes restrictions on the use of functions from the C standard library. Consult this section if you are using any system or memory-allocation library functions.

See Sections C.5–C.8 for information on how to pass specific kinds of data.

C.3.1 The BASIC Interface to Other Languages

The BASIC **DECLARE** statement provides a flexible and convenient interface to other languages. It is available with Microsoft QuickBASIC, Versions 4.0 and later. Earlier versions of BASIC that do not provide the **DECLARE** statement also do not provide libraries that are compatible with other languages.

See Section C.3.1.1, "The DECLARE Statement," in this manual, and the discussions of the **DECLARE** statement in the *BASIC Language Reference*, and *Programming in BASIC: Selected Topics*, for more information on the syntax of the **DECLARE** statement.

C.3.1.1 The DECLARE Statement

The **DECLARE** statement's syntax differs slightly for **FUNCTION** and **SUB** procedures. For **FUNCTION** procedures, the **DECLARE** statement's syntax is as follows:

DECLARE FUNCTION *name* [[**CDECL**]] [[**ALIAS** "*aliasname*"]][([[*parameterlist*]])]

For **SUB** procedures, use this syntax for the **DECLARE** statement:

DECLARE SUB *name* [[**CDECL**]] [[**ALIAS** "*aliasname*"]][([[*parameterlist*]])]

The *name* field is the name that appears in the BASIC source file for the **SUB** or **FUNCTION** procedure you wish to call. Here are the recommended steps for using the **DECLARE** statement to call other languages:

1. For each distinct interlanguage routine you plan to call, include a **DECLARE** statement at the beginning of any module in which the routine is called. (QuickBASIC cannot automatically generate **DECLARE** statements for other-language routines.)

For example, your program may call the subprogram `Maxparam` five different times, each time with different arguments. However, you need to declare `Maxparam` just once for each module. The **DECLARE** statements must be placed near the beginning of the module, preceding all executable statements. A good way to do this is with an include file.

2. If you are calling a routine defined in a C module, use **CDECL** in the **DECLARE** statement (unless the C routine is defined with the **pascal** or **fortran** keyword).

   CDECL directs BASIC to use the C naming and calling conventions during each subsequent call to *name*.

3. If you are calling a C function with a name containing characters that would be illegal in BASIC (for example, the underscore), you can use the **ALIAS** feature. See Section C.3.1.2. for an explanation of **ALIAS**. If you use the **CDECL** keyword, you can use a period in place of the underscore. BASIC then replaces the period with an underscore.

4. Use the parameter list to determine how each parameter is to be passed. See Section C.3.1.3 for information on how to use a parameter list.

5. Once the routine is properly declared, call it just as you would a BASIC subprogram or function.

The other fields are explained in the following discussion.

C.3.1.2 Using ALIAS

As noted above, the use of the **ALIAS** keyword may be necessary if you want to use an underscore as part of the C identifier. Similarly, though it is not likely to be a problem, C places fewer characters of a name into an object file (31, in addition to the leading underscore) than BASIC, which places up to 40 characters of a name into an object file.

Note

You do *not* need the **ALIAS** feature to remove the type declaration characters (%, &, !, #, $). BASIC automatically removes these characters when it generates object code. Thus, `Fact%` in BASIC matches `fact` in C.

The **ALIAS** keyword directs BASIC to place *aliasname* into the object file, instead of *name*. The BASIC source file still contains calls to *name*. However, these calls are interpreted as if they were actually calls to *aliasname*.

■ Example

In the example below BASIC places the *aliasname* `quad_result`, rather than the name `QuadResult`, into the object code. This avoids the use of a mixed-case identifier for the C function, but provides the same type of recognizability as the BASIC name.

```
DECLARE FUNCTION QuadResult% ALIAS "quad_result" (a, b, c)
```

C.3.1.3 Using the Parameter List

The syntax for *parameterlist* is displayed below, followed by explanations of each field. Note that you can use **BYVAL** or **SEG**, but not both.

■ Syntax

[{**BYVAL** | **SEG**}] *variable* [**AS** *type*] [,[{**BYVAL** | **SEG**}] *variable* [**AS** *type*]]...

Use the **BYVAL** keyword to declare a value parameter. In each subsequent call, the corresponding argument will be passed by value (the default method for C modules).

Note

BASIC provides two ways of "passing by value." The usual method of passing by value is to use an extra set of parentheses, as in

```
CALL Holm((A))
```

This method actually creates a temporary value, whose address is passed. **BYVAL** provides a true method of passing by value, because the value itself is passed, not an address. Only by using **BYVAL** is a BASIC program made compatible with a non-BASIC routine that expects a value parameter.

Use the **SEG** keyword to declare a far reference parameter. In each subsequent call, the far (segmented) address of the corresponding argument will

be passed. See the entry for the **DECLARE** statement and Chapter 2, "Data Types," in the *BASIC Language Reference* for information and cautions on the use of the **SEG** keyword.

You can choose any legal name for *variable*, but only the type associated with the name has any significance to BASIC. As with other variables, the type can be indicated with a type declaration character (%, &, !, #, $) or by implicit declaration.

The **AS** *type* clause overrides the default type declaration of *variable*. The *type* field can be **INTEGER**, **LONG**, **SINGLE**, **DOUBLE**, **STRING**, or a user-defined type. Or it can be **ANY**, which directs BASIC to permit any type of data to be passed as the argument.

■ Examples

In the example below, `Calc2` is declared as a C routine that takes three arguments: the first two are integers passed by value, and the last is a single-precision real number passed by value.

```
DECLARE FUNCTION Calc2! CDECL (BYVAL a%, BYVAL b%, BYVAL c!)
```

This example declares a subprogram `Maxout` that takes an integer passed by far reference and a double-precision real number passed by value.

```
DECLARE SUB Maxout (SEG var1 AS INTEGER, BYVAL var2 AS DOUBLE)
```

C.3.2 Alternative BASIC Interfaces

Though the **DECLARE** statement provides a particularly convenient interface, there are other methods of implementing mixed-language calls.

Instead of modifying the behavior of BASIC with **CDECL**, you can modify the behavior of C by applying the **pascal** or **fortran** keyword to the function definition. (These two keywords are functionally equivalent.) Or you can compile the C module with the **/Gc** option, which specifies that all C functions, calls, and public symbols use the conventions of BASIC/FORTRAN/Pascal.

For example, the following C function uses the BASIC/FORTRAN/Pascal conventions to receive an integer parameter:

```
int pascal fun1(n)
int n;
{
 }
```

You can specify parameter-passing methods even though you omit the **DECLARE** statement or omit the parameter list, or both, as follows:

- You can make the call with the **CALLS** statement. The **CALLS** statement causes each parameter to be passed by far reference.
- You can use the **BYVAL** and **SEG** keywords in the actual parameter list when you make the call.

In the following example, **BYVAL** and **SEG** have the same meaning that they have in a BASIC **DECLARE** statement. When you use **BYVAL** and **SEG** this way, however, you need to be careful because neither the type nor the number of parameters will be checked (as they would be if there were a **DECLARE** statement). Also note that, if you do not use a **DECLARE** statement, you must use either the **fortran** or **pascal** keyword in the C function definition, or compile the C function with the **/Gc** option.

```
CALL Fun2(BYVAL Term1, BYVAL Term2, SEG Sum);
```

C.3.3 BASIC Calls to C

This section applies the steps outlined in Section C.3.1.1 to two example programs. An analysis of programming considerations follows each example.

Note

QuickBASIC provides a BASIC system-level function (**B_OnExit**) that can be called from other-language routines to log a termination procedure that will be called when a BASIC program terminates or is restarted when a Quick library is present. See Section C.9 for a description of this routine.

Calling C from BASIC with No Return Value

The example below demonstrates a BASIC main module calling a C function, maxparam. The function maxparam returns no value, but adjusts the lower of two arguments to equal the higher argument.

■ Example

```
' BASIC source file - calls C function returning no value
'
DECLARE SUB Maxparam CDECL (A AS INTEGER, B AS INTEGER)
'
' DECLARE as subprogram, since there is no return value
' CDECL keyword causes Maxparam call to be made with C
' conventions. Integer parameters passed by near reference
' (BASIC default).
'
X% = 5
Y% = 7
PRINT USING "X% = ##  Y% = ##";X% ;Y%    ' X% and Y% before call
CALL Maxparam(X%, Y%)                     ' Call C function
PRINT USING "X% = ##  Y% = ##";X% ;Y%    ' X% and Y% after call
END
```

```
/* C source file */
/* Compile in MEDIUM or LARGE memory model */
/* Maxparam declared VOID because no return value */

void maxparam(p1, p2)
int near *p1;     /* Integer params received by near ref. */
int near *p2;     /* NEAR keyword not needed in MEDIUM model. */
{
        if (*p1 > *p2)
                *p2 = *p1;
        else
                *p1 = *p2;
}
```

You should keep the following programming considerations in mind when calling C from BASIC with no return value:

- **Naming Conventions**

  The **CDECL** keyword causes Maxparam to be called with the C naming convention (as _maxparam).

- **Calling Conventions**

  The **CDECL** keyword causes Maxparam to be called with the C calling convention, which pushes parameters in the reverse order to the way they appear in the source code.

- **Parameter-Passing Methods**

  Since the C function `maxparam` may alter the value of either parameter, both parameters must be passed by reference. In this case, near reference was chosen; this method is the default for BASIC (so neither **BYVAL** nor **SEG** is used) and is specified in C by using near pointers.

  Far reference could have been specified by applying **SEG** to each argument in the **DECLARE** statement. In that case, the C parameter declarations would use far pointers.

Calling C from BASIC with a Function Call

The example below demonstrates a BASIC main module calling a C function, `fact`. This function returns the factorial of an integer value.

■ Example

```
' BASIC source file - calls C function with return value
'
DECLARE FUNCTION Fact% CDECL (BYVAL N AS INTEGER)
'
' DECLARE as function returning integer (%)
' CDECL keyword causes Fact% call to be made with
' C conventions. Integer parameter passed by value.
'
X% = 3
Y% = 4
PRINT USING "The factorial of X%    is ####"; Fact%(X%)
PRINT USING "The factorial of Y%    is ####"; Fact%(Y%)
PRINT USING "The factorial of X%+Y% is ####"; Fact%(X%+Y%)
END
```

```
/* C source file */
/* Compile in MEDIUM or LARGE model */
/* Factorial function, returning integer */

int fact(n)
int n;          /* Integer passed by value, the C default */
{
        int     result = 1;

        while (n > 0)
                result *= n--;  /* Parameter n modified here */
        return(result);
}
```

You should keep the following programming considerations in mind when calling C from BASIC with a function call:

- **Naming Conventions**

  The **CDECL** keyword causes `Fact` to be called with the C naming convention (as `_fact`).

- **Calling Conventions**

  The **CDECL** keyword causes `fact` to be called with the C calling convention, which pushes parameters in reverse order and specifies other low-level differences.

- **Parameter-Passing Methods**

  The C function above should receive the parameter by value. Otherwise the function will corrupt the parameter's value in the calling module. True passing by value is achieved in BASIC only by applying **BYVAL** to the parameter in the **DECLARE** statement; in C, passing by value is the default (except for arrays).

C.3.4 Restrictions on Calls from BASIC

BASIC has a much more complex environment and initialization procedure than C. Interlanguage calling between BASIC and other languages is possible only because BASIC intercepts a number of library function calls from the other language and handles them in its own way. In other words, BASIC creates a host environment in which the C routines can function.

However, BASIC is limited in its ability to handle some C function calls. This section considers two kinds of limitations: C memory-allocation functions, which may require a special declaration, and a few specific C-library functions, which cannot be called at all.

C.3.4.1 Memory Allocation

If your C module is medium model and you do dynamic memory allocation with **malloc()**, or if you execute explicit calls to **_nmalloc()** with any memory model, then you need to include the following lines in your BASIC source code before you call C:

```
DIM mallocbuf%(2048)
COMMON SHARED /NMALLOC/ mallocbuf%()
```

The array can have any name; only the size of the array is significant. However, the name of the common block must be **NMALLOC**. In the QuickBASIC 4.0 in-memory environment, you need to put this declaration in a module that you incorporate into a Quick library. See Chapter 8, "Quick Libraries" and Chapter 9, "Compiling and Linking from DOS," for more information on Quick libraries.

The example above has the effect of reserving 4K of space in the common block **NMALLOC**. When BASIC intercepts C **malloc** calls, BASIC allocates space out of this common block.

Warning

When you call the BASIC intrinsic function **CLEAR**, all space allocated with near **malloc** calls is lost. If you use **CLEAR** at all, use it only before any calls to **malloc**.

When you make far-memory requests in mixed-language programs, you may find it useful to call the BASIC intrinsic function **SETMEM** first. This function can be used to reduce the amount of memory BASIC is using, thus freeing up memory for far allocations.

C.3.4.2 Incompatible Functions

The following C functions are incompatible with BASIC and should be avoided:

- All forms of **spawn()** and **exec()**
- **system()**
- **getenv()**
- **putenv()**

Calling these functions results in the BASIC error message

```
Advanced feature unavailable
```

In addition, you should not link with the *x***VARSTK.OBJ** modules (where *x* is a memory model), which C provides to allocate memory from the stack.

Note

The global C run-time variables **environ** and **_pgmptr** are defined as NULL. All functionality of these variables, as well as the functions noted above, can be emulated using BASIC statements and intrinsic functions.

C.3.4.3 Allocating String Space

Other-language routines can allocate dynamic string space by calling the following BASIC **FUNCTION**:

```
FUNCTION GetSpace$ (x) STATIC
GetSpace$ = STRING$(x, CHR$(0))
END FUNCTION
```

The GetSpace procedure returns a near pointer to a string descriptor that points to x bytes of string space. Because this space is managed by BASIC, it can move any time BASIC language code is executed. Therefore the space must be accessed through the string descriptor, and the string descriptor must not be modified by other-language code. To release this space, pass the near pointer to the string descriptor to the following BASIC **SUB** procedure:

```
SUB FreeSpace(a$) STATIC
a$ = ""
END SUB
```

To return a string that is the result of a routine in another language, return a near pointer to a static string descriptor that is declared in the other-language code. Because BASIC moves such strings around, the static string descriptor allocated by the other-language code becomes invalid after the function returns (or makes a call to any BASIC procedure).

C.3.4.4 Performing I/O on BASIC Files

Other-language routines can perform input and output on files opened by the BASIC **OPEN** statement by calling BASIC procedures. The following example is a BASIC **SUB** that can be called to print an integer to a BASIC file opened as `fileno`:

```
SUB DoPrint(fileno%, x%) STATIC
PRINT #fileno%, x%
END SUB
```

C.3.4.5 Events and Errors

BASIC events including **COM**, keyboard, **TIMER**, and **PLAY** may occur during execution of other-language code. The other-language code can allow such events to be handled as follows:

- When compiling with the **BC** command, compile the BASIC procedure with event-handling options selected (for instance, **/v** or **/w**).
- Within the QuickBASIC environment, simply specify event-handling syntax in the procedure itself, then use the Run menu's Make EXE File or Make Library command to create an executable file or to incorporate the procedure into a Quick library.

The following BASIC **SUB** lets you declare BASIC errors:

```
SUB MakeError(x%) STATIC
ERROR x%
END SUB
```

When your other-language routine passes the error number to this procedure, the **ERROR** statement is executed and BASIC recovers the stack back to the previous call to non-BASIC code. The BASIC statement containing the call to the non-BASIC code is the statement that **RESUME** would reexecute.

C.4 C Calls to BASIC

Microsoft C can call routines written in Microsoft BASIC, if the main program is in BASIC. Sections C.4.1–C.4.2 describe the necessary syntax for calling BASIC from C. Only simple parameter lists are used.

See Sections C.5–C.8 for information on how to pass particular kinds of data.

C.4.1 The C Interface to Other Languages

The C interface to other languages uses the standard C **extern** statement, combined with the special **fortran** or **pascal** keyword. Using either of these keywords causes the routine to be called with the naming and calling conventions of FORTRAN/Pascal/BASIC. Here are the recommended steps for using this statement to execute a mixed-language call from C:

1. Write an **extern** declaration for each mixed-language routine called.

   The **extern** statement should precede all calls to the routine. The exact rules of syntax for using the **fortran** and **pascal** keywords with the **extern** statement are presented below.

   Instead of using the **fortran** or **pascal** keyword, you can simply compile with **/Gc**. The **/Gc** option causes all functions in the module to use the BASIC/FORTRAN/Pascal naming and calling conventions. If you want most (but not all) of the routines in the module to use the BASIC/FORTRAN/Pascal conventions, you can compile the module with **Gc**, but use the **cdecl** keyword in the definitions of those functions for which you want to override the effect of the **Gc** switch. Note that when you use **cdecl** in a C function, it must be typed in lowercase letters.

2. Use parameter type declarations within the **extern** statement.

   This step is essential if you are going to specify pointers that are not the default size. (Near pointers are the default for medium model; far pointers are the default for large model.)

3. To pass an argument by reference, pass a pointer to the object.

   C automatically translates array names into addresses. Therefore, arrays are automatically passed by reference.

4. Once a routine has been properly declared with an **extern** statement, call it just as you would call a C function.

5. Always compile the C module in medium or large model.

Using the fortran and pascal Keywords

There are two rules of syntax that apply when you use the **fortran** or **pascal** keyword:

1. The **fortran** and **pascal** keywords modify the items immediately to their right.
2. The special **near** and **far** keywords can be used with the **fortran** and **pascal** keywords in declarations. The sequences **fortran far** and **far fortran** are equivalent.

The keywords **pascal** and **fortran** actually have the same effect on the program; the use of one or the other makes no difference except for documentation purposes. Use either keyword to declare a BASIC routine.

■ Examples

The examples below demonstrate the syntax rules presented above for the **fortran** and **pascal** keywords.

The following example declares `thing` to be a BASIC, Pascal, or FORTRAN function taking two **short** parameters and returning a **short** value.

```
extern short pascal thing(short, short);      /* Example 1 */
```

The next example declares `thing` to be a pointer to a BASIC, Pascal, or FORTRAN routine that takes a **long** parameter and returns no value. The keyword **void** is appropriate when the called routine is a BASIC **SUB** procedure, since it indicates that no return value is expected.

```
extern void (fortran *thing)(long);           /* Example 2 */
```

The next example declares `thing` to be a **near** BASIC, Pascal, or FORTRAN routine. The routine receives a **double** parameter by reference (because it expects a pointer to a **double**) and returns a **short** value.

```
extern short near pascal thing(double *);       /* Example 3 */
```

The following example is equivalent to the previous one (`pascal near` is equivalent to `near pascal`).

```
extern short pascal near thing(double *);       /* Example 4 */
```

C.4.2 Calls from C to BASIC

When you call BASIC, you must use the BASIC/FORTRAN/Pascal conventions to make the call. No BASIC routine can be executed unless the main module of the program is in BASIC, because a BASIC routine requires the environment to be initialized in a way that is unique to BASIC. No other language performs this special initialization.

However, it is possible for a program to start up in BASIC, call a C function that does most of the work of the program, and then call BASIC **SUB** and **FUNCTION** procedures as needed. Figure C.3 illustrates how this can be done.

Note

All other-language-to-BASIC calling for programs within the QuickBASIC environment must be confined within a Quick library. In other words, a C function can call a BASIC procedure within the Quick library, but it cannot call a procedure defined within the QuickBASIC environment itself.

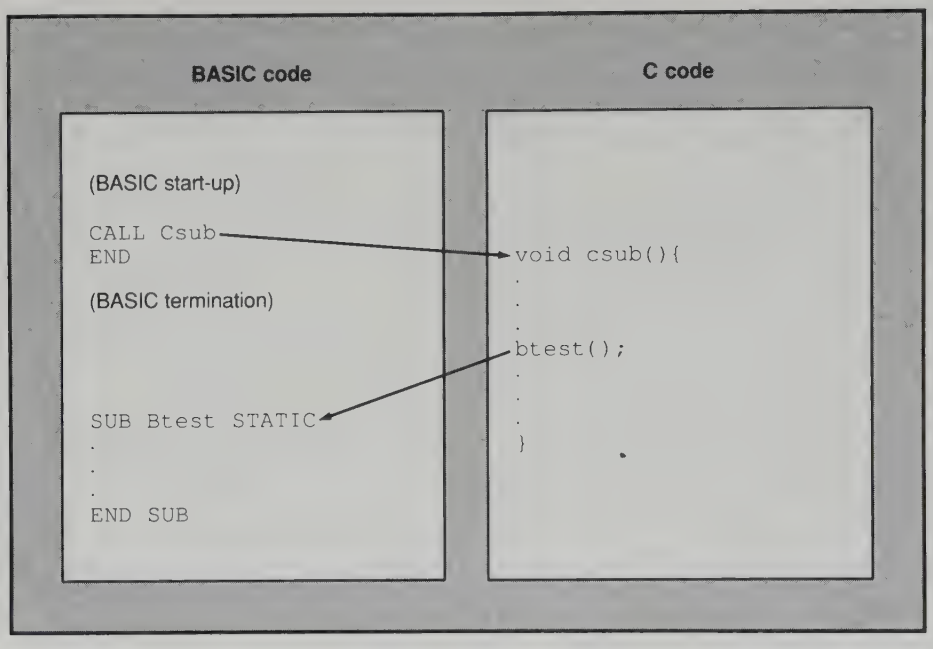


Figure C.3 C Call to BASIC

The following rules are recommended when you call BASIC from C:

1. Start in a BASIC main module. You need to use QuickBASIC Version 4.0 or later. You also need to use the **DECLARE** statement to provide an interface to the C module. (See Section C.3, "BASIC Calls to C," for more information.)

2. Once in the C module, declare the BASIC routine as **extern**, and include type information for parameters. Use either the **fortran** or **pascal** keyword in the C declaration of the BASIC procedure to override the default C calling convention.

3. Make sure that all data are passed as near pointers. BASIC can *pass* data in a variety of ways, but it is unable to receive data in any form other than near reference.

   With near pointers, the program assumes that the data are in the default data segment. If you want to pass data that are *not* in the default data segment, then first copy the data to a variable that is in the default data segment (this is only a consideration with large-model programs).

4. Compile the C module in medium or large model.

■ Example

The example below demonstrates a BASIC program that calls a C function. The C function then calls a BASIC function that returns twice the number passed it and a BASIC subprogram that prints two numbers.

```
' BASIC source
'
DEFINT A-Z
DECLARE SUB Cprog CDECL()
CALL Cprog
END
' This is the BASIC FUNCTION called in cprog().
FUNCTION Dbl(N) STATIC
    Dbl = N*2
END FUNCTION
' This is the BASIC SUB called in cprog().
SUB Printnum(A,B) STATIC
    PRINT "The first number is ";A
    PRINT "The second number is ";B
END SUB
```

```
/* C source; compile in medium or large model */

extern int fortran dbl(int near *);
extern void fortran printnum(int near *, int near *);

void cprog()
{
        int near a = 5;   /* NEAR guarantees that the data  */
        int near b = 6;   /*   will be placed in default    */
                          /*   data segment (DGROUP)        */

        printf("Twice of 5 is %d\n", dbl(&a));
        printnum(&a, &b);
}
```

In the example above, note that the *addresses* of `a` and `b` are passed, since BASIC expects to receive addresses for parameters. Also note that the keyword **near** is used to declare each pointer in the C function declaration of `printnum`; this keyword would be unnecessary if it was known that the C module was compiled in medium model rather than large.

Calling and naming conventions are resolved by the **CDECL** keyword in BASIC's declaration of `Cprog`, and by **fortran** in C's declaration of `dbl` and `printnum`.

C.5 Passing Data by Reference or Value

Section C.3.1.3 introduced the general concepts of passing by reference and passing by value. It also noted that, by default, BASIC passes by reference, and C passes by value.

This section describes features in each language that override the default. For example, using the **BYVAL** keyword in a **DECLARE** statement causes BASIC to pass a given parameter by value rather than by reference.

Section C.5.1 summarizes parameter-passing methods for BASIC, discussing how to pass arguments by value, by near reference, and by far reference. Section C.5.2 discusses the same issues for C. To write a successful mixed-language interface, you must consider how each parameter is passed by the calling routine and how each is received by the called routine.

C.5.1 BASIC Arguments

The default for BASIC is to pass all arguments by near reference.

Note

Every BASIC **SUB** or **FUNCTION** procedure always *receives* parameters by near reference. The rest of this section describes only how BASIC *passes* parameters.

Passing BASIC Arguments by Value

An argument is passed by value when the called routine is first declared with a **DECLARE** statement, and the **BYVAL** keyword is applied to the argument. However, arrays and user-defined types cannot be passed by value in BASIC.

Passing BASIC Arguments by Near Reference

The BASIC default is to pass by near reference. Use of **SEG**, **BYVAL**, or **CALLS** changes this default.

Passing BASIC Arguments by Far Reference

BASIC passes each argument in a call by far reference when **CALLS** is used to invoke a routine. Using **SEG** to modify a parameter in a preceding **DECLARE** statement will also cause BASIC to pass that parameter by far reference.

■ Examples

The example below passes the first argument (a%) by value, the second argument (b%) by near reference, and the third argument (c%) by far reference.

```
DECLARE SUB Test(BYVAL a%, b%, SEG c%)
.
.
.
CALL Test(x%, y%, z%)
```

The example below passes each argument by far reference.

```
CALLS Test2(x%, y%, z%)
```

C.5.2 C Arguments

The default for C is to pass all arrays by reference (near or far, depending on the memory model) and all other data types by value. C uses far data pointers for compact, large, and huge models, and near data pointers for small and medium models.

Passing C Arguments by Value

The C default is to pass all nonarrays (which includes all data types other than those explicitly declared as arrays) by value.

Passing C Arguments by Reference (Near or Far)

In C, passing a pointer to an object is equivalent to passing the object itself by reference. After control is passed to the called function, each reference to the parameter itself is prefixed by an asterisk (*).

Note

To pass a pointer to a object, prefix the parameter in the call statement with an ampersand (&). To receive a pointer to an object, prefix the parameter's declaration with an asterisk (*). In the latter case, this may mean adding a second asterisk (*) to a parameter which already has an asterisk (*). For example, to receive a pointer by value, declare it as

```
int         *ptr;
```

but to receive the same pointer by reference, declare it as

```
int         **ptr;
```

The default for arrays is to pass by reference.

Effect of Memory Models on Size of Reference

In C, near reference is the default for passing pointers in small and medium models. Far reference is the default in the compact, large, and huge models.

Near pointers can be specified with the **near** keyword, which overrides the default pointer size. However, if you are going to override the default pointer size of a parameter, then you must explicitly declare the parameter type in function declarations as well as function definitions.

Far pointers can be specified with the **far** keyword, which overrides the default pointer size.

C.6 Numeric and String Data in Mixed-Language Programming

This section discusses passing and receiving different kinds of data. Discussion includes the differences in string format and methods of passing strings between BASIC and C.

C.6.1 Integer and Real Numbers

Integer and real numbers are usually the simplest kinds of data to pass between languages. However, the type of numerical data is named differently in each language; furthermore, not all data types are available in every language, and another type may have to be substituted in some cases.

Table C.3 shows equivalent data types in BASIC and C.

Table C.3
Equivalent Numeric Data Types

BASIC	C
x%	**short**
INTEGER	**int**
---	**unsigned short***
---	**unsigned**
x&	**long**
LONG	**long**
---	**unsigned long***
x!	**float**†
x (default)	**float**†
SINGLE	**float**†
x#	**double**
DOUBLE	**double**
---	**unsigned char**‡

* Not available in BASIC. A signed integer may be substituted, but take care not to exceed range.

† C automatically converts **float** to **double** in assignment or when passed by value.

‡ Not available in BASIC. C automatically converts **char** and **unsigned char** to **int** in assignment or when passed by value.

Warning

As noted in Table C.3, C sometimes performs automatic data conversions which other languages do not perform. You can prevent C from performing such conversions by declaring a variable as the only member of a structure and then passing this structure. For example, you can pass a variable *x* of type **float** by first declaring the structure

```
struct {
          float   x;
} x_struct;
```

If you pass a variable of type **char** or **float** by value and do not take this precaution, then the C conversion may cause the program to fail.

C.6.2 Strings

Strings are stored in a variety of formats. Therefore, some transformation is frequently required to pass strings between languages.

This section presents the string format(s) used in each language, and then describes methods for passing strings within specific combinations of languages.

C.6.2.1 String Formats

The following section describes how a string is stored by each language, as well as how a string is passed as an argument.

BASIC String Format

Most strings are stored in BASIC as four-byte string descriptors, as shown in Figure C.4.

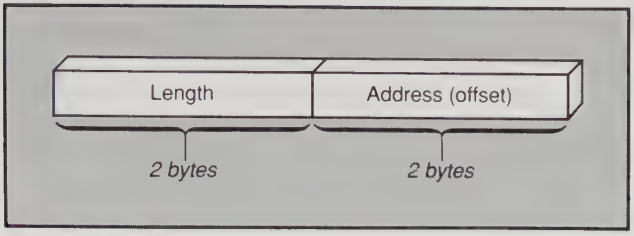


Figure C.4 BASIC String-Descriptor Format

The first field of the string descriptor contains a two-byte integer indicating the length of the actual string text. The second field contains the address of this text. This address is an offset into the default data area and is assigned by BASIC's string-space management routines. These management routines need to be available to reassign this address whenever the length of the string changes or memory is compacted, yet these management routines are only available to BASIC. Therefore, other languages should not alter the length, or the address, of a BASIC string.

C String Format

C stores strings as simple arrays of bytes and uses a terminating null character (numerical 0, ASCII NUL) as a delimiter. For example, consider the string declared as follows:

```
char str[] = "String of text"
```

The string is stored in 15 bytes of memory as shown in Figure C.5.

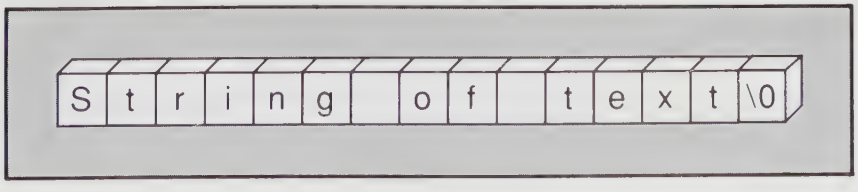


Figure C.5 C String Format

Since `str` is an array like any other, it is passed by reference, just as other C arrays are.

C.6.2.2 Passing BASIC Strings

When a BASIC string (such as `A$`) appears in an argument list, BASIC passes the address of a string descriptor rather than the actual string data. The BASIC string descriptor is not compatible with the string formats of other languages.

Warning

When you pass a string from BASIC to another language, the called routine should under no circumstances alter the length or address of the string. Other languages lack BASIC's string-space management routines. Therefore, altering the length of a BASIC string is liable to corrupt parts of the BASIC string space. Changes that do not affect length and address, however, are relatively safe.

If the routine that receives the string calls any BASIC routine, the address of the string data may move. The second field of the string descriptor is then updated with the new address of the string text.

However, the **SADD** and **LEN** functions extract parts of the string descriptor. **SADD** extracts the address of the actual string data, and **LEN** extracts the length. The results of these functions can then be passed to other languages.

BASIC should pass the result of the **SADD** function by value. Bear in mind that the string's *address*, not the string itself, is passed by value. This amounts to passing the string itself by reference. The BASIC module passes the string address, and the other module receives the string address. The address returned by **SADD** is declared as type integer, but is actually equivalent to a C near pointer.

Pass `LEN(A$)` as you would normally pass a two-byte integer. Use the value returned by **SADD** immediately because several BASIC operations may cause movement of strings in memory. Note that **SADD** cannot be used with fixed-length strings. See Chapter 2, "Data Types," in the *BASIC Language Reference* for more information.

C.6.2.3 Passing BASIC Strings to C

Before attempting to pass a BASIC string to C, you may first want to append a null byte on the end with an instruction such as

```
A$ = A$ + CHR$(0)
```

The string now conforms to the C string format. Note that when used in a BASIC string expression the + indicates concatenation, or joining, of two strings.

You can use either of the following two methods for passing a string from BASIC to C:

1. Pass the string address and string length as separate arguments, using the **SADD** and **LEN** functions. If you are linking to a C run-time library routine, this is the only workable method.

   In the following example, SADD(A$) returns the near address of the string data. This address must be passed by value, since it is equivalent to a pointer (even though it is treated by BASIC as an integer). Passing by reference would attempt to pass the *address* of the address, rather than the address itself.

   ```
   DECLARE SUB Test CDECL(BYVAL S%, BYVAL N%)
   CALL Test(SADD(A$), LEN(A$))
   .
   .
   .
   void Test(s, n)
   char near *s;
   int  n;
   {
    /* body of function */
    }
   ```

   C must receive a near pointer since only the near (offset) address is being passed by BASIC. Near pointers are the default pointer size in medium-model C.

2. Pass the string descriptor itself, with a call statement such as

   ```
   CALL Test2(A$)
   ```

   In this case, the C function must declare a structure for the parameter that has the appropriate fields (address and length) for a BASIC string descriptor. The C function should then expect to receive a pointer to a structure of this type. Such a structure declaration and function definition might look as follows:

   ```
   struct bas_str{          /* structure declaration */
                    char near *sd_addr ;
                    int        sd_len  ;
                } ;

   test (basic_string)       /* function definition */
        struct bas_str *basic_string ;
   {
   /* body of function would go here */
    }
   ```

   Note that calls back to BASIC from within the C function above may invalidate the string-descriptor information.

C.6.2.4 Passing C Strings

When a C string appears in an argument list, C passes the address of the string. (A C string is just an array and so is passed by reference.) C can pass data to BASIC in the form of a string descriptor.

Passing C Strings to BASIC

To pass a C string to BASIC, first allocate a string in C; then, create a structure identical to a BASIC string descriptor. Pass this structure by near reference, as in the example below:

```
char cstr[] = "ABC";
struct {
        char    *sd_addr;
        int     sd_len;
} str_des;
str_des.sd_addr = cstr;
str_des.sd_len = strlen(cstr);
bsub(&str_des);
```

Make sure that the string originates in C, not in BASIC. Otherwise, BASIC may attempt to move the string around in memory.

C.7 Special Data Types

This section considers special types of data that are either arrays or structured types (that is, data types that contain more than one field).

C.7.1 Arrays

When you program in only one language, arrays do not present special problems; the language is consistent in its handling of arrays. When you program with more than one language, however, you need to be aware of two special problems that may arise with arrays:

1. Arrays are implemented differently in BASIC than in other Microsoft languages, so that you must take special precautions when you pass an array from BASIC to another language (including assembly).
2. Arrays are declared and indexed differently in each language.

This section considers each of these problems in turn.

C.7.1.1 Passing Arrays from BASIC

Most Microsoft languages permit you to reference arrays directly. In C, for example, an array name is equivalent to the address of the first element. This simple implementation is possible because the location of data for an array never changes.

BASIC uses an *array descriptor*, however, which is similar in some respects to a BASIC string descriptor. The array descriptor is necessary because BASIC may shift the location of array data in memory, as BASIC handles memory allocation for arrays dynamically. See Chapter 2, "Data Types," in the *BASIC Language Reference*, for specific information and cautions about passing BASIC arrays.

C has no equivalent of the BASIC array descriptor. More importantly, it lacks access to BASIC's space-management routines for arrays. Therefore, you may safely pass arrays from BASIC *only* if you follow three rules:

1. Because almost all arrays (with the exception of variable-length strings) represent far data, **VARSEG** should always be used in passing them. To pass the far address of an array, apply both the **VARPTR** and **VARSEG** functions and pass each result by value. The receiving language gets the address of the first element and considers it to be the address of the entire array. It can then access the array with its normal array-indexing syntax.

   Alternatively, a far reference to an array can be passed by passing its first element by far reference as in the following examples:

   ```
   CALLS Proc1 (A(0,0))

   CALL Proc2 (SEG A(0,0))
   ```

2. The routine that receives the array must not, under any circumstances, make a call back to BASIC. If it does, then the location of the array data may change, and the address that was passed to the routine becomes meaningless.

3. BASIC may pass any member of an array by value. With this method, the above precautions do not apply.

Note

You can apply the **LBOUND** and **UBOUND** functions to a BASIC array, to determine lower and upper bounds, and then pass the results to another routine. This way, the size of the array does not need to be determined in advance. See the *BASIC Language Reference* and *Programming in BASIC: Selected Topics* for more information on the **LBOUND** and **UBOUND** functions.

C.7.1.2 Array Indexing and Declaration

Each language varies somewhat in the way that arrays are declared and indexed.

Indexing Arrays

Array indexing is purely a source-level consideration and involves no transformation of data. There are two differences in the way that elements are indexed by different languages:

1. Lower bounds.

   BASIC and C index the first element of an array as 0. BASIC gives the user the option of specifying lower bounds at any integer values.

2. Row-major order vs. column-major order.

   This issue only affects arrays with more than one dimension. With row-major order (used by C) the column dimension (represented by the rightmost subscript), changes the fastest. With column-major order (used by BASIC), the row dimension (represented by the leftmost subscript), changes the fastest. Thus, in C the first four elements of an array whose final element is `X [2] [2]`, are:

   ```
   X[0] [0]    X[0] [1]    X[0] [2]    X[1] [0]
   ```

   In BASIC, the corresponding four elements are:

   ```
   X(0,0)    X(1,0)    X(2,0)    X(0,1)
   ```

   Similarly, the following references both refer to the same place in memory for an array:

   ```
   arr1[2][8]          /*   in C   */

   Arr1(8,2)           ' in BASIC
   ```

The examples above assume that both the C and BASIC arrays use lower bounds of 0.

Declaring Arrays

Table C.4 shows equivalences for array declarations in each language. In this table, *r* is the number of elements of the row dimension, and *c* is the number of elements of the column dimension.

Table C.4
Equivalent Array Declarations

Language	Array Declaration	Notes
BASIC	**DIM** *x*(*r*–*1*,*c*–*1*)	With default lower bounds of 0
C	*type* *x*[*r*][*c*]	When passed by reference

Describing arrays in terms of rows and columns is a convenient analogy for understanding 2-dimensional arrays. However, the format of the declarations shown in Table C.4 can be extended to any number of dimensions that you may use. For example, the C declaration

```
int     arr1[2][10][15][20] ;
```

is equivalent to the BASIC declaration

```
DIM     Arr1%(19, 14, 9, 1)
```

These are equivalent in the sense that the C array element represented by `arr1[i][j][k][l]` references the same memory location as the BASIC array element `Arr1(l,k,j,i)`

Note

In BASIC, the constants that are used in the array declaration `DIM Arr%(5,5)` represent the upper bounds of the array. Therefore, the last element is indexed as `Arr%(5,5)`. The constants used in a C array declaration represent the actual number of elements in each dimension, not upper bounds as they do in BASIC. Therefore, the last element in the C array declared as `int arr[5][5]` is indexed as `arr[4][4]`, rather than as `arr[5][5]`.

When you compile a BASIC program with the **bc** command, you can select the **/r** compile option, which specifies that row-major order is to be used, rather than column-major order. Within the QuickBASIC environment, this option is not available.

C.7.2 Storage of Structures and User-Defined Types

The C **struct** type and the BASIC user-defined type are equivalent. Therefore, these data types can be passed between C and BASIC.

However, these types may be affected by the storage method. By default, C uses word alignment (unpacked storage) for all data except byte-sized objects and arrays of byte-sized objects. This storage method specifies that occasional bytes may be added as padding, so that word and double-word objects start on an even boundary. (In addition, all nested structures and records start on a word boundary.) Figure C.6 shows the contrast between packed and word-aligned storage:

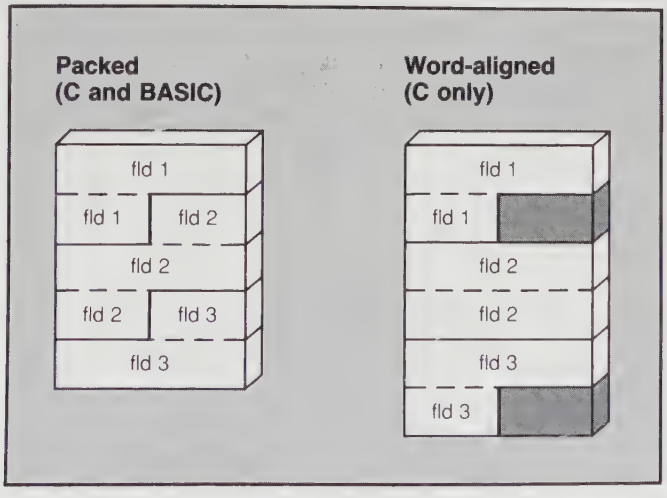


Figure C.6 Structure and Record Storage

BASIC packs structures. Therefore, if a structure is being passed between BASIC and C, it is important that a calling routine and the called routine agree on storage method. Otherwise, data are not properly received. The simplest method for ensuring compatibility between languages is to turn on packing for C modules. The packed storage method may sacrifice some speed, but it has the advantage of creating smaller executable files.

C.8 Pointers, Address Variables, and Common Blocks

Rather than pass data directly, you may want to pass the *address* of a piece of data. Passing the address amounts to passing the data itself by reference. In some cases, such as BASIC arrays (see Section C.7.1.1), passing an address is the only way to share particular kinds of data between two languages.

BASIC does not have a formal address type. However, BASIC provides ways for storing and passing addresses.

BASIC programs can access a variable's segment address with **VARSEG** and its offset address with **VARPTR**. The values returned by these intrinsic functions should then be passed or stored as ordinary integer variables. If you pass them to another language, pass by value. Otherwise you will be attempting to pass the *address* of the address, rather than the address itself. Use the values returned by **VARSEG** and **VARPTR** immediately, because BASIC operations may cause movement of variables in memory. See Chapter 2, "Data Types," in the *BASIC Language Reference* for more information.

To pass a near address, pass only the offset; if you need to pass a far address, you may need to pass the segment and offset separately. Pass the segment address first, unless **CDECL** is in effect.

C.8.1 Common Blocks

You can pass individual members of a BASIC common block in an argument list, just as you can with any data. However, you can also give a different language module access to the entire common block at once.

C modules can reference the items of a common block by first declaring a structure or record with fields that correspond to the common-block variables.

Having defined a structure or user-defined type with the appropriate fields, the C module must then connect with the common block itself.

To pass the address of a common block, simply pass the address of the first variable in the block. (In other words, pass the first variable by reference.) The receiving C module should expect to receive a structure by reference.

■ Example

In the example below, the C function `initcb` receives the address of the variable `N`, which it considers to be a pointer to a structure with three fields:

```
'    BASIC SOURCE CODE
'
         COMMON /Cblock/ n%,x#,y#
         DECLARE SUB INITCB (n%)
.
.
.
         CALL INITCB(n%)
.
.
.
/* C source code */

struct block_type {
         int     n;
         double  x;
         double  y;
};

void initcb(block_hed)
struct block_type *block_hed;
{
         block_hed->n = 1;
         block_hed->x = 10.0;
         block_hed->y = 20.0;
}
```

C.8.2 Using a Varying Number of Parameters

Some C functions, most notably **printf**, can be called with a different number of arguments each time. To call such a function from another language, you need to suppress the type-checking that normally forces a call to be made with a fixed number of parameters. In BASIC, you can remove this type checking by omitting the parameter list from the **DECLARE** statement, as noted in Section C.3.2, "Alternative BASIC Interfaces."

Because the number of parameters is not fixed, the routine you call should have some mechanism for determining how many parameters to expect. Often this information is indicated by the first parameter. For example, the C function **printf** scans the format string passed as the first parameter. The number of fields in the format string determines how many additional parameters the function should expect. The following example illustrates two ways of calling the C-library function **printf**.

■ Examples

In the first example a fixed number of arguments is declared, and the keyword **BYVAL** precedes each parameter in the **DECLARE** statement to insure that the true address of the string argument is passed.

```
DEFINT A-Z

DECLARE SUB printf CDECL (BYVAL format, BYVAL value)

String1$ = "Value passed to the formatted string is %d"+CHR$(0)
Number = 19

CALL printf(SADD(String1$), Number)

END
```

The following variation of the preceding example uses a special form of the **DECLARE** statement to declare the `printf` function. The parentheses that normally contain the formal parameters are omitted from the declaration. This informs QuickBASIC of two facts:

1. The procedure is not written in BASIC.
2. The procedure can have a variable number of arguments. The **BYVAL** keyword is still used in the program, but this time it is passed with each argument in the `printf` call.

In both examples, the C-language format character %d is used in the string passed to `printf`, which replaces it with the value of the second argument.

```
DEFINT A-Z

DECLARE SUB printf CDECL

String1$ = "Value passed to the formatted string is %d"+CHR$(0)
Number = 19

CALL printf(BYVAL SADD(String1$), BYVAL Number)

END
```

The example above describes the BASIC call to **printf**. When you link the object file created by such a program from the command line, the linker resolves the reference to the **printf** function when the executable file is created. However, if you want to call **printf** from within the QuickBASIC environment, you must make the function available within a Quick library. The simplest way to do this is to write a "dummy" C function that calls **printf**, and compile it (make sure you are compiling in medium model, as noted previously). Then, instead of linking it with a BASIC program on the **link** command line, incorporate the object file into a Quick library as shown in the following example:

```
link /q dummy.obj, ,,bqlb40.lib;
```

In this case a Quick library called `dummy.qlb` is created; it contains only the **printf** function. You can then make calls to **printf** if you load this library when you invoke QuickBASIC, as follows:

```
qb /l dummy
```

Using this method, you can create Quick libraries that include useful functions from the standard C library, as well as other-language routines you write yourself. See Chapter 8, "Quick Libraries," for more information.

C.9 The B_OnExit Routine

You can use **B_OnExit** when your other-language routines take special actions that need to be undone before program termination or rerunning of the program. For example, within the QuickBASIC environment, an executing program that calls other-language routines in a Quick library may not always run to normal termination. If such routines need to take special actions at termination (for example, deinstallation of previously installed interrupt vectors), you can guarantee that your termination routines will always be called if you include an invocation of **B_OnExit** in the routine. The following example illustrates such a call (for simplicity, the example omits error-handling code). Note that such a function would be compiled in C in large model.

```
#include <malloc.h>

extern pascal far B_OnExit();  /* Declare the routine    */

int *p_IntArray;

void InitProc()
{
   void TermProc();      /* Declare TermProc function    */

   /* Allocate far space for 20-integer array:          */

   p_IntArray = (int *)malloc(20*sizeof(int));

   /* Log termination routine (TermProc) with BASIC:    */

   B_OnExit(TermProc);
}

                         /* The TermProc function is     */
void TermProc()          /* called before any restarting */
{                        /* or termination of program.   */

   free(p_IntArray);     /* Release far space allocated  */
}                        /* previously by InitProc.      */
```

If the `InitProc` function were in a Quick library, the call to **B_ OnExit** would insure proper release of the space reserved in the call to **malloc**, should the program end before normal termination could be performed. The routine could be called several times, since the program can be executed several times from the QuickBASIC environment. However, the `TermProc` function itself would be called only once each time the program runs.

The following BASIC program is an example of a call to the `InitProc` function:

```
DECLARE SUB InitProc CDECL

X = SETMEM(-2048)          ' Make room for the malloc memory
                           ' allocation in C function.
CALL InitProc
END
```

If more than 32 routines are registered, **B_ OnExit** returns NULL, indicating there is not enough space to register the current routine. (Note that **B_ OnExit** has the same return values as the Microsoft C run-time-library routine **onexit**.)

B_OnExit can be used with assembly-language or any other-language routines you place in a Quick library. With programs compiled and linked completely from the command line, **B_OnExit** is optional.

C.10 Assembly-Language-to-BASIC Interface

With the Microsoft Macro Assembler (MASM) you can write assembly-language modules that can be linked to modules developed with Microsoft BASIC. This section outlines the recommended programming guidelines for writing assembly-language routines compatible with Microsoft QuickBASIC.

Writing assembly-language routines for Microsoft high-level languages is easiest when you use the simplified segment directives provided with the Microsoft Macro Assembler, Version 5.0. In general, this manual assumes that you have Version 5.0. For information on writing assembly-language routines without the simplified segment directives, turn to Section C.10.4 to look up **SEGMENT**, **GROUP**, and **ASSUME** statements.

C.10.1 Writing the Assembly-Language Procedure

You can call assembly-language procedures using essentially the same conventions as for compiler-generated code. This section describes how you use those conventions to call assembly-language procedures. Procedures that observe these conventions can be called recursively and can be effectively used with the Stack Trace feature of the Microsoft CodeView debugger, included with Microsoft Macro Assembler, Version 5.0, and Microsoft C, Version 5.0.

The standard assembly-language-interface method consists of these steps:

- Setting up the procedure
- Entering the procedure
- Allocating local data (optional)
- Preserving register values
- Accessing parameters
- Returning a value (optional)
- Exiting the procedure

Sections C.10.1.1–C.10.1.8 describe each of these steps.

Note

The Microsoft Macro Assembler, Version 5.0 distribution disks contain an include file, **MIXED.INC**, that automatically performs many of the stack-maintenance chores described in the next few sections. Documentation for **MIXED.INC** is contained in the file **MIXED.DOC**, also included on the Version 5.0 distribution disks. Later versions may contain this information within the printed manuals.

C.10.1.1 Setting Up the Procedure

The linker cannot combine the assembly-language procedure with the calling program unless compatible segments are used and unless the procedure itself is declared properly. The following four points may be helpful:

1. If you have Version 5.0 of the Microsoft Macro Assembler, use the **.MODEL** directive at the beginning of the source file. This directive automatically causes the appropriate kind of returns to be generated (**NEAR** for small or compact model, **FAR** otherwise). Modules called from BASIC should be **.MODEL MEDIUM**. If you have a version of the Macro Assembler previous to 5.0, declare the procedure **FAR**.
2. If you have Version 5.0 or later of the Microsoft Macro Assembler, use the simplified segment directives **.CODE** to declare the code segment and **.DATA** to declare the data segment. (Having a code segment is sufficient if you do not have data declarations.) If you are using an earlier version of the assembler, look up the **SEGMENT**, **GROUP**, and **ASSUME** directives in Section C.10.4, "The Microsoft Segment Model."
3. The procedure label must be declared public with the **PUBLIC** directive. This declaration makes the procedure available to be called by other modules. Also, any data you want to make public to other modules must be declared as **PUBLIC**.
4. Global data or procedures accessed by the routine must be declared **EXTRN**. The safest way to use code **EXTRN** declarations is to place the directive outside of any segment definition.

C.10.1.2 Entering the Procedure

Two instructions begin the procedure:

```
push    bp
mov     bp,sp
```

This sequence establishes **BP** as the "framepointer." The framepointer is used to access parameters and local data, which are located on the stack. The value of the base register **BP** will remain constant throughout the procedure, so that each parameter can be addressed as a fixed displacement off of **BP**. (**SP** cannot be used for this purpose because it is not an index or base register. Also, the value of **SP** may change as more data are pushed onto the stack.)

The instruction `push bp` preserves the value of **BP**. This value will be needed by the calling procedure as soon as the current procedure terminates. The instruction `mov bp, sp` captures the value that the stack pointer had at the time of entry to the procedure (thus establishing parameter addressability).

C.10.1.3 Allocating Local Data (Optional)

An assembly procedure can use the same technique that is used by high-level languages for implementing local data. To set up local data space, simply decrease the contents of **SP** in the third instruction of the procedure. (To ensure correct execution, you should always increase or decrease **SP** by an even amount.) Decreasing **SP** reserves space on the stack for the local data. The space must be restored at the end of the procedure.

```
push    bp
mov     bp,sp
sub     sp,space
```

In the text above, *space* is the total size in bytes of the local data. Local variables are then accessed as fixed, negative displacements off of **BP**.

■ Example

The example below uses two local variables, each of which is two bytes in size. **SP** is decreased by 4, since there are four bytes total of local data. Later, each of the variables is initialized to 0. These variables are never formally declared with any assembler directive; the programmer must keep track of them manually. Local variables are also called dynamic, stack, or automatic variables.

```
push    bp
mov     bp,sp
sub     sp,4
.
.
.
mov     WORD PTR [bp-2],0
mov     WORD PTR [bp-4],0
```

C.10.1.4 Preserving Register Values

A procedure called from any of the Microsoft high-level languages should preserve the values of **SI**, **DI**, **SS**, and **DS** (in addition to **BP**, which is already saved). Therefore, push any of the register values that the procedure alters. If the procedure does not change the value of any of these registers, then the registers do not need to be pushed.

The recommended method (used by the high-level languages) is to save registers after the framepointer is set and local data space (if any) is allocated.

In the example below, **DI** and **SI** (in that order) must be popped before the end of the procedure.

```
push    bp              ; Save old framepointer.
mov     bp,sp           ; Establish current framepointer.
sub     sp,4            ; Allocate local data space.
push    si              ; Save SI and DI.
push    di
.
.
.
```

Note

Although saving the values of **DI** and **SI** is recommended, it was not necessary with procedures used with previous versions of QuickBASIC and is not necessary for QuickBASIC Version 4.0.

C.10.1.5 Accessing Parameters

Once you have established the procedure's framepointer, allocated local data space (if desired), and pushed any registers that need to be preserved, you can write the main body of the procedure. To write instructions that can access parameters, consider the general picture of the stack frame after a procedure call, as illustrated in Figure C.7.

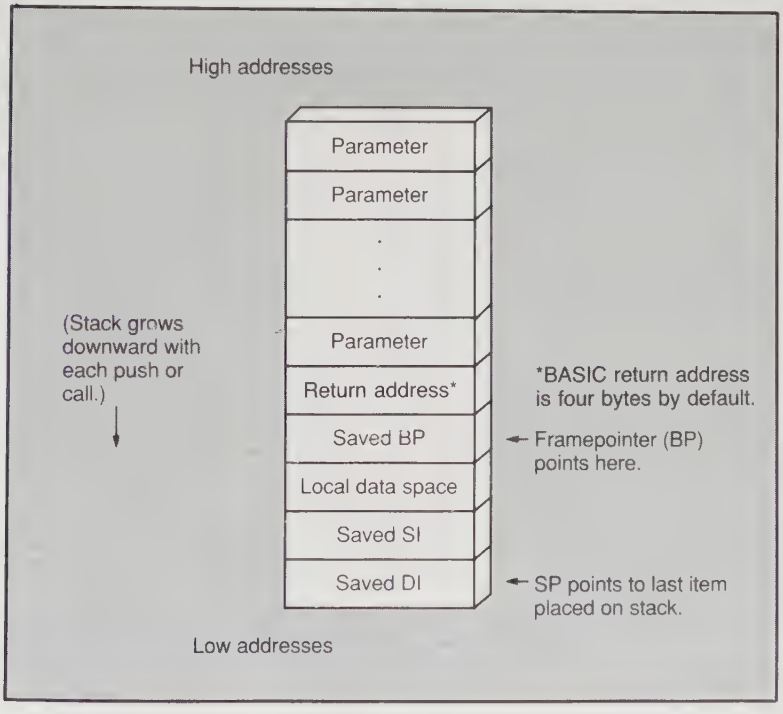


Figure C.7 The Stack Frame

The stack frame for the procedure is established by the following sequence of events:

1. The calling program pushes each of the parameters on the stack, after which **SP** points to the last parameter pushed.
2. The calling program issues a **CALL** instruction, which causes the return address (the place in the calling program to which control ultimately returns) to be placed on the stack. Because BASIC always uses a **FAR** call, this address is four bytes long. **SP** now points to this address.
3. The first instruction of the called procedure saves the old value of **BP**, with the instruction `push bp`. Now **SP** points to the saved copy of **BP**.
4. **BP** is used to capture the current value of **SP**, with the instruction `mov bp,sp`. **BP** therefore now points to the old value of **BP**.
5. Whereas **BP** remains constant throughout the procedure, **SP** may be decreased to provide room on the stack for local data or saved registers.

In general, the displacement (off of **BP**) for a parameter X is equal to

```
2 + size of return address
  + total size of parameters between X and BP
```

For example, consider a procedure that has received one parameter, a two-byte address. Since the size of the return address is always four bytes in BASIC, the displacement of the parameter would be

```
Argument's displacement =  2 + size of return address
                        =  2 + 4
                        =  6
```

The argument can thus be loaded into **BX** with the following instruction:

```
        mov     bx,[bp+6]
```

Once you determine the displacement of each parameter, you may want to use string equates or structures so that the parameter can be referenced with a single identifier name in your assembly source code. For example, the parameter above at **BP+6** can be conveniently accessed if you put the following statement at the beginning of the assembly source file:

```
Arg1    EQU     [bp+6]
```

You could then refer to this parameter as `Arg1` in any instruction. Use of this feature is optional.

Note

Microsoft high-level languages always push segment addresses before pushing offset address. Furthermore, when pushing arguments larger than two bytes, high-order words are always pushed before low-order words.

This standard for pushing segment addresses before pushing offset addresses facilitates the use of the **LES** and **LDS** instruction.

C.10.1.6 Returning a Value (Optional)

BASIC has a straightforward convention for receiving return values when the data type to be returned is simple (that is, not a floating-point value, an array, or structured type) and is no more than four bytes long. This includes all pointers and all parameters passed by reference, as shown in the following list:

Data Type	Returned in Register
2-byte integer(%)	**AX**
4-byte integer(&)	High-order portion in **DX**; low-order portion in **AX**
All other types	Near offset in **AX**

An assembly-language procedure called by BASIC must use a special convention to return floating-point values, user-defined types and arrays, and integer values larger than four bytes, as explained in Section C.10.1.7.

C.10.1.7 Numeric Return Values Other Than Two- and Four-Byte Integers

In order to create an interface for numeric return values that are neither two-byte integers (%) nor four-byte integers(&), BASIC modules take the following actions before they call your procedure (assuming that the BASIC declarations do not specify the **CDECL** keyword):

1. When the call to your procedure is made, an extra parameter is passed; this parameter contains the offset address of the actual return value. This parameter is placed immediately above the return address. (In other words, this parameter is the last one pushed.)
2. The segment address of the return value is contained in both **SS** and **DS**.

The extra parameter (which contains the offset address of the return value) is always located at **BP+6**. Furthermore, its presence automatically increases the displacement of all other parameters by two, as shown in Figure C.8.

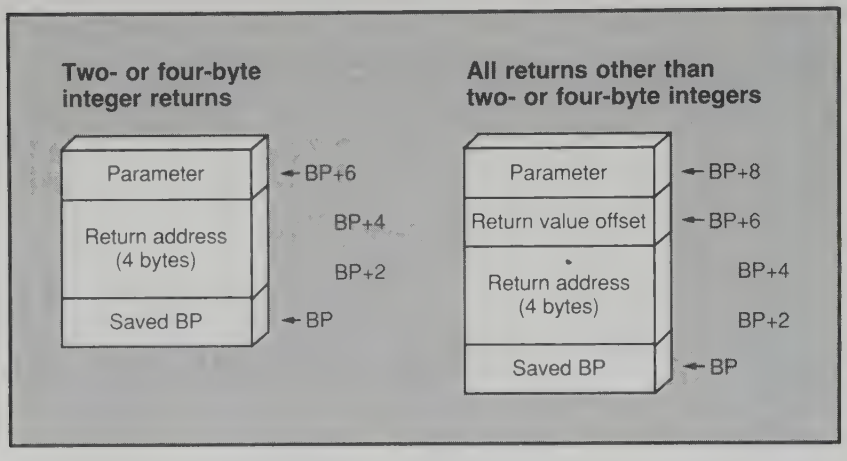


Figure C.8 BASIC Return Values

Your assembly-language procedure can successfully return numeric values other than two- and four-byte integers if you follow these steps:

1. Put the data for the return value at the location pointed to by the return-value offset.
2. Copy the return-value offset (located at **BP+6**) to **AX**. This is necessary because the calling module expects **AX** to point to the return value.
3. Exit the procedure as described in the next section.

C.10.1.8 Exiting the Procedure

Several steps may be involved in terminating the procedure:

1. If any of the registers **SS**, **DS**, **SI**, or **DI** have been saved, these must be popped off the stack in the reverse order from that in which they were saved.

2. If local data space was allocated at the beginning of the procedure, **SP** must be restored with the instruction

   ```
   mov sp,bp
   ```

3. Restore **BP** with the instruction

   ```
   pop bp
   ```

   This step is always necessary.

4. Since the BASIC calling convention is being used, you must use the `ret` *n* form of the instruction to adjust the stack with respect to the parameters that were pushed by the caller.

■ Examples

The example below shows the simplest possible exit sequence. No registers were saved, no local data space was allocated, and no parameters were passed to the routine.

```
pop     bp
ret
```

The example below shows an exit sequence for a procedure that has previously saved **SI** and **DI**, allocated local data space, used the BASIC calling convention, and received 6 bytes of parameters. The procedure must therefore use `ret 6` to restore the six bytes of parameters on the stack.

```
pop     si
pop     sp,bp       ; Remove local data space.
pop     bp          ; Restore old framepointer.
ret     6           ; Exit, and remove 6 bytes of args.
```

C.10.2 Calls from BASIC

A BASIC program can call an assembly-language procedure in another source file with the **CALL** or **CALLS** statement. Proper use of the **DECLARE** statement is also important. In addition to the steps outlined in Section C.10.1, "Writing the Assembly-Language Procedure," the following guidelines may be helpful:

1. Declare procedures called from BASIC as **FAR**.

2. Observe the BASIC calling convention:

   a. Parameters are placed on the stack in the same order in which they appear in the BASIC source code. The first parameter is highest in memory (because it is also the first parameter to be placed on the stack, and the stack grows downward).

   b. By default, BASIC parameters are passed by reference as two-byte addresses.

   c. Upon exit, the procedure must reset **SP** to the value it had before the parameters were placed on the stack. This is accomplished with the instruction `ret` *size*, where *size* is the total size in bytes of all the parameters.

3. Observe the BASIC naming convention.

   BASIC outputs symbolic names in uppercase characters, which is also the default behavior of the assembler. BASIC recognizes up to 40 characters of a name, whereas the assembler recognizes only the first 31 (this should rarely create a problem).

Note

The QuickBASIC distribution disk 3 includes a Quick library called QB.QLB and an include file called QB.BI. These files contain several routines that facilitate calling assembly-language routines from within the QuickBASIC environment. See Chapter 8, "Quick Libraries," for information on using this library.

In the following example program, BASIC calls an assembly-language procedure that calculates $A \times 2^B$, where A and B are the first and second parameters, respectively. The calculation is performed by shifting the bits in A to the left, B times.

```
' BASIC program
'
DEFINT A-Z
'
DECLARE FUNCTION Power2(A%,B%)
'
PRINT "3 times 2 to the power of 5 is ";
PRINT Power2(3,5)
END
```

To understand how to write the assembly-language procedure, recall how the parameters are placed on the stack, (shown in Figure C.9):

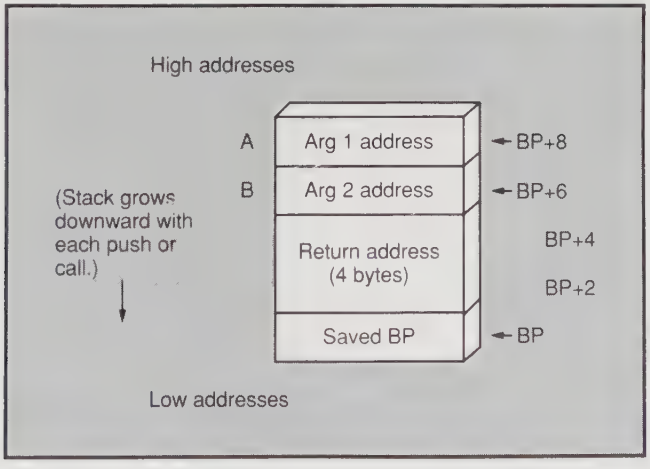


Figure C.9 BASIC Stack Frame

The return address is four bytes long because procedures that are called from BASIC must be **FAR**. Arg 1 (parameter 1) is higher in memory than Arg 2 because BASIC pushes arguments (parameters) in the same order in which they appear. Also, each argument is passed as a two-byte offset address, the BASIC default.

The assembly-language procedure can be written as follows:

```
.MODEL MEDIUM
.CODE
        PUBLIC  Power2
Power2  PROC
        push    bp              ; Entry sequence - saved old BP
        mov     bp,sp           ; Set stack framepointer

        mov     bx,[bp+8]       ; Load Arg1 into
        mov     ax,[bx]         ;   AX
        mov     bx,[bp+6]       ; Load Arg2 into
        mov     cx,[bx]         ;   CX
        shl     ax,cl           ; AX = AX * (2 to power of CX)
                                ; Leave return value in AX

        pop     bp              ; Exit sequence - restore old BP
        ret     4               ; Return, and restore 4 bytes
Power2  ENDP
        END
```

Note that each parameter must be loaded in a two-step process because the *address* of each is passed rather than the value. Also, note that the stack is restored with the instruction `ret 4` since the total size of the parameters is four bytes. (The preceding example is simplified to illustrate the interlanguage interface. Code for handling possible errors, such as overflow, is not included, but is a significant consideration with such procedures.)

C.10.3 Using CDECL in Calls from BASIC

You can use the **CDECL** keyword in the **DECLARE** statement in your BASIC module to call an assembly-language routine. If you do so, the C calling conventions, rather than those of BASIC, determine the order of the arguments as received in the assembly-language routine, and also the manner in which returns are handled. The primary advantage of using **CDECL** is that then you can call the assembly-language routine with a variable number of arguments. The technique is analogous to calling a C function from BASIC using **CDECL** (see Section C.3). In using **CDECL**, observe the C calling convention:

1. Parameters are placed on the stack in the reverse order to that in which they appear in the BASIC source code. This means the first parameter is lowest in memory (because the stack grows downward, and it is the last parameter to be placed on the stack).

2. Return with a simple `ret` instruction. Do *not* restore the stack with `ret` *size*, since using **CDECL** causes the calling routine to restore the stack itself as soon as the calling routine resumes control.

   When the return value is not a two-byte or four-byte numeric value, a procedure called by BASIC with **CDECL** in effect must allocate space for the return value, and then place its address in **AX**. A simple way to create space for the return value is to declare it in a data segment.

3. If **CDECL** is used in the BASIC **DECLARE** statement, you must name the assembler procedure with a leading underscore, unless you use the **ALIAS** feature (see Section C.3.1.2).

C.10.4 The Microsoft® Segment Model

If you use the simplified segment directives by themselves, you do not need to know the names assigned for each segment. However, versions of the Macro Assembler prior to 5.0 do not support these directives. With older versions of the assembler, you should use the **SEGMENT**, **GROUP**, **ASSUME**, and **ENDS** directives equivalent to the simplified segment directives.

Table C.5 shows the default segment names created by each directive for medium model, the only model applicable to BASIC. Use of these segments ensures compatibility with Microsoft languages and helps you to access public symbols. This table is followed by a list of three steps, illustrating how to make the actual declarations.

Table C.5

Default Segments and Types for Medium Memory Model

Directive	Name	Align	Combine	Class	Group
.CODE	*name*_TEXT	WORD	PUBLIC	'CODE'	---
.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
.STACK	STACK	PARA	STACK	'STACK'	DGROUP

The directives in Table C.5 refer to the following kinds of segments:

Directive	Description of Segment
.CODE	The segment containing all the code for the module.
.DATA	Initialized data.
.DATA?	Uninitialized data. Microsoft compilers store uninitialized data separately because it can be more efficiently stored than initialized data.
.CONST	Constant data. Microsoft compilers use this segment for such items as string and floating-point constants.
.STACK	Stack. Normally, this segment is declared in the main module for you and should not be redeclared.

The following steps describe how to use Table C.5 to create directives:

1. Use Table C.5 to determine the segment name, align type, combine type, and class for your code and data segments. Use all of these attributes when you define a segment. For example, the code segment is declared as follows:

   ```
   name_TEXT       SEGMENT   WORD PUBLIC 'CODE'
   ```

   The *name*_TEXT and all the attributes are taken from Table C.5. You substitute your module name for *name*. If the combine type is private, simply do not use any combine type.

2. If you have segments in **DGROUP**, put them into **DGROUP** with the **GROUP** directive, as in:

   ```
   GROUP           DGROUP    _DATA _BSS
   ```

3. Use **ASSUME** and **ENDS** as you would normally. Upon entry, **DS** and **SS** both point to **DGROUP**; therefore, a procedure that makes use of **DGROUP** should include the following **ASSUME** directive:

   ```
   ASSUME          CS:TEXT, DS:DGROUP, SS:DGROUP
   ```

Note

If your assembly-language procedures use real numbers, they must use IEEE-format numbers to be compatible with QuickBASIC. This is the default for Microsoft Macro Assembler, Version 5.0. With earlier versions, you must specify the **IR** command-line option or the **8087** directive.

Appendix D
Command Summary

D.1 Invocation Options....337
D.2 Menu Commands and Shortcut Keys....339
D.3 Debugging Commands....344
D.4 Editing Commands....345
D.5 Window Commands....348

D.1 Invocation Options

Table D.1 lists and explains all of the options available when invoking qb from the DOS command line.

Table D.1
Invocation Options for the qb Command

Option	Description
/ah	Allows dynamic arrays of records, fixed-length strings, and numeric data to be larger than 64K each. If this option is not specified, the maximum size is 64K per array. Note that this option has no effect on the way data items are passed to procedures.
/b	Starts in black and white with a CGA
/c:*buffersize*	Sets the size of the buffer receiving remote data
/cmd *string*	Stores everything from /cmd to the end of the command line for the **COMMAND$** function. When present, this option must be the last one on the DOS command line.
/g	Causes QuickBASIC to update the screen as fast as possible. Snow may appear if you use this option with a CGA.
/h	Displays the highest resolution possible on your hardware
/l [*libraryname*]	Loads the Quick library specified by *libraryname*
/mbf	Causes the QuickBASIC conversion functions to treat IEEE-format numbers as Microsoft Binary format numbers
/run *sourcefile*	Loads, compiles, and runs the specified source file
sourcefile	Loads the specified source file

Table D.2 lists and explains all of the options available when invoking bc from the DOS command line.

Table D.2

Invocation Options for the bc Command

Option	Description
/a	Creates a listing that shows the assembly-language code which the compiler generates for each line of the source file
/ah	Allows dynamic arrays of records, fixed-length strings, and numeric data to be larger than 64K each
/c:*buffersize*	Sets the size of the buffer receiving remote data
/d	Generates debugging code for run-time error checking and enables CTRL+BREAK
/e	Indicates the presence of **ON ERROR** with **RESUME** *linenumber* statements
/mbf	Causes the QuickBASIC conversion functions to treat IEEE-format numbers as Microsoft Binary format numbers
/o	Substitutes the BCOM40.LIB run-time library for BRUN40.LIB
/r	Stores arrays in row order rather than column order
/s	Writes quoted strings to the object file instead of the symbol table, requiring less memory
/v	Enables event trapping between BASIC statements for communications (**COM**), lightpen (**PEN**), joystick (**STRIG**), timer (**TIMER**), music buffer (**PLAY**), and function keys (**KEY**)
/w	Enables event trapping for same statements as /v, but checks between lines for occurrence of an event
/x	Indicates presence of **ON ERROR** with **RESUME**, **RESUME NEXT**, or **RESUME 0**
/zd	Generates line-number records corresponding to the line numbers of the source file. The records are for use with the Microsoft SYMDEB symbolic debugger.
/zi	Generates debugging information for use with the Microsoft CodeView debugger

D.2 Menu Commands and Shortcut Keys

Table D.3 lists and explains the QuickBASIC menu commands and shortcut keys.

Table D.3
Menu Commands and Shortcut Keys

Menu	Command (and Shortcut Key)	Action
File	New Program	Removes currently loaded program from memory
	Open Program	Loads new program into memory
	Merge	Inserts file into current module
	Save	Saves current module as a disk file
	Save As	Saves current module with specified name and format
	Save All	Writes all loaded modules to files on disk
	Create File	Creates a new module, include file, or document
	Load File	Loads an existing module, include file, or document
	Unload file	Removes a loaded module, include file, or document from memory
	Print	Prints specified text or modules
	DOS Shell	Invokes new DOS shell
	Exit	Exits QuickBASIC
Edit	Undo (ALT+BKSP)	Reverses last edit
	Cut (SHIFT+DEL)	Removes text and copies it to Clipboard
	Copy (CTRL+INS)	Copies text to Clipboard
	Paste (SHIFT+INS)	Inserts contents of Clipboard at current location
	Clear (DEL)	Deletes text without copying to Clipboard
	New SUB	Opens a window for a new **SUB** procedure
	New FUNCTION	Opens a window for a new **FUNCTION** procedure
	Syntax Checking	Turns editor's automatic syntax checking on or off

Table D.3 *(continued)*

Menu	Command (and Shortcut Key)	Action
View	SUBs (F2)	Displays a list of loaded **SUB**, **FUNCTION**, module, include, or document files
	Next SUB (SHIFT+F2)	Displays next subprogram in active window
	(CTRL+F2)	Displays previous subprogram in active window
	Split	Toggles between split screen and single View window
	Next Statement	Displays next statement to be executed
	Output Screen (F4)	Switches to output screen
	Include File	Displays lines from include file for editing
	Include Lines	Displays lines from include file
	Options	Changes display attributes
Search	Find	Finds specified text
	Selected Text (CTRL+\)	Finds selected text
	Repeat Last Find (F3)	Finds next occurrence of specified text
	Change (CTRL+Q A)	Finds and changes specified text
	Label	Finds line label
Run	Start (SHIFT+F5)	Runs current program
	Restart	Restarts current program
	Continue (F5)	Continues execution
	Modify COMMAND$	Sets string returned by **COMMAND$** function
	Make EXE File	Creates executable file on disk
	Make Library	Creates Quick library on disk
	Set Main Module	Changes main module

Table D.3 *(continued)*

Menu	Command (and Shortcut Key)	Action
Debug	Add Watch	Sets watch expression
	Watchpoint	Sets watchpoint
	Delete Watch	Deletes specified Watch-window entry
	Delete All Watch	Deletes Watch-window entries
	Trace On	Highlights currently executing statement
	History On	Records statement execution order
	Toggle Breakpoint (F9)	Sets/clears breakpoint at cursor location
	Clear All Breakpoints	Clears all breakpoints
	Set Next Statement	Indicates next statement to be executed
Calls	*procedure*	Displays active procedure calls, moves cursor to *procedure* call line
Help	General (F1)	Provides information on general topics
	Topic (SHIFT+F1)	Provides context-sensitive information
	Close Help (ESC)	Closes Help window

Table D.4 lists and explains all of the QuickBASIC shortcut keys and equivalent menu commands.

Table D.4
QuickBASIC Shortcut Keys

Shortcut Key	Action	Menu Command
F1	Displays information on general topics	General command from Help menu
F2	Displays a list of loaded **SUB**, **FUNCTION**, module, include, or document files	SUBs command from View menu
F3	Finds next occurrence of previously specified text	Repeat Last Find command from Search menu
F4	Displays output screen	Output Screen command from View menu
F5	Continues program execution from current statement	Continue command from Run menu
F6	Makes next window the active window	None
F7	Executes program to current cursor position	None
F8	Executes next program statement, traces through procedure	None
F9	Toggles breakpoint	Toggle Breakpoint command from Debug menu
F10	Executes next program statement, traces around procedure	None
SHIFT+F1	Displays context-sensitive information about QuickBASIC keywords	Topic command from Help menu
SHIFT+F2	Displays next subprogram in active window	Next SUB command from View menu

Table D.4 *(continued)*

Shortcut Key	Action	Menu Command
SHIFT+F5	Starts program execution from beginning	Start command from Run menu
SHIFT+F6	Makes previous window the active window	None
SHIFT+F8	History back	None
SHIFT+F10	History forward	None
SHIFT+DEL	Deletes selected text, saving in Clipboard	Cut command from Edit menu
SHIFT+INS	Inserts contents of Clipboard at current cursor position	Paste command from Edit menu
CTRL+F2	Displays previous subprogram in active window	None
CTRL+F5	Restores full-screen window to previous size	None
CTRL+F10	Makes active window full screen/restore split screen	None
CTRL+\	Finds selected text	Selected Text command from Search menu
CTRL+INS	Copies selected text to Clipboard	Copy command from Edit menu
CTRL+Q A	Finds and changes selected text	Change command from Search menu
ALT+BKSP	Undoes last edit	Undo command from Edit menu
DEL	Deletes selected text, not saving in Clipboard	Clear command from Edit menu
ESC	Closes Help window, dialog box, error message box, or menu	Close Help command from Help menu

D.3 Debugging Commands

Table D.5 summarizes the keyboard commands used for debugging.

Table D.5
Keyboard Debugging Commands

Action	Key
View output screen	F4
Continue execution from current statement	F5
Start execution from beginning	SHIFT+F5
Execute to current cursor position	F7
Execute the next program statement, trace through procedure	F8
Trace history back	SHIFT+F8
Toggle breakpoint	F9
Execute the next program statement, trace around procedure	F10
Trace history forward	SHIFT+F10

D.4 Editing Commands

Table D.6 summarizes the editing commands.

Table D.6
Editing Commands

Moving Cursor	Keypad Keys	WordStar-Style Key Combinations
Character left	LEFT	CTRL+S
Character right	RIGHT	CTRL+D
Word left	CTRL+LEFT	CTRL+A
Word right	CTRL+RIGHT	CTRL+F
Line up	UP	CTRL+E
Line down	DOWN	CTRL+X
First indentation level	HOME	---
Column 1 of current line	---	CTRL+Q S
Beginning of next line	CTRL+ENTER	CTRL+J
End of line	END	CTRL+Q D
Top of window	---	CTRL+Q E
Bottom of window	---	CTRL+Q X
Beginning of module/procedure	CTRL+HOME	CTRL+Q R
End of module/procedure	CTRL+END	CTRL+Q C
Set marker	---	CTRL+K *n*†
Move to marker	---	CTRL+Q *n*†

Table D.6 *(continued)*

Inserting	Keypad Keys	WordStar-Style Key Combinations
Insert mode on or off	INS	CTRL+V
Line below	END ENTER	---
Line above	---	HOME CTRL+N
Contents of Clipboard	SHIFT+INS	---
Tab at cursor or beginning of each selected line	TAB	CTRL+I
Control character at cursor position	---	CTRL+P CTRL+*key*‡
Deleting		
Current line, saving in Clipboard	---	CTRL+Y
To end of line, saving in Clipboard	---	CTRL+Q Y
Character left	BKSP	CTRL+H
Character at cursor	DEL	CTRL+G
Word	---	CTRL+T
Selected text, not saving in Clipboard	DEL	CTRL+G
Selected text, saving in Clipboard	SHIFT+DEL	---
Leading spaces for one indentation level of selected lines	SHIFT+TAB	---

Table D.6 *(continued)*

Copying	Keypad Keys	WordStar-Style Key Combinations
Selected text, saving in Clipboard	CTRL+INS	---
Scrolling		
Up one line	UP	CTRL+W
Down one line	DOWN	CTRL+Z
Page up	PGUP	CTRL+R
Page down	PGDN	CTRL+C
Left one window	CTRL+PGUP	---
Right one window	CTRL+PGDN	---
Selecting		
Character left	SHIFT+LEFT	---
Character right	SHIFT+RIGHT	---
Current line	SHIFT+DOWN	---
Line above	SHIFT+UP	---
Word left	SHIFT+CTRL+LEFT	---
Word right	SHIFT+CTRL+RIGHT	---
Screen up	SHIFT+PGUP	---
Screen down	SHIFT+PGDN	---
Screen left	SHIFT+CTRL+PGUP	---
Screen right	SHIFT+CTRL+PGDN	---
To beginning of module/ procedure	SHIFT+CTRL+HOME	---
To end of module/procedure	SHIFT+CTRL+END	---

† Replace *n* with a number from 0–3 to identify the desired marker.

‡ The CTRL+P combination is followed by CTRL and the desired key.

D.5 Window Commands

Table D.7 shows the keyboard and mouse versions of each window command. Unless otherwise stated, click with the left button when using the mouse.

Table D.7
Window Commands

Action	Keyboard	Mouse
Go to Help screen	F1	Click on General command from Help menu.
Context-sensitive Help	SHIFT+F1	Click on Topic command from Help menu.
Repeat previous find	F3	Click on Repeat Last Find command from Search menu.
Switch to output screen	F4	Click on Output Screen command from View menu.
Split View window	None	Click on Split command from View menu.
Continue execution	F5	Click on Continue command from Run menu.
Go to next window	F6	Click in desired window.
Go to previous window	SHIFT+F6	Click in desired window.
Show next procedure	SHIFT+F2	Click on Next SUB command from View menu.
Show previous procedure	CTRL+F2	None

Table D.7 *(continued)*

Action	Keyboard	Mouse
Execute to here	F7 at location	Click right button at location.
Execute next statement	F8	None
Set/clear breakpoint here	F9 at location	Click on Toggle Breakpoint command from Debug menu.
Trace over procedure	F10	None
Display previous history	SHIFT+F8	None
Go forward in history window	SHIFT+F10	None
Make active window full screen/restore split screen	CTRL+F10	Double-click title bar or click maximize icon with left button.
Enter menu-selection mode	ALT or F12	None
Enlarge active window	ALT+PLUS	Drag title bar up.
Shrink active window	ALT+MINUS	Drag title bar down.
Scroll up line in window	UP	Click on up arrow.
Scroll up page in window	PGUP	None
Scroll to top of window	CTRL+HOME	None
Scroll down line in window	None	Click on down arrow.
Scroll down page in window	PGDN	None
Scroll to bottom of window	CTRL+END	None
Move cursor up	UP	None
Move cursor down	DOWN	None

GLOSSARY

The purpose of this glossary is to define unique and difficult terms used in all the manuals that accompany QuickBASIC Version 4.0. Neither the individual definitions nor the list of terms is intended to be a comprehensive dictionary of computer-science terminology.

Animate

A QuickBASIC debugging feature in which each line in a running program is highlighted as it executes. The Trace On command from the Debug menu turns on animation.

Argument

A data item supplied to a BASIC statement, function, or procedure. In the following example, the string "Hello" is an argument to the BASIC **PRINT** statement:

```
PRINT "Hello"
```

An argument may be a constant or a variable. Arguments are often used to pass information from one part of a program to another.

Automatic variable

A variable in a **SUB** or **FUNCTION** procedure whose value is not saved between invocations of the procedure. Automatic variables are frequently used in recursive procedures where you do not want values saved between calls. Automatic variables are the default when you omit the **STATIC** keyword from the **SUB** or **FUNCTION** statement.

Base name

The part of a file name before the extension. For example, SAMPLE is the base name of the file SAMPLE.BAS.

Basic Input/Output System (BIOS)

Machine language programs built into a computer. These programs provide the lowest level of operations in a computer system. You can directly use the BIOS programs with the **CALL** INTERRUPT and **CALL** INTERRUPTX statements.

Binary

A mathematical term referring to the base two numbering system, which uses only two digits: 0 and 1. In input/output operations, binary files or data are those which contain nonprinting characters as well as ordinary ASCII characters. Binary file access is the ability to access any byte within a file.

Calling by reference

See "Passing by reference."

Calling by value

See "Passing by value."

Check box

A box that allows you to turn an option on or off. An X appears in the box when the option is on.

Choose

To carry out a command by performing an action such as picking a command button from a dialog box or an item from a menu.

Clause

Part of a BASIC statement. In

```
IF NOT Found(Keyword$) THEN
    ErrorFlag=TRUE
    CALL ErrorProc("Not a keyword")
ELSE
    CALL AddToTable(Keyword$)
END IF
```

the following portion is a clause:

```
ELSE
    CALL AddToTable(Keyword$)
```

Click

To press and release one of the mouse buttons while pointing the mouse at an object on the screen.

Clipboard

A storage area that holds text that you are copying or moving. Text does not accumulate in the clipboard: new text added to the clipboard deletes any text that was already there.

Command

A QuickBASIC menu item or a DOS command.

Compile

The action QuickBASIC or BC performs to translate BASIC statements to a form that can be executed by the computer.

Compile time

The time during which BASIC statements are translated. Errors such as incorrect syntax are detected during translation and are called compile-time errors.

Compiler

A program that translates a BASIC program into a language understood by the computer.

Constant

A value that does not change during program execution. A variable, on the other hand, is a value that can—and usually does—change during program execution.

See "Symbolic constant."

Context

Shown on the QuickBASIC status bar to identify the executing part of a running program. This context consists of the name of a module or the name of a module followed by the name of a procedure.

Cursor

The blinking line or box on the screen. The cursor indicates the current location where QuickBASIC or your BASIC program is ready to accept input.

DEF FN function

The object defined by a single-line **DEF FN** statement or by a **DEF FN...END DEF** block. Earlier versions of BASIC utilized **DEF FN** functions to allow users to define their own functions that would work like BASIC functions such as **ABS** and **SQR**.

DEF*type*

A name used in these manuals for the group of BASIC statements that redefine a program's default variable type (**DEFDBL**, **DEFINT**, **DEFLNG**, **DEFSNG**, **DEFSTR**).

Default data type

The type that BASIC uses for numeric data unless you specify otherwise with a **DEF***type* statement. The default type for numeric data is single precision.

Dialog box

A box that appears when you choose a menu command that requires additional information.

Dimension

The number of subscripts required to specify a single array element. For instance, the statement

```
DIM A$(4,12)
```

declares a two-dimensional array, while

```
DIM B$(16,4,4)
```

declares a three-dimensional array.

Document

A way of classifying a file when you load it into the QuickBASIC editor. Loading a file as a document turns off syntax checking, capitalization of keywords, and other smart-editor features. You can load a BASIC program as a document if you want to edit it as you would with a word processor.

Double precision

A real (floating-point) value that occupies eight bytes of memory. Double precision values are accurate to 15 or 16 digits.

Drag

To point the mouse at an object on the screen, press a mouse button, and then move the mouse while holding the button down.

$DYNAMIC

Indicates an array that is allocated at run time.

See $STATIC.

Executable file

A file with an extension of .EXE, .COM, or .BAT. Executable files can be run by typing the file name at the DOS prompt.

Execute

To run a program, or carry out a specific statement in the Immediate window.

Extension

The part of a file name following the period. For example, .DAT is the extension in the file name TEST.DAT.

Far address

A memory location specified by using a segment (location of a 64K block) and an offset from the beginning of the segment. Far addresses require four bytes—two for the segment, and two for the offset. A far address is also known as a segmented address.

FLAGS

A register containing information about the status of the central processing unit and the last operation performed.

Flashing underscore

See "Cursor."

Floating point

Notation, similar to scientific notation, that is a convenient method for expressing very large or very small numbers or numbers with fractions. A floating-point number has two parts: an exponent and a fractional part known as the mantissa.

See "IEEE format," and also see Chapter 2, "Data Types," in the *BASIC Language Reference.*

FUNCTION

A block of statements beginning with **FUNCTION** and ending with **END FUNCTION** that defines a procedure that can be called like the BASIC functions **SQR** or **HEX$**. **FUNCTION** procedures can be used instead of the **DEF FN** functions used in older forms of BASIC.

Global constant

A constant available throughout a module. Symbolic constants defined in the module-level code are global constants.

Global variable

A variable that is available throughout a module. Global variables can be declared in **COMMON**, **DIM**, or **REDIM** statements by using the **SHARED** attribute.

Heap

An area of random-access memory used by BASIC to store, among other things, variables and arrays.

Highlight

A reverse-video area in a text box, window, or menu marking the current command chosen or text that has been selected for copying or deleting.

See "Input focus."

IEEE format

A method of representing floating-point numbers internally. IEEE format gives more accurate results than the Microsoft Binary format used in earlier versions of QuickBASIC and makes it easier to use a math coprocessor. The acronym IEEE stands for Institute of Electrical and Electronics Engineers, the organization that developed this standard.

See "Floating point."

Immediate window

The area at the bottom of the QuickBASIC screen used to enter and immediately execute statements.

Include file

A source file that is compiled into a program with the **$INCLUDE** metacommand.

Input focus

Indicates an item in a dialog box that is the object of the next action. The item that has the input focus is highlighted either in high-intensity or reverse video.

Integer

A whole number represented inside the machine as a 16-bit two's complement binary number. An integer has a range of –32,768 to +32,767.

See "Long integer."

Interpreter

A program that translates and executes one BASIC statement at a time. An interpreter translates a statement every time it executes the statement. BASICA is a language interpreter.

Compare with "Compiler."

Keyword

The name of a BASIC statement, function, or operator. Examples of keywords are **MOD**, **OPEN**, **OR**, **PRINT**, and **SIN**.

Library

A file having the extension .LIB and containing one or more compiled object files. Linking a compiled program with a library gives that program access to all the procedures in that library.

See "Quick library."

Line styling

A method of drawing dashed or dotted lines with the **LINE** graphics statement. This is done by specifying a 16-bit style mask in the **LINE** statement.

Link

The step that the linker performs to produce an executable file. The link step resolves references to procedures or variables in other modules and creates a complete program ready for execution.

Link time

The time when the linker is executing. During this time it collects and links object files and library files.

See "Compile time" and "Run time."

List box

A box listing a series of items. For example, the box listing files in the Open Program dialog box is a list box.

See "Dialog box."

Load

To copy a file into memory.

Local constant

A constant whose scope is limited to a procedure (**FUNCTION** or **SUB**) in a module. All constants declared in **CONST** statements inside a procedure are local constants.

Local variable

A variable whose scope is confined to a particular unit of code, such as the module-level code, or a procedure (**FUNCTION** or **SUB**) within a module. All variables in a procedure are local unless they appear in a **SHARED** statement within the procedure, or in a **COMMON SHARED**, **DIM SHARED**, or **REDIM SHARED** statement at the module level of the module containing the procedure definition.

See "Module-level code," "Procedure."

Logical coordinates

Pairs of numbers (for example, x,y) that uniquely define any point in two-dimensional space. The numbers x and y can be any real numbers. A set of logical coordinates can be mapped to the screen with the graphics **WINDOW** statement. Logical coordinates let you plot data using coordinates appropriate to the graph or picture without worrying about the actual screen coordinates.

See "Physical coordinates."

Long integer

A whole number represented inside the machine by a 32-bit two's complement value. Long integers have a range of –2,147,483,648 to +2,147,483,647.

See "Integer."

Machine code

A series of binary numbers that a microprocessor executes as program instructions.

Main module

The module containing the point where program execution begins (the program's entry point).

See "Module."

Math coprocessor

An optional hardware component, such as an 8087 or 80287 chip, that improves the speed and accuracy of arithmetic involving floating-point numbers. QuickBASIC automatically takes advantage of a math coprocessor if it is present, or emulates it in software if it is not.

See "Floating point."

Memory map

A representation of where in memory the computer expects to find certain types of information.

Menu bar

The bar at the top of the QuickBASIC display containing the menu titles.

Metacommands

Special commands enclosed in comments in a source file that tell the compiler to take certain actions while it is compiling the program. For example, the **$INCLUDE** metacommand tells the compiler to include statements from another file.

Microsoft Binary format

A method of representing floating-point numbers internally. Microsoft Binary format is used by earlier versions of QuickBASIC.

See "Floating point" and "IEEE format."

Module

A discrete group of statements. Every program has at least one module (the main module). In most cases, a module is the same as a source file. When you save a program containing multiple modules, each module is saved in a separate disk file.

Module-level code

Program statements within any module that are outside a **SUB** or **FUNCTION** definition. Error- or event-handling code and declarative statements such as **COMMON**, **DECLARE**, and **TYPE** can appear only at the module level.

Mouse

A pointing device that fits under your hand and rolls in any direction on a flat surface. By moving the mouse, you can move the mouse pointer in a corresponding direction on the screen.

See "Pointer."

Near address

A memory location specified by using only the offset from the start of the segment. A near address requires only two bytes.

See "Far address."

Object file

A file (with the extension .OBJ) containing relocatable machine code produced by compiling a program and used by the linker to form an executable file.

Object module

The contents of an object file after the file has been made part of a stand-alone (.LIB) library.

See "Library."

Offset

The number of bytes from the beginning of a segment in memory to a particular byte in that segment.

Open

The operation performed on a file by the BASIC **OPEN** statement. The **OPEN** statement prepares a file for reading or writing.

Overflow

Occurs when the value assigned to a numeric variable falls outside the allowable range for that variable's type.

Page

See "Window."

Parameter

A variable in a **DEF FN**, **FUNCTION**, or **SUB** statement. Parameters are replaced by actual variables and values when you invoke the function or subprogam.

Passing by reference

Transferring the address of an argument to a **SUB** or **FUNCTION**. Passing by reference allows a procedure to change the values of arguments passed to it.

Passing by value

Transferring the value (rather than the address) of an argument to a **DEF FN**, **FUNCTION**, or **SUB** statement.

Physical coordinates

The screen coordinates. The physical coordinates (0,0) refer to the upper-left corner of the screen, unless you have defined a graphics viewport with a **VIEW** statement, in which case (0,0) are the coordinates of the upper-left corner of the viewport.

The range of the physical coordinates depends on the display adapter, the monitor and the specifications set by the **SCREEN** statement.

See "Logical coordinates."

Pointer

The reverse-video rectangle that moves to indicate the current position of the mouse. The mouse pointer only appears if a mouse is installed. To select an item with the mouse, move the mouse until the pointer rests on the item, then press the left button.

Procedure

A general term for a **SUB** or **FUNCTION**.

Program

One or more modules forming an instructional set that performs a specific task.

See "Module."

PUBLIC

A Macro Assembler directive used to declare symbols used in more than one module. PUBLIC symbols are important in mixed-language programs that include assembly-language procedures. You must declare as PUBLIC the name of every procedure that you wish to call from BASIC and the names of data items that you wish to share between the BASIC and assembly-language portions of the program. BASIC **SUB** and **FUNCTION** procedures are implicitly public.

Quick library

A file produced in one of the following ways:

- With the Make Library command on the QuickBASIC Run menu
- With the /Q option of the link command

A Quick library contains procedures that can be used by more than one program. Quick libraries were known in previous versions of QuickBASIC as "user libraries." User libraries cannot be used as Quick libraries, and vice versa.

Record

A term that refers to either one of the following:

- A variable declared as having a user-defined type.

  See "User-defined type."

- A series of fields treated as a meaningful unit in a data file.

Registers

Areas in the processor where data can be temporarily stored during machine-level processing. The registers used in the 8086-family of processors are: AX, BX, CX, DX, SP, BP, SI, DI, DS, ES, SS, CS, IP, and the FLAGS register.

See "FLAGS."

Run

To execute a program.

Run time

The time during which a program is executing. Run time refers to the execution time of a program, rather than to the execution time of the compiler or the linker. Some errors—known as run-time errors—can be detected only while the program is running.

Run-time module

A module containing most of the routines needed to implement the BASIC language. It is a peculiarity of the run-time module that it is an executable (.EXE) file. The run-time module is named BRUN40.EXE and is, for the most part, a library of routines.

Scope

The range of statements over which a variable or constant is defined.

See "Global constant," "Global variable," "Local constant," and "Local variable."

Scroll

To move text up and down, or left and right, in order to see parts that cannot fit on the screen.

Scroll bars

The bars that appear at the right side and bottom of the View window and some list boxes. Scroll bars allow you to scroll the contents of a View window or text box with the mouse.

Select

To pick items in a dialog box such as options, list boxes, or text boxes. "Select" also describes the action of picking areas of text or graphics to edit.

Simple variable

A BASIC variable that is not an array. Simple variables can be of numeric, string, or user-defined types.

Single precision

A real (floating-point) value that occupies four bytes of memory. Single-precision values are accurate to seven decimal places.

Source file

A text file containing BASIC statements. Although all modules are source files, not all source files (include files, for example) are modules.

QuickBASIC uses source files in two different formats. Source files saved in the QuickBASIC format load quickly but cannot be directly read or manipulated by an ordinary text editor. Text files (or ASCII files) can be read or changed by any text editor.

Stack

An area of random-access memory used by BASIC to store objects such as intermediate results of some arithmetic computations, automatic variables in procedures, and arguments passed to procedures. The stack is also where BASIC keeps track of information such as where a procedure was invoked.

Stand-alone file

An executable file that can be run without a run-time file—a program compiled then linked with BCOM40.LIB.

Statement

A combination of one or more labels, BASIC keywords, variables, and operators. A BASIC statement can stand by itself on a single program line. The term statement refers to one-line statements such as **OPEN** or **INPUT** as well as an entire multiline compound statement such as **IF...END IF** or **SELECT CASE...END SELECT**.

Parts of compound statements such as **ELSE** or **CASE** are referred to as "clauses."

$STATIC

Indicates an array allocated at compile time.

See "Automatic variable," "**$DYNAMIC**," and "**STATIC**"; and also see Appendix C, "Metacommands," in the *BASIC Language Reference*.

STATIC

A local variable in a **SUB** or **FUNCTION** whose value is saved between calls.

Status bar

The line at the bottom of the QuickBASIC screen. The status bar displays your text position, keyboard status, current context of execution, and other program information.

String

A contiguous sequence of characters, often identified by a symbolic name. In this example, "Hello" is a string:

```
PRINT "Hello"
```

A string may be a constant or a variable. QuickBASIC 4.0 supports fixed-length and variable-length strings.

SUB

A unit of code delimited by the **SUB** and **END SUB** statements. A **SUB** (sometimes called a "subprogram") provides an alternative to the older BASIC **GOSUB** type of subroutine.

Subprogram

See "SUB."

Subroutine

A unit of BASIC code terminated by the **RETURN** statement. Program control is transferred to a subroutine with a **GOSUB** statement.

Symbolic constant

A constant represented by a symbol rather than the constant itself. Symbolic constants are defined using the **CONST** statement and can make a program easier to read and modify.

Text box

A box where you type information needed to carry out a command. A text box appears within a dialog box. What you type appears in the box. The text box may be blank when the dialog box appears, or the box may contain text if there is a default option or you have selected something applicable to that command.

Tiling

Filling a figure with a pattern, instead of a solid color, by using the graphics statement **PAINT**.

Title bar

A bar at the top of a View window that shows the name of the module (and procedure, if appropriate) currently displayed in that window.

Toggle

A function key or menu selection that turns a feature off, if it is on, or turns the feature on, if it is off. When used as a verb, toggle means to reverse the status of a feature. For example, the Trace On command on the Debug menu is a toggle that switches the animated display of a trace on or off.

Two's complement

A way of representing whole numbers inside the machine. Positive numbers are represented by straightforward binary numbers. Negative numbers are represented by the two's complement of the positive value.

See Chapter 2, "Data Types," in the *BASIC Language Reference.*

Underflow

An error condition caused when a calculation produces a result too small for the computer to handle internally.

Unresolved external

A symbol referenced in one assembly-language module, but not made PUBLIC in another module that is linked with it. Unresolved external references are usually caused by misspellings or by omitting the name of the module containing the desired symbol from the link command line.

User-defined type

A composite data structure that can contain both string and numeric variables, similar to a C-language "structure" or Pascal "record." User-defined types are defined with **TYPE** statements.

The data structure is defined by a **TYPE...END TYPE** statement. In the following code fragment, Node is a user-defined type, Heap is an array of records, and one element of the Heap array, for instance, Heap(50), is a record:

```
TYPE Node
    LeftLink   AS  INTEGER
    RightLink  AS  INTEGER
    DataField  AS  STRING * 80
END TYPE

DIM Heap(1000) AS Node
```

See Chapter 2, "Data Types," in the *BASIC Language Reference*, and Section 3.4.6.2 in *Programming in BASIC: Selected Topics.*

User library

A form of library similar to a Quick library, but used in earlier versions of QuickBASIC. User libraries are incompatible with QuickBASIC Version 4.0.

See "Quick library."

View window

The area on the screen used to display or edit portions of a source file.

Watch window

Window that displays information about variables or expressions that you are monitoring during debugging. For each watched item, the Watch window displays the item's context, its name, and its current value if the item is watchable.

Watchable

A debugging term that describes whether a variable or expression can be watched in the current context. An item is watchable only while execution is occurring in the part of the program from which you entered the item in the Watch window.

See "Context."

Window

A term that refers to either one of the following:

- An area on the screen used to display part of a file or to enter statements. "Window" refers only to the area on the screen, and does not refer to what is displayed in the area.

  See "Immediate window," "Watch window," and "View window."

- The portion of a logical coordinate system mapped to the physical coordinates by the **WINDOW** graphics statement.

World coordinates

See "Logical coordinates."

& (ampersand)
  C address operator, 304
  type declaration character, 287
* (asterisk)
  C indirection operator, 304
  LIB command symbol, 231
$ (dollar sign), type declaration character, 287
! (exclamation point), type declaration character, 287
/ (forward slash), linker option character, 218
– (minus sign), LIB command symbol, 231, 232
–* (minus sign-asterisk), LIB command symbol, 231
–+ (minus sign-plus sign), LIB command symbol, 231, 232
% (percent sign), type declaration character, 287
+ (plus sign), LIB command symbol, 232
# (pound sign), type declaration character, 287
? (question mark), shortcut for PRINT, 145
; (semicolon), LIB command symbol, 230

8087, 12
80287, 12

,A option (BASICA), 20, 239
/a option (bc), 210
ABSOLUTE, 197
Active window
  changing, 78
  command button, 159
  line and column counters, 68
  option
    Change command, 127
    Find command, 124
    Print command, 109
  switching, 78
Add Watch command, 174, 176
Adding files to program, 132
Address sizes
  code, 284
Address sizes *(continued)*
  data, 285
Address variables, 316
ADR, address variable, 316
/ah option
  bc, 210, 338
  bc or qb, 263
  qb, 337
ALIAS statement, in mixed-language programs, 287
All Modules option
  Change command, 127
  Find command, 124
  Print command, 109
ALT key
  described, 69
  entering high-order ASCII characters, 129
Animate, defined, 351
Animate mode, 265
Animation, while debugging, 171
Apostrophe ('), entering, xxv
Arguments
  command-line, 148
  defined, 351
  linker options, 218
Arithmetic overflow
  checking, 153
  error, cause, 153
Array descriptors, BASIC, 311
Arrays
  bounds
    checking, 153
    error, cause, 153
  C language, 312
  declarations and indexing, 313
  mixed-language programs, 311
  row-order option, 262
Arrow keys. *See* DIRECTION keys
AS statement, mixed-language programs, 289
ASCII characters
  corresponding to control sequences, 130
  high order, 129
Assembly language
  books about, xxviii
  calling from BASIC
    described, 330
    using CDECL, 332

Assembly language *(continued)*
entry sequences, 323, 326
exit sequences, 329
listings, 256
local data, 323, 326
parameters, accessing, 325
passing by near reference, 331
procedures, 279
register usage, 324
return value, 327
routines in Quick libraries, 199
simplified segment directives, 321
ASSUME directive (Macro Assembler), 334
Asterisk (*) LIB command symbol, 231
Attribute, STATIC, 271
Auto Save command, 258
AUTO statement (BASICA), 240
Automatic
formatting, 116
variables
defined, 351
mixed-language programs, 324
AUX (device name), 213

/b option (qb), 49
Backing up disks, 7
.BAS
extension, 56
file, execution by QuickBASIC, 241
Base name, defined, 351
BASIC
books about, xxvii
calling convention, 330
initialization, 299
passing
by far reference, 303
by near reference, 302
by value, 302
return value, 327
BASICA
compatibility, 239
compiler, differences from, 27
QuickBASIC
converting to, 239
differences from, 29
starting, 19
/BATCH option (LINK), 220
Batch file, when to use, 206
bc command
differences among versions, 263
file-name usage, 208
invoking from command line, 207
options
/a, 210, 338
bc command *(continued)*
options *(continued)*
/ah, 210, 263, 263, 338
/c, 210, 338
/d, 210, 262, 338
/e, 210, 262, 338
list of, 210
/mbf, 210, 338
/o, 211, 338
/r, 211, 262, 338
/s, 211, 262, 338
table of, 262, 338
/v, 211, 262, 338
/w, 211, 262, 338
/x, 211, 262, 338
/zd, 211, 338
/zi, 211, 338
BC.EXE
described, 8, 149, 205
environment-variable setting, 13
BCOM40.LIB
advantage of, 152
described, 8, 149
invocation, 152
Bibliography, xxvii
Binary
defined, 352
file access
difference among versions, 253
OPEN statement syntax, 271
Binary-format programs, 24
BIOS, defined, 351
BLOAD statement, 241
Bold text, indicating keywords, xxv
B_OnExit routine, 319
BQLB40.LIB, 8
Breakpoints
described, 168
using, 172, 175
Breaks, checking at run time, 153
BRUN40.EXE
advantage of, 152
described, 8
BRUN40.LIB
default for linker, 214
described, 149
invocation, 152
run-time module library, 8
BSAVE statement, 241
Bugs, prevention, 166. *See also* Debugging
BUILDLIB utility, 263
Buttons
Cancel, Change dialog box, 128
Change All, 128
command, illustrated, 74

Buttons *(continued)*
mouse, 70
BYVAL statement, mixed-language programs, 288, 302

/c option
bc, 210
qb, 50
C language
arrays, 312
BASIC, calls from, 290
BASIC, calls to, 297, 299
default pointer size, 304
extern statement, 297
far keyword, 298
fortran keyword, 297, 298
functions, 279
/Gc compiler option, 297
memory models, 284
naming convention, 280
near keyword, 298
parameter-passing defaults, 284, 303
pascal keyword, 297, 298
passing
by far reference, 297, 303
by near reference, 297, 303
by value, 303
pointers, 297, 304, 316
routines in Quick libraries, 199
structures, 315, 316
type declarations, 297
CALL ABSOLUTE statement, 198
Call by reference. *See* Passing by reference
Call by value. *See* Passing by value
CALL INT86OLD statement, 198
CALL INTERRUPT statement, 198, 351
CALL statement
DECLARE, used with, 268
mixed-language programs, 278
optional use of, 268
QuickBASIC/interpreter differences, 241
Calling
assembly from BASIC, using CDECL, 332
conventions, mixed-language, 282
Calls
menu, 180
stack, 181
CALLS statement, mixed-language programs, 302, 303
Cancel command button, Change command, 128
Canceling
Case, Find command, 124
Case significance, linker options, 218, 226
Cassette BASIC, 240
CDECL statement
BASIC calls to assembly language, 332
mixed-language programs, 278, 287
CHAIN statement
BCOM40.LIB, 152
BRUN40.LIB, 152
QuickBASIC/interpreter differences, 241
Change All command button, 128
Change command
Active Window option, 127
All Modules option, 127
Cancel, 128
Cancel button, 128
Change All option, 128
Change button, 128
Current Module option, 127
dialog box, 126
errors, correcting, 128
Match Upper/Lowercase option, 127
Search options, 127
Skip button, 128
Whole Word option, 127
Change To text box, Change command, 127
Changing
active window, 78
window size, 78
Check boxes
defined, 352
illustrated, 74
Checking between statements option, 262
Choosing
commands
defined, 352
differences among versions, 257
ENTER key, 70
keyboard, 69
menus, 69
mouse, 70
options, differences among versions, 257
Clause, defined, 352
Clear All Breakpoints command, 175
Clear command, 118
CLEAR statement, differences among versions, 268
Clearing
breakpoints, 175

Clearing *(continued)*
screens, 68
Click, defined, 352
Clipboard
defined, 117, 352
large text blocks, 117
CLNG function, 268
Closing menus, ESC key, 69
CLS statement, 268
/cmd option (qb), 50
.CODE directive (Macro Assembler), 322
CodeView debugger
compatibility with BC, 182
conventions for procedure calls, 321
interlanguage calls, 182
linker option for, 226
QuickBASIC and, 46
/CODEVIEW option (LINK), 226
Color monitors
command names on, 70
starting QuickBASIC, 21, 34
COLOR statement, 268
Column-major order, in arrays, 313
COM statement, 214
Command buttons, 74
COMMAND$ function, 48, 148
Command line
arguments
COMMAND$ function, 48
processing in QuickBASIC, 148
creating from Quick library, 198
Commands
Auto Save, 258
bc. *See* bc command
canceling
keyboard, 70
mouse, 70
Change, 126
choosing
ENTER key, 70
keyboard, 69
menus, 68, 69
mouse, 70
Clear, 118
Compile, 260
Copy, 118
Create File, 102
Cut, 118, 120
debugging, table of, 344
defined, 353
Delete All Watch, 170
Delete Watch, 170
differences among versions, 258
displaying, 69
Commands *(continued)*
DOS
DIR, 39
DISKCOPY, 7
PATH, 14
SET, 15
VER, 12
DOS Shell, 80
editing, summary, 134
editing, table of, 345
Errors, 260
Exit, 81
Find, 122
Include Files, 103
Included Lines, 103
Label, 125
LINK. *See* Linker
Load, 258
Load Module, 91, 133
Make EXE File, 150, 260
menus
on-line Help, 52
table of, 339
Merge, 107, 132
Modify COMMAND$, 148
New FUNCTION, 156
New Program, 88
New SUB, 156
Next Error, 260
Next Statement, 180
Next SUB, 159
Open Program, 88
Output Screen, 143
Paste, 118, 120
Print, 109
qb, 262. *See also* qb command
Quit, 258
Repeat Last Find, 125
Save, 92
Save All, 92, 99
Save As, 92
Selected Text, 125
Set Main Module, 100
Set Next Statement, 179
Shell, 258
shortcut keys, 71, 339, 342
Split, 77
Start, 25
starting QuickBASIC, 49
SUBs, 36, 76, 98, 158
Syntax Checking, 114
Undo, 119
Unload Module, 101
Common blocks, in mixed-language programs, 316

COMMON statement
  AS clause, 268
  BCOM40.LIB, 152
  BRUN40.LIB, 152
  QuickBASIC/interpreter differences, 241
Compact memory model, 284, 322
Compatibility
  among versions, files, 272
  BASICA and GW-BASIC, 239
Compile
  command, 260
  options, 262
  time, defined, 353
Compiler
  defined, 353
  differences from BASICA, 27
  differences from interpreter, 40, 46
Compile-time errors. *See* Errors
Compiling
  defined, 353
  DOS, and linking from, 43, 206
  mixed-language programs, 284
CON (device name), 213
CONST statement, 268
Constants
  defined, 353
  global, defined, 355
  local, defined, 357
  symbolic, defined, 364
CONT statement (BASICA), 240
Context
  debugging, 168
  defined, 353
  execution, 168
  Immediate window, 169
  line on screen, 67
  watched expression, 176
Context-sensitive Help, 54
Continue command, 178
Control commands, execution, 178
Controlling
  linker, 218
  segments, 222
  stack size, 227
Converting
  BASICA and GW-BASIC programs, 239
  data files, 247
  IEEE format, program for, 251
Coordinates
  logical, defined, 358
  physical, defined, 360
Copy command, 118
Copying, text, 118
/CPARMAXALLOC option (LINK), 226
Create File command
  differences among versions, 258
  illustrated, 96
  include files, 102
Creating
  executable files, 28, 38, 150
  FUNCTION procedures, 156
  main module, 140
  multiple-module programs, 160
  object files, 43
  SUB procedures, 156
Cross-reference listing, 233
CTRL key, entering special control sequence characters, 130
CTRL+BREAK
  checking, 153
  Debug option, 153
  enabling, 153
CTRL+Y, deleting text with, 119
Current Module option
  Change command, 127
  Find command, 124
  Print command, 109
Cursor
  defined, 353
  illustrated, 66, 68
Cut command, 118, 120
CVDMBF function, 250, 269
CVL function, 268
CVSMBF function, 250, 269

/d option (bc), 210, 262
.DATA directive (Macro Assembler), 322
Data address size, 285
Debug
  menu
    Add Watch command, 176
    Clear All Breakpoints command, 175
    Delete All Watch command, 341
    Delete Watch command, 170
    History On command, 172
    illustrated, 170
    Set Next Statement command, 179
    Toggle Breakpoint command, 175
    Trace On command, 172
    Watchpoint command, 175
  option, 260, 262
Debugging
  breakpoints, 168, 172, 175
  /CODEVIEW linker option, 226
  commands, table of, 344
  compile-time errors, 40, 46
  context

Debugging *(continued)*
context *(continued)*
illustrated, 173
using watchpoints, 175
differences among versions, 264
edit and continue, 179
features, 168, 173
general information, 165
History On command, 172
Immediate window, 167
interactive, 40, 46
keys, differences among versions, 265
mixed-language programs, 182, 276
reinitializing variables, 180
restart for stepping, 180
single mouse command, 179
tracing, 168, 171
watch variable, 168
watching values during, 176
watchpoints, 175
while writing code, 166
DECLARE statement
AS clause, 268
description, 286
differences among versions, 269
mixed-language programs, 278
procedure declarations, 159
Declaring, procedures, 159
DEF FN function, defined, 353
DEF USR statement (BASICA), 240
Default pointer size, C, 304
Default segment names, 333
DEFDBL statement, 240
DEFINT statement, 240
DEFLNG statement, 269
DEFSNG statement, 240
DEFSTR statement, 240
DEF*type*
defined, 353
QuickBASIC/interpreter differences, 241
Delete All Watch command
illustrated, 170
table, in, 341
DELETE statement (BASICA), 240
Delete Watch command
illustrated, 170
table, in, 341
Deleting
modules, 101
text, 118, 119
Delivering Quick libraries to end users, 199
Devices, cassette, 240
DGROUP (Macro Assembler), 334
Dialog boxes
Change command, 126
clearing, 68
defined, 23, 354
Find command, 123
illustrated, 73, 74
Load File command, 95
Make EXE File, illustrated, 150
Merge command, 107
mouse, using with, 75
Open Program command, 89
Print command, 109
Save As command, 93
selecting items, 75
text box, defined, 75
Differences among versions
file compatibility, 272
table of, 246
DIM statement
AS clause, 268
QuickBASIC/interpreter differences, 241
TO clause, 269
Dimension, defined, 354
DIR command (DOS), 39
Directories, changing from within QuickBASIC, 91
Disk contents
distribution disks, 7
listing, 90
DISKCOPY command (DOS), 7
Disks, backing up, 7
Displaying
commands on menus, 69
modules, 98
Documents
defined, 354
loading, 91
DO...LOOP statement, 269
DOS
books about, xxviii
commands. *See* Commands
compiling and linking from, 43, 206
creating
Quick library, 198
stand-alone programs, 206
environment variables, 13
linking from command line, 212
running programs from, 28, 29, 148
DOS Shell command, 80
/DOSSEG option (LINK), 227
Double precision, defined, 354
Double-precision real numbers, mixed-language programs, 305
Drag, defined, 354
DRAW statement, 241

$DYNAMIC, defined, 354
Dynamic variables, in mixed-language programs, 324

/e option (bc), 210
Edit and continue, while debugging, 179
Edit menu
  Clear command, 118
  Copy command, 118
  Cut command, 118, 120
  New FUNCTION command, 156
  New SUB command, 156
  Paste command, 118, 120
  Syntax Checking command, 114
  Undo command, 119
EDIT statement (BASICA), 240
Editing
  commands, table of, 134, 345
  differences among versions, 261
  include files, 103
  modules, 159
  procedures, 159
Editor
  automatic formatting, 116
  clipboard, 117
  selecting text, 117
  syntax checking, 114
Emulation, math coprocessor, 12, 15
ENTER key, equivalent to SPACEBAR, 257
Environment variables
  LIB, 15, 216
  LINK, 218
  NO87, 15
  PATH, 14
  setting, 13
  TMP, used by LINK, 217
EQU directive (Macro Assembler), 326
ERROR statement, 147
Error location, Produce Debug code option required, 153
Error messages
  clearing, 68
  redirection, 256
  run-time, 143
Error-handling messages, 151
Error-message window, 257
Errors
  command, 260
  compile-time, debugging, 40, 46
  run time, 147, 153
  syntax, checking, 140
  when running a program, 143
ESC key, 69
Event Trapping option, 262
Event trapping, Quick library, 195
.EXE
  extension, 56
  files. *See* Executable files
EXE requiring BRUN40.EXE option, 151, 152
Executable files
  creating, 28, 38, 148, 150
  defined, 27, 354
  files needed, 149
  making compact, 201
  packing, 221
  Quick libraries, 151
  running from DOS, 39
Execute
  defined, 354
  option (qb), 49
Execute to the Cursor command (F7), 179
Execution
  Continue command, 178
  control commands, 178
  preset points, between, 175
  suspending with watchpoints and breakpoints, 175
  to the cursor, 179
  unreachable code, 179
  unset points, between, 179
/EXEPACK option (LINK), 221
Exit command, 81
EXIT DEF statement, 269
EXIT DO statement, 269
EXIT FOR statement, 269
EXIT FUNCTION statement, 269
EXIT SUB statement, 269
Exiting QuickBASIC, 81
Extensions
  BC and LINK usage, 209
  defined, 355
  file name, 56
extern statement, C language, used in, 297
Extra files produced by QuickBASIC, 198

Far address, defined, 355
far keyword, C language, 298
Far reference parameters
  BASIC, 288, 303
  C language, 297, 303
File
  menu
    Create File command, 96
    DOS Shell command, 80
    Exit command, 81

File *(continued)*
menu *(continued)*
illustrated, 68
Load File command, 91, 133
Merge command, 107
New Program command, 88
Open Program command, 88
Print command, 109
Save All command, 92, 99
Save As command, 92
Save command, 92
Unload File command, 101
names
extensions, 56
specifying, 57
without extensions, 57
FILEATTR function, 269
File-name extensions
BC and LINK usage, 209
described, 56
File-naming conventions
bc command, 208
described, 55
linker, 215
Quick library, 198
Files
adding to program, 132
ASCII format, 239
backing up, 7
binary, defined, 352
compatibility among versions, 272
executable
creating, 149
defined, 354
with Quick libraries, 151
extra, produced by QuickBASIC, 198
include, defined, 356
inserting in active window, 107
loading, 90
.MAK, 100
map
creating, 225
described, 223
listing formats, 223
/MAP option (LINK), 223
merging, 107
object, creating, 43
object, defined, 359
required, list, 9
source, defined, 363
stand-alone, defined, 363
Find and Verify button, 128
Find command
Active Window option, 124
All Modules option, 124
Current Module option, 124
Find command *(continued)*
described, 122
dialog box, 123
Find What text box, 124
Match Upper/Lowercase option, 124
Search options, 124
Whole Word option, 124
Find What text box
Change command, 127
Find command, 124
Finding text, 122
Fixed-disk setup, 9
FLAGS register, defined, 355
Flashing underscore. *See* Cursor
Floating-point
defined, 355
numbers, in mixed-language programs, 305
precision, within Quick libraries, 197
fortran keyword
in C language, 297
rules of syntax, 298
Forward slash (/), linker option character, 218
Framepointer, 323
FREEFILE function, 269
FUNCTION
defined, 355
procedures
creating, 156
illegal in include files, 155
mixed-language programs, 279
statement
AS clause, 268
breaking program into units, 31, 41, 47
STATIC attribute, 271
FUNCTION...END FUNCTION statement, 269
Functions
BASICA, prohibited by QuickBASIC, table of, 240
CLNG, 268
conversion option (qb), 50
CVDMBF, 250, 269
CVL, 268
CVSMBF, 250, 269
FILEATTR, 269
FREEFILE, 269
interpreter, USR, 240
LCASE$, 270
LEN, 270
LTRIM$, 270
mixed-language programs, 279
MKDMBF$, 250, 270
MKL$, 270

Functions *(continued)*
MKSMBF$, 250, 270
prohibited by QuickBASIC, 240
RTRIM$, 270
SETMEM, 271
UCASE$, 270
VARPTR$, 241
VARPTR
differences among versions, 272
mixed-language programs, 312, 316
VARSEG
differences among versions, 272
mixed-language programs, 312, 316

/g option (qb), 49
/Gc option, mixed-language programs, 297
GET statement, user-defined records, 270
Global
constants, defined, 355
variables, defined, 355
GROUP directive (Macro Assembler), 334
GW-BASIC, 239

/h option (qb), 50, 263
Hard-disk setup, 9
Heap, defined, 356
Help
accompanying QuickBASIC menu commands, 52
BASIC syntax, 54
context sensitive, 54
general, 53
invoking from screen, 67
invoking on line, 67
menu
Close Help command, 341
commands, 341
context-sensitive, 54
general, 53
General command, 341
Topic command, 341
/HELP option (LINK), 219
Highlight
defined, 356
moving, 70
High-resolution-display option (qb), 50, 263
History On command, 168, 172
Huge memory model, 284

IEEE format
accuracy, 248
converting to, 247, 250
defined, 356
differences among versions, 247
exponential numbers, printing, 249
/mbf option, using with old programs, 249
number ranges, 248
numbers
additional range and accuracy, 31, 41, 48
differences from Microsoft Binary, 248
printing numbers, 249
IF statement, 31, 41, 48
Ignoring case, 226
Immediate window
calculations, 145
calling procedures from, 146
capacity, 144
changing, active window, 78
context, 169, 173
defined, 356
described, 79
illustrated, 66, 68
position, 76
preventing bugs, 167
run-time error messages, 167
statements not allowed, 145
statements not supported, 145
testing ideas in, 143
testing programs, 40, 46, 145
window size, changing, 78
Include Files command, 103
Include files
defined, 356
editing, 103
loading, 91
procedures not allowed in, 264
$INCLUDE metacommand, 104
Included Lines command, 103
Indenting text, 131
/INFORMATION option (link), 220
Input focus, defined, 356
Insert mode
described, 116
differences among versions, 255
Inserting text, 118
Installing QuickBASIC
floppy disks, 10
hard disk, 9
INT86OLD, 197
Integers
defined, 356
long, 31, 41, 48

Integers *(continued)*
mixed-language programs, 305
Integrated programming environment, 45
Interactive debugging, 40, 46
Interlanguage calls. *See* Mixed-language programs
Interpreted BASIC, compatibility, 239
Interpreter
See also BASICA
defined, 356
differences from compiler, 40, 46
Interprocedure calling, 180
INTERRUPT, 197
Interrupt support routines, 197
Italic text, showing placeholders, xxv

Joining lines, 132

Keyboard, choosing commands from, 69
Keys
debugging
differences among versions, 265
table of, 344
editing, differences among versions, 261
ENTER, 257
shortcut
differences among versions, 258
executing commands directly, 69, 71
table of, 339, 342
SPACEBAR, 257
Keywords
capitalization of, 116
defined, 357

/l option (qb), 50, 196, 197
Label command, 125
Language differences among versions, 266
Large memory model, 284, 322
LCASE$ function, 270
LEN function, 270
.LIB
extension, 56
libraries. *See* Libraries, stand alone
LIB
combining libraries, 231
command symbols
asterisk (*), 231
minus sign (–), 231, 232
LIB *(continued)*
command symbols *(continued)*
minus sign-asterisk (–*), 231
minus sign-plus sign (–+), 231, 232
plus sign (+), 231, 232
default responses, 230
described, 227
environment variable, 15, 196, 216
invoking, 228
library modules
adding, 231
deleting, 231
extracting, 231
extracting and deleting, 231
replacing, 231
listing files, 233
options
page size, specifying, 233
/PAGESIZE (/P), 234
response file, 228
LIB.EXE, described, 8, 205
Libraries
See also Quick libraries
default, ignoring, 216, 222
defined, 357
described, 189
search path, 216
specifying for LINK, 216
stand alone
combining, 231
creating parallel library, 199
defined, 189
deleting object modules, 231
described, 198
executable files, 151
extracting and deleting modules, 231
LIB command line, 230
listing, 233
object modules, including, 231
object modules, replacing, 231
standard places, 216
types contrasted, 189
user. *See* Quick library
Library Manager. *See* LIB
Line and column counters, illustrated, 66
Line label, finding, 125
Line styling, defined, 357
Line-number checking, 153
/LINENUMBERS option (LINK), 225
Lines, joining, 132
link command, 215. *See also* Linker
LINK environment variable, 218
Link time, defined, 357

Linker
See also Libraries
BCOM40.LIB, advantage of, 152
BRUN40.EXE, advantage of, 152
defaults, 214
invoking from command line, 212
mixed-language programs, 285
options
abbreviations, 218
/BATCH (/B), 220
case sensitivity, 218, 226
/CODEVIEW (/CO), 226
/CPARMAXALLOC (/CP), 226
create Quick library, 221
debugging with CodeView debugger, 226
displaying, 219
displaying line numbers, 225
displaying process information, 220
/DOSSEG (/DO), 227
EXEPACK (/E), 221
/HELP (/HE), 219
ignoring default libraries, 216, 222
/INFORMATION (/I), 220
/LINENUMBERS (/LI), 225
LINK environment variable, 218
map file, 223
/MAP (/M), 223
/NODEFAULTLIBRARYSEARCH (/NOD), 216, 222
/NOIGNORECASE (/NOI), 226, 280
/NOPACKCODE (/NOP), 222
/NOPACKCODE (/PAC), 225
numerical arguments, 218
order on command line, 218
ordering segments, 227
paragraph space, allocating, 226
/PAUSE (/P), 219
pausing, 219
preventing linker prompting, 220
/QUICKLIB (/Q), 221
segments, 222
/SEGMENTS (/SE), 222
setting stack size, 227
/STACK (/ST), 227
unsuitable for BC, 227
Quick library, other-language routines, 199
response file, 212, 215
specifying libraries, 216
temporary output file, 217, 220
LINK.EXE, described, 8, 149, 205
Linking
See also Linker
defined, 357
Linking *(continued)*
from DOS, 44
List boxes
defined, 357
scroll bar, illustrated, 74
LIST statement (BASICA), 240
Listing files, cross-reference (LIB), 233
LLIST statement (BASICA), 240
Load File command, 91, 133, 258
LOAD statement (BASICA), 240
Loading
defined, 357
documents, 91
files, 90
include files, 91
modules, 91
programs, 21, 88
Quick libraries, 196
QuickBASIC Version 2.0 or 3.0 programs, 34
Local constants, defined, 357
Local variables, defined, 358
Logical coordinates, defined, 358
Long integers
advantages of, 31, 41, 48, 252
converting to, 268
defined, 358
LSET statement, differences among versions, 270
.LST files. *See* Listing files
LTRIM$ function, 270

Machine code, defined, 358
Macro Assembler. *See* Assembly language, Mixed-language programs
Main module
creating, 140
defined, 99, 358
.MAK files
described, 100
Quick library considerations, 196
saving, 100
Make EXE and EXIT command button, 151
Make EXE command button, 151
Make EXE File dialog box, illustrated, 150
Make Library and Exit command, 195
Make Library command, 194, 195
.MAP files. *See* Map files
Map files
creating, 223, 225
extension, 225
format, 223

Map files *(continued)*
/MAP option, (LINK), 223
program entry point in, 224
segment lists in, 223
symbol tables in, 223
/MAP option (LINK), 223
Match Upper/Lowercase option
Change command, 127
Find command, 124
Math coprocessor, 12, 15, 358
Maximize box, illustrated, 66
/mbf option
bc, 210
bc or qb, 247, 249, 263
qb, 50
Memory
calculating available, 201
compiling in, 143
map, defined, 358
minimum configuration, xxiii
models
assembly-language procedures, 322
C module, 298
compact, 322
in mixed-language programs, 284
large, 322
medium, 284, 322
small, 322
requirements
Quick libraries, 200
QuickBASIC, to install, xxiii
Menu bar
defined, 359
illustrated, 66, 68
Menus
Calls, 180
clearing, 68
closing, 69
commands
choosing, 68
differences among versions, 258
on-line Help, 52
table of, 339
Debug, 170
defined, 22
differences among versions, 257
displaying, 69
Edit, 118
File, 68
Help, 53
Run, 149
Search, 121
selecting, 69
View, 103
Merge
command, 107, 132
Merge *(continued)*
dialog box, 107
MERGE statement (BASICA), 240
Merging files, 107
Metacommands
defined, 359
$DYNAMIC, 354
$INCLUDE, 104
$STATIC, 363
Microsoft
Binary format
defined, 359
differences from IEEE format, 248
Library Manager. *See* LIB
Mouse, using, 12
segment model, 333
Minimize string data option, 262
Minus sign-asterisk (-*), LIB command
symbol, 231
Minus sign-plus sign (-+), LIB
command symbol, 231
Mixed-language programs
arrays, 311
assembly language, calls with
CDECL, 332
BASIC
calls from C, 297, 299
calls to C, 286, 290
parameter-passing defaults, 284, 302
C language, parameter-passing
defaults, 303
calling
assembly-language routines, 330
conventions, 282
common blocks, 316
compiling, 284
debugging, 182, 276
DECLARE, use of, 286
defined, 275
differences in data formats, 304
linking, 285
naming conventions
ALIAS feature, 287
defined, 280
type-declaration characters, 287
parameter
list, 288
passing, 283
passing
arrays, 311
by far reference, 288
by near reference, 300, 330
by value, 288, 302
strings, 308, 310
Quick libraries, 199

Mixed-language programs *(continued)*
string formats, 306
type declaration characters, 287
variables, 316
MKDMBF$ function, 250, 270
MKL$ function, 270
MKSMBF$ function, 250, 270
.MODEL directive (Macro Assembler), 322
Modes
insert, 116
overtype, 116
Modify COMMAND$ command, 25, 148, 340
Module-level code, defined, 359
Modules
defined, 87, 359
deleting, 101
displaying, 98
editing, 159
loading, 91
main
creating, 140
defined, 99, 358
memory, in, 41, 47
multiple, 87
Monitors
color, starting QuickBASIC, 21
monochrome
command names on, 70
starting QuickBASIC, 21, 34
use with Color Graphics Adapter, 49
MOTOR statement (BASICA), 240
Mouse
buttons used, 70
click, defined, 352
commands
canceling with, 70
choosing with, 70
debugging, during, 179
defined, 359
dialog boxes, using with, 75
drag, defined, 354
loading files with, 90
pointer, illustrated, 66, 68
scrolling with, 71
using, 12
MOUSE.COM, 8, 12
Moving
highlight, 70, 70
text, 120
Multiple modules
differences among versions, 254
saving, 99

Naming conventions
files, in QuickBASIC, 55
mixed-language programs, 280
Near address, defined, 359
near keyword, C language, 298
Near reference parameters
assembly language, 331
BASIC, 300, 302, 330
C language, 297, 303
New FUNCTION command, 156
New Program command
differences among version, 258
modules, creating, 95
starting new program, 88
table, in, 339
NEW statement (BASICA), 240
New SUB command, 156
Next Error command, 260
NEXT statement, 272
Next Statement command, 180
Next SUB command, 159
NO87 environment variable, 15
NOCOM.OBJ file, 214
/NODEFAULTLIBRARYSEARCH option (LINK), 216, 222
/NOIGNORECASE option (LINK), 226, 280
/NOPACKCODE option (LINK), 222
Not watchable
described, 175
while watching expressions, 176
Notational conventions, xxv
NUL (device name), 213
NUL.MAP file, 214
Numbers, IEEE format, 31, 41, 48

/o option (bc), 211
.OBJ files. *See* Object files
Object
files
compatibility among versions, 272
creating, 43
defined, 359
described, 148
modules
library, deleting from, 231
library, including in, 231
listing, 233
Offset, defined, 360
On-line Help, 52
Open, defined, 360
Open Program command
described, 88
dialog box, 89
list box, listing files with, 90

Open Program command *(continued)*
list box, loading files with, 89
OPEN statement, 271
Opening
files, 88
screen, illustrated, 68
Options. *See* bc command, Linker, qb command
Options boxes, illustrated, 74
Other-language routines. *See* Mixed-language programs
Output Screen command, 143
Outputting
high-order ASCII characters, 129
low-order ASCII control sequences, 130
Overflow, defined, 360
Overlay linker. *See* Linker
Overtype mode
described, 116
differences among versions, 255

/PAC option (LINK), 225
Packed storage, in mixed-language programs, 315
PACKING.LST, 8
Page, defined, 360
Page size, library, 233
/PAGESIZE option (LIB), 234
PALETTE statement, 268
Paragraph space, 226
Parameter type declarations, in C, 297
Parameters
assembly language, accessing from, 325
calling conventions, effect of, 282
defined, 360
list, in mixed-language programs, 288
passing, in mixed-language programs, 283
pascal keyword, 297, 298
Passing
arrays, 311
by far reference
BASIC, 288, 303
C language, 297, 303
described, 283
by near reference
assembly language, 331
BASIC, 300, 302, 330
C language, 297, 303
described, 283
by reference, defined, 360
by value
BASIC, 288, 302
Passing *(continued)*
by value *(continued)*
C language, 303
defined, 360
described, 283
common blocks, 316
order of, 282
strings
from BASIC, 308
from C, 310
Paste command, 118, 120
PATH environment variable, 14
/PAUSE option (LINK), 219
Pausing program execution, 175
Physical coordinates, defined, 360
Place markers
moving to, 129
setting, 129
PLAY statement, QuickBASIC/interpreter differences, 241
Plus sign (+), LIB command symbol
specifying library, 232
using, 231
Pointers
defined, 361
specifying, 297, 304, 316
Print command
Active Window option, 109
All Open Files option, 109
Current Module option, 109
described, 109
dialog box, 109
Selected Text option, 109
Printing, described, 109
PRN (device name), 213
Procedures
calling from Immediate window, 146
declaring, 159
defined, 361
editing, 159
mixed-language programs, 279
Procedure-step tracing, 171
Produce Debug Code option, 151
Programming environment
differences in versions, 256
integrated, 45
Programs
BASICA, GW-BASIC, converting to QuickBASIC, 239
binary format, 24
defined, 361
exiting, 81
loading, 21, 88
multiple module, creating, 160

Programs *(continued)*
QuickBASIC Version 2.0 or 3.0, loading, 34
running
changing variable values in, 146
described, 24
DOS, 28, 29, 39
QuickBASIC, outside, 148
subordinate, placement in windows, 36
testing, 145
translating, 24
typing, 140
Prohibited BASICA functions in QuickBASIC, table of, 240
ProKey, using QuickBASIC with, 254
PUBLIC directive (Macro Assembler), 322
PUBLIC symbol, 361
Public symbols, listing, 233
PUT statement, user-defined records, 270

qb command
/ah option, 50, 263, 337
arguments, 49
/b option, 49, 337
/c option, 50, 337
/cmd option, 49, 50, 337
differences among versions, 262
/g option, 49, 337
/h option, 50, 263
/l option, 50, 196, 197, 337
/mbf option, 50, 263, 337
options, list of, 49
options, table of, 337
/run option, 49, 196, 263, 337
syntax, 49
QB.EXE
described, 8
environment-variable setting, 13
QB.HLP, 8
QB.INI, 52
QB.LIB, 8
QB.QLB
automatic loading, 197
described, 8
library, 196
.QLB extension, 56
QLBDUMP.BAS, 197
Question mark (?), shortcut for PRINT keyword, 145
Quick libraries
advantages, 190
CALL ABSOLUTE statement, 198
Quick libraries *(continued)*
CALL INT86OLD statement, 198
CALL INTERRUPT statement, 198
compatibility among versions, 272
contents, 191
creating
command line, from, 198
described, 191
files needed, 192
QuickBASIC, from within, 193
using linker, 221
default, 196, 197
defined, 361
delivery to end user, 199
described, 41, 47, 189
executable files, making compact, 201
file naming, 195, 198
files needed to create, 192
floating point precision, 197
listing contents, 197
loading, 196, 197
memory requirements, 200
other files produced by, 198
other-language routines, 192, 199
search path, 196
updating .MAK file, 196
updating previous Quick library, 193
using with executable files, 151
QuickBASIC
differences from BASICA, 29
programs, running, 38
Save commands, 93
Version 2.0 or 3.0, experience with, 33
QuickBASIC—Fast Load and Save option, 94
/QUICKLIB option (LINK), 221
Quick-library loading option (qb), 50
Quit command, 258
Quitting QuickBASIC, 81

/r option (bc), 211, 262
README.DOC, 8
Real numbers, in mixed-language programs, 305
Receiving parameters, in mixed-language programs, 282
Records, defined, 361
Recursion, 31, 41, 48
Registers, defined, 362
Reinitializing variables, 180
Remote-reception buffer-size option (qb), 50
RENUM statement (BASICA), 240

Repeat Last Find command, 125
Replacing text, 126
Response file
  example, 215
  LIB, 228
  linker, 212
Restart command, 180
Resume Next option, 262
RESUME statement,
    QuickBASIC/interpreter
    differences, 241
RETURN statement
  Debug option, 153
  GOSUB, checking for, 153
Return value, offset, 327
RETURN without GOSUB error,
    cause, 153
Routine, B_OnExit, 319
Row-major order, in arrays, 313
RTRIM$ function, 270
/run option (qb), 49, 196, 263
Run, defined, 362
Run menu
  Continue command, 178
  illustrated, 25
  Make EXE File command, 150
  Make Library command, 194, 195
  Modify COMMAND$ command, 148
  Restart command, 180
  Set Main Module command, 100
  Start command, 25
RUN statement, 241
Run time
  defined, 362
  error messages, 143
  errors
    checking, 153
    hint regarding, 167
    reporting locations, 151
    simulating, 147
  module, defined, 362
Running programs
  described, 24
  from DOS, 39
  QuickBASIC, outside , 148
  QuickBASIC, within, 38

/s option (bc), 211, 262
Save All command, 92, 99
Save As command
  described, 92
  dialog box, 93
  Fast Load and Save option, 94
  QuickBASIC option, 93
Save command
  described, 92
  Text option, 94
SAVE statement (BASICA), 239, 240
Saving
  files, 92
  .MAK files, 100
  multiple modules, 99
  programs
    ASCII format, 239
    description, 92
Scope, defined, 362
Screen
  clearing, 68
  options, setting, 51
  QuickBASIC, described, 66
SCREEN statement, 268
Screen-update-speed option (qb), 49
Scroll
  arrows, illustrated, 66
  bars
    defined, 362
    illustrated, 66, 68
  box, defined, 71
  box, illustrated, 66
Scrolling
  defined, 362
  keyboard commands, 79
  mouse, 71
Search
  menu
    Change command, 122
    described, 121
    Find command, 122
    Label command, 125
    Repeat Last Find command, 125
    Selected Text command, 125
  options
    Change command, 127
    Find command, 124
  path
    libraries, 216
    Quick library, 196
SEEK statement, 271
SEG statement, in mixed-language
    programs, 288
SEGMENT directives (Macro
    Assembler), 334
Segment
  directives, simplified, 321
  lists, in map files, 223
  packing, 222
/SEGMENTS option (LINK), 222
Segments
  far addresses, 334
  number allowed, 222

Segments *(continued)*
order, 227
SELECT CASE statement
described, 31, 48
differences among versions, 271
Selected Text
command, 125
option, Print command, 109
Selecting
defined, 362
menus, 69
text, 117
Semicolon (;), LIB command symbol, 230
Separate-compilation method. *See* bc command
SET command (DOS), 15
Set Main Module command
described, 100
differences among versions, 259
illustrated, 25, 100
listed, 340
Set Next Statement command, 179
SETMEM function, 271
Setting up QuickBASIC
floppy disks, 10
hard disk, 9
SETUP.BAT, 8, 9
SHARED statement, AS clause, 268
Shell command, 258
Shortcut keys
commands, 71
debugging commands, 344
described, 69
differences among versions, 258
menu commands, table of, 339
Shortcut keys, table of, 342
SideKick, using QuickBASIC with, 254
Simple variables, defined, 362
Simplified segment directives, 333
Single precision, defined, 363
Single-precision real numbers, in mixed-language programs, 305
Single-step tracing, 171
Small memory model, 284, 322
Smart editor. *See* Editor
Source code, 143
Source files
compatibility among versions, 272
defined, 363
format, 239
Special characters, methods for entering, 129
Split command, 77
Split Window command button, 159
/STACK option (LINK), 227
Stack
defined, 363
frame, 323, 325
size, setting, 227
variables, in mixed-language programs, 324
Stack Trace, CodeView feature, 321
Stand alone
files, defined, 363
libraries. *See* Libraries
programs, creating from DOS, 206
Stand-Alone EXE File option, 151
Standard places, libraries, 216
Start command
described, 25, 38
differences among versions, 260
Starting QuickBASIC
BASIC compiler users, 42
BASICA users, 19
color monitor, 21, 34
monochrome monitor, 21, 34
qb command, 49
QuickBASIC users, 33
Statements
ALIAS, 287
AS, in mixed-language programs, 289
BASICA, prohibited by QuickBASIC, table of, 240
BLOAD, 241
BSAVE, 241
BYVAL, in mixed-language programs, 288, 302
CALL, modification required, 241
CALLS, in mixed-language programs, 302, 303
CDECL, in mixed-language programs, 278
CHAIN
BCOM40.LIB, 152
BRUN40.LIB, 152
modification required, 241
CLEAR, differences among versions, 268
CLS, 268
COLOR, 268
COM, 214
COMMAND$, 148
COMMON
AS clause, 268
BCOM40.LIB, 152
BRUN40.LIB, 152
interpreter differences, 241
CONST, 268
DECLARE
AS clause, 268
differences among versions, 269

Statements *(continued)*
DECLARE *(continued)*
in mixed-language programs, 278, 286
procedure declarations, 159
DEF FN, 353
DEFDBL, 240
defined, 363
DEFINT, 240
DEFLNG, 269
DEFSNG, 240
DEFSTR, 240
DEF*type*, 241
DIM
AS clause, 268
interpreter differences, 241
TO clause, 269
DO...LOOP, 269
DRAW, 241
ERROR, 147
EXIT DEF, 269
EXIT DO, 269
EXIT FOR, 269
EXIT FUNCTION, 269
EXIT SUB, 269
FUNCTION, 41
AS clause, 268
described, 31, 47
FUNCTION...END FUNCTION, 269
GET, user-defined records, 270
GOSUB, 153
IF, 31, 48
interpreter
AUTO, 240
CONT, 240
DELETE, 240
EDIT, 240
LIST, 240
LLIST, 240
LOAD, 240
MERGE, 240
MOTOR, 240
NEW, 240
RENUM, 240
SAVE, 240
LSET, 270
NEXT, 272
not allowed in Immediate window, 144
OPEN, 271
PALETTE, 268
PLAY, 241
prohibited by QuickBASIC, 240
PUT, user-defined records, 270
requiring modification in QuickBASIC, 240
Statements *(continued)*
RESUME, 241
RETURN, 153
RUN, 241
SCREEN, 268
SEEK, 271
SEG, in mixed-language programs, 288
SELECT CASE
described, 31, 48
differences among versions, 271
SHARED, AS clause, 268
STATIC, defined, 363
SUB
AS clause, 268
described, 31, 47
TYPE
described, 31, 41, 47
differences among versions, 272
WEND, 272
WIDTH, 268, 272
$STATIC, defined, 363
STATIC
attribute, 271
defined, 363
Status bar
described, 68
illustrated, 66
Stepping
single
into procedures, 171
over procedures, 171
when debugging, 171
String, defined, 364
String format
BASIC, 306
C language, 307
String-handling functions.
See also Specific function
new, 270
Structures, C language, 315
SUB
defined, 364
procedures
creating, 156
illegal in include files, 155
placement in windows, 36
placing in include files, 264
statement, 31, 47
AS clause, 268
STATIC attribute, 271
Subordinate programs, placement in windows, 36
Subprograms
defined, 364
mixed-language programs, 279

Subprograms *(continued)*
placement in windows, 36
Subroutines
defined, 364
mixed-language programs, 279
SUBs command, 36, 76, 98, 158
SuperKey, using QuickBASIC with, 254
Suspending execution, watchpoints and breakpoints, 175
Switching
active window, 78
between environment and output screen, 143
Symbol tables, in map files, 223
Symbolic
constants, defined, 268, 364
debugger, 165
Syntax Checking command
defined, 114
differences among versions, 253
table, in, 339
turning off, 115
Syntax
errors, checking, 114, 140
notation
choices, xxv
optional items, xxv
placeholders, xxv
System requirements, xxiii

Tab stops, setting, 131
Testing, programs, 145
Text
boxes
defined, 364
illustrated, 74
copying, 118
deleting, 118
finding, 122
indenting, 131
inserting, 118
moving, 120
option, Save commands, 94
replacing, 126
selecting, 117
Tiling, defined, 364
Title bar
described, 68
illustrated, 66
TMP environment variable, used by LINK, 217
Toggle Breakpoint command, 175
Toggle, defined, 365
Trace On command, 171
Tracing
animated mode, 171
described, 168, 171
Tracking, values during debugging, 176
Translating
programs, 24
source code in memory, 143
Two's complement, defined, 365
TYPE statement
described, 31, 41, 47
differences among versions, 272
Type-declaration characters, 287, 289
Typeface
examples, xxv
key names, xxvi
keywords, xxv
placeholders, xxv
Types, user-defined, 315, 365
Typing, programs, 140
Typographic conventions, xxv

UCASE$ function, 270
Underflow, defined, 365
Undo command, 119
Unload File command, 101
Unpacked storage, mixed-language programs, 315
Unresolved external, 365
Unsigned integers, in mixed-language programs, 305
User libraries
User-defined types, 247, 315, 365
USR function (BASICA), 240
Utility, BUILDLIB, 263

/v option (bc)
described, 211
differences among versions, 262
Value parameters
BASIC, 288, 302
C language, 303
passing, 283
Values
changing in running programs, 146
watching during debugging, 176
Variables
address, in mixed-language programs, 316
automatic
defined, 351
mixed-language programs, 324
dynamic, in mixed-language programs, 324
global, defined, 355

Variables *(continued)*
local, defined, 358
scope, defined, 362
simple, defined, 362
VARPTR, function
differences among versions, 272
mixed-language programs, 312, 316
VARPTR$ function
DRAW, use with, 241
PLAY, use with, 241
VARSEG function
differences among versions, 272
in mixed-language programs, 312, 316
Version differences, file compatibility, 272
View menu
Include Files command, 103
Included Lines command, 103
Next Statement command, 180
Next SUB command, 159
Options command, 51
Output Screen command, 143
Split command, 77
SUBs command, 76, 98
View window
defined, 366
described, 76
illustrated, 66, 68
splitting, 77
VM.TMP file, 217

/w option (bc)
described, 211
differences among versions, 262
Watch
variable, described, 168
window
commands, 174
context of variables, 366
described, 76, 168
illustrated, 173
Watchable, defined, 366
Watching
expressions in Watch window, 168
variable values and expressions, 176
Watchpoint command, 174, 175
Watchpoints
described, 168, 175
distinguished from watch expressions, 175
WEND statement, differences among versions, 272
Whole Word option
Change command, 127
Find command, 124
WIDTH statement, 268, 272
Windows
active, 68, 78
defined, 366
differences among versions, 257
error-message, 257
Immediate
active window, changing, 78
calculations, 145
calling procedures from, 146
capacity, 144
defined, 356
described, 79
illustrated, 66, 68
position, 76
statements not supported, 145
testing ideas in, 143
testing programs, 145
window size, changing, 78
interpreter, 40, 46
size, changing, 78
View
defined, 366
described, 76
illustrated, 66, 68
splitting, 77
Watch, 76, 173
WordStar keys
similarity to, 261
table of, 134, 345
World coordinates. *See* Logical coordinates

/x option (bc)
described, 211
differences among versions, 262

/zd option (bc), 211
/zi option (bc), 211, 226

Microsoft Corporation
16011 NE 36th Way
Box 97017
Redmond, WA 98073-9717

Microsoft®

0987 Part No. 00618